Oracle® Application Development Framework

Developer's Guide For Forms/4GL Developers

10*g* Release 3 (10.1.3.0)

B25947-02

June 2008

Oracle Application Development Framework Developer's Guide For Forms/4GL Developers, 10*g* Release 3 (10.1.3.0)

B25947-02

Copyright © 2008, Oracle. All rights reserved.

Contributing 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

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Preface xxxvii

Audience xxxvii
Documentation Accessibility xxxvii
Related Documents xxxviii
Conventions xxxviii

Part I Getting Started with Oracle ADF Applications

1 Introduction to Oracle ADF Applications

1.1 Introduction to Oracle ADF 1-1
1.2 Framework Architecture and Supported Technologies 1-1
1.2.1 View Layer Technologies Supported 1-2
1.2.2 Controller Layer Technologies Supported 1-3
1.2.3 Business Services Technologies Supported by ADF Model 1-3
1.2.4 Recommended Technologies for Enterprise 4GL Developers 1-3
1.3 Declarative Development with Oracle ADF and JavaServer Faces 1-4
1.3.1 Declarative Data Access and Validation with ADF Business Components 1-4
1.3.2 Declarative User Interface Design and Page Navigation with JavaServer Faces 1-5
1.3.2.1 Declarative User Interface Design with JSF 1-5
1.3.2.2 Declarative Page Navigation with JSF 1-7
1.3.3 Declarative Data Binding with Oracle ADF Model Layer 1-8
1.3.4 Simple, Event-Driven Approach to Add Custom Logic 1-9
1.3.4.1 Simple-to-Handle Events in the Controller Layer 1-10
1.3.4.2 Simple-to-Handle Events in the Business Service Layer 1-11
1.3.4.3 Simple to Globally Extend Basic Framework Functionality 1-12
1.4 Highlights of Additional ADF Features 1-13
1.4.1 Comprehensive JDeveloper Design-Time Support 1-14
1.4.2 Sophisticated AJAX-Style Web Pages Without Coding 1-16
1.4.3 Centralized, Metadata-Driven Functionality 1-17
1.4.4 Generation of Complete Web Tier Using Oracle JHeadstart 1-17

2 Overview of Development Process with Oracle ADF and JSF

2.1 Introduction to the Development Process 2-1
2.2 Creating an Application Workspace to Hold Your Files 2-2
2.3 Thinking About the Use Case and Page Flow 2-3

2.4 Designing the Database Schema 2-5
2.5 Creating a Layer of Business Domain Objects for Tables 2-5
2.5.1 Dragging and Dropping to Reverse-Engineer Entity Objects for Tables 2-6
2.5.2 Adding Business Validation Rules to Your Entity Object 2-6
2.5.3 Defining UI Control Hints for Your Entity Objects 2-9
2.6 Building the Business Service to Handle the Use Case 2-9
2.6.1 Creating a Application Module to Manage Technicians 2-9
2.6.2 Creating View Objects to Query Appropriate Data for the Use Case 2-10
2.6.3 Using View Objects in the Application Module's Data Model 2-13
2.6.4 Testing Your Service 2-15
2.6.5 The Data Control for Your Application Module Enables Data Binding 2-17
2.7 Dragging and Dropping Data to Create a New JSF Page 2-18
2.8 Examining the Binding Metadata Files Involved 2-21
2.9 Understanding How Components Reference Bindings via EL 2-22
2.10 Configuring Binding Properties If Needed 2-23
2.11 Understanding How Bindings Are Created at Runtime 2-24
2.12 Making the Display More Data-Driven 2-24
2.12.1 Hiding and Showing Groups of Components Based on Binding Properties 2-24
2.12.2 Toggling Between Alternative Sets of Components Based on Binding Properties. 2-25
2.13 Adding the Edit Page and Finishing the Use Case 2-27
2.13.1 Adding Another View Object to the Data Model 2-27
2.13.2 Creating the Edit Page 2-28
2.13.3 Synchronizing the Search and Edit Page 2-29
2.13.4 Controlling Whether Data Appears Initially 2-30
2.13.5 Running the Final Result 2-30
2.14 Considering How Much Code Was Involved 2-31

3 Oracle ADF Service Request Demo Overview

3.1 Introduction to the Oracle ADF Service Request Demo 3-1
3.1.1 Requirements for Oracle ADF Service Request Application 3-2
3.1.2 Overview of the Schema 3-2
3.2 Setting Up the Oracle ADF Service Request Demo 3-4
3.2.1 Downloading and Installing the Oracle ADF Service Request Application 3-4
3.2.2 Installing the Oracle ADF Service Request Schema 3-5
3.2.3 Creating the Oracle JDeveloper Database Connection 3-7
3.2.4 Running the Oracle ADF Service Request Demo in JDeveloper 3-8
3.2.5 Running the Oracle ADF Service Request Demo Unit Tests in JDeveloper 3-10
3.3 Quick Tour of the Oracle ADF Service Request Demo 3-11
3.3.1 Customer Logs In and Reviews Existing Service Requests 3-12
3.3.2 Customer Creates a Service Request 3-14
3.3.3 Manager Logs In and Assigns a Service Request 3-17
3.3.4 Manager Views Reports and Updates Technician Skills 3-19
3.3.5 Technician Logs In and Updates a Service Request 3-22

Part II Building Your Business Services

4 Overview of ADF Business Components

4.1 Prescriptive Approach and Reusable Code for Business Services 4-1
4.2 What are ADF Business Components and What Can They Do? 4-2
4.3 Relating ADF Business Components to Familiar 4GL Tools 4-4
4.3.1 Familiar Concepts for Oracle Forms Developers 4-4
4.3.2 Familiar Concepts for PeopleTools Developers 4-6
4.3.3 Familiar Concepts for SiebelTools Developers 4-7
4.3.4 Familiar Functionality for ADO.NET Developers 4-8
4.4 Overview of ADF Business Components Implementation Architecture 4-8
4.4.1 Based on Standard Java and XML 4-8
4.4.2 Works with Any Application Server or Database 4-9
4.4.3 Implements All of the J2EE Design Patterns You Need 4-9
4.4.4 Components are Organized into Packages 4-9
4.4.5 Architecture of the Base ADF Business Components Layer 4-11
4.4.6 Components Are Metadata-Driven With Optional Custom Java Code 4-11
4.4.6.1 Example of an XML-Only Component 4-11
4.4.6.2 Example of a Component with Custom Java Class 4-12
4.4.7 Recommendations for Configuring ADF Business Components Design Time Preferences 4-13
4.4.7.1 Recommendation for Initially Disabling Custom Java Generation 4-13
4.4.7.2 Recommendation for Disabling Use of Package XML File 4-14
4.4.8 Basic Datatypes 4-14
4.4.9 Generic Versus Strongly-Typed APIs 4-15
4.4.10 Client-Accessible Components Can Have Custom Interfaces 4-16
4.4.10.1 Framework Client Interfaces for Components 4-16
4.4.10.2 Custom Client Interfaces for Components 4-16
4.5 Understanding the Active Data Model 4-17
4.5.1 What is an Active Data Model? 4-17
4.5.2 Examples of the Active Data Model In Action 4-17
4.5.3 Active Data Model Allows You to Eliminate Most Client-Side Code 4-18
4.6 Overview of ADF Business Components Design Time Facilities 4-19
4.6.1 Choosing a Connection, SQL Flavor, and Type Map 4-19
4.6.2 Creating New Components Using Wizards 4-20
4.6.3 Quick-Creating New Components Using the Context Menu 4-20
4.6.4 Editing Components Using the Component Editor 4-21
4.6.5 Visualizing, Creating, and Editing Components Using UML Diagrams 4-21
4.6.6 Testing Application Modules Using the Business Components Browser 4-21
4.6.7 Refactoring Components 4-21

5 Querying Data Using View Objects

5.1 Introduction to View Objects 5-1
5.2 Creating a Simple, Read-Only View Object 5-2
5.2.1 How to Create a Read-Only View Object 5-2
5.2.2 What Happens When You Create a Read-Only View Object 5-5
5.2.3 What You May Need to Know About View Objects 5-5
5.2.3.1 Editing an Existing View Object Definition 5-5

5.2.3.2 Working with Queries That Include SQL Expressions 5-5
5.2.3.3 Controlling the Length, Precision, and Scale of View Object Attributes 5-6
5.3 Using a View Object in an Application Module's Data Model 5-6
5.3.1 How to Create an Application Module 5-7
5.3.1.1 Understanding the Difference Between View Object Components and View Object Instances 5-8
5.3.2 What Happens When You Create an Application Module 5-10
5.3.3 What You May Need to Know About Application Modules... 5-11
5.3.3.1 Editing an Application Module's Runtime Configuration Properties.............. 5-11
5.4 Defining Attribute Control Hints 5-12
5.4.1 How to Add Attribute Control Hints 5-12
5.4.2 What Happens When You Add Attribute Control Hints 5-13
5.4.3 What You May Need to Know About Message Bundles... 5-14
5.5 Testing View Objects Using the Business Components Browser 5-14
5.5.1 How to Test a View Object Using the Business Components Browser 5-14
5.5.2 What Happens When You Use the Business Components Browser........................... 5-15
5.5.3 What You May Need to Know About the Business Components Browser 5-16
5.5.3.1 Customizing Configuration Options for the Current Run 5-16
5.5.3.2 Enabling ADF Business Components Debug Diagnostics 5-17
5.6 Working Programmatically with View Object Query Results 5-18
5.6.1 Common Methods for Working with the View Object’s Default RowSet 5-18
5.6.2 Counting the Number of Rows in a RowSet... 5-19
5.7 How to Create a Command-Line Java Test Client 5-19
5.7.1 What Happens When You Run a Test Client Program... 5-21
5.7.2 What You May Need to Know About Running a Test Client...................................... 5-22
5.8 Filtering Results Using Query-By-Example View Criteria 5-22
5.8.1 How to Use View Criteria to Filter View Object Results 5-22
5.8.2 What Happens When You Use View Criteria to Filter View Object Results.............. 5-24
5.8.3 What You May Need to Know About Query-By-Example Criteria............................. 5-24
5.8.3.1 Use Attribute Names in View Criteria, Column Names in WHERE Clause.... 5-24
5.8.3.2 Testing View Criteria in the Business Component Browser 5-25
5.8.3.3 Altering Compound Search Conditions Using Multiple View Criteria Rows . 5-25
5.8.3.4 Searching for a Row Whose Attribute Value is NULL Value 5-26
5.8.3.5 Searching Case-Insensitively 5-26
5.8.3.6 Clearing View Criteria in Effect 5-26
5.8.3.7 Applying View Criteria Causes Query to be Re-parsed..................................... 5-26
5.9 Using Named Bind Variables.. 5-26
5.9.1 Adding a Named Bind Variable 5-26
5.9.2 What Happens When You Add Named Bind Variables.. 5-27
5.9.3 What You May Need to Know About Named Bind Variables 5-28
5.9.3.1 Errors Related to Bind Variables 5-28
5.9.3.2 Bind Variables Default to NULL If No Default Supplied.................................... 5-29
5.9.3.3 Setting Existing Bind Variable Values at Runtime 5-29
5.9.3.4 Adding a Named Bind Variable at Runtime 5-30
5.9.3.5 Understanding the Default Use of Inline Views for Read-Only Queries.......... 5-32
5.10 Working with Master/Detail Data 5-33
5.10.1 How to Create a Read-Only View Object Joining Tables 5-34
5.10.1.1 Using the Query Builder to Simplify Creating Joins 5-34

5.10.1.2 Testing the Join View ... 5-36
5.10.2 How to Create Master/Detail Hierarchies Using View Links ... 5-36
5.10.3 What Happens When You Create Master/Detail Hierarchies Using View Links.. 5-38
5.10.4 What You May Need to Know About View Links ... 5-38
5.10.4.1 View Link Accessor Attributes Return a RowSet ... 5-39
5.10.4.2 How to Access a Detail Collection Using the View Link Accessor ... 5-39
5.10.4.3 How to Enable Active Master/Detail Coordination in the Data Model ... 5-41
5.11 Generating Custom Java Classes for a View Object ... 5-43
5.11.1 How To Generate Custom Classes ... 5-43
5.11.1.1 Generating Bind Variable Accessors ... 5-44
5.11.1.2 Generating View Row Attribute Accessors ... 5-45
5.11.1.3 Exposing View Row Accessors to Clients ... 5-45
5.11.1.4 Configuring Default Java Generation Preferences ... 5-46
5.11.2 What Happens When You Generate Custom Classes ... 5-46
5.11.2.1 Seeing and Navigating to Custom Java Files ... 5-47
5.11.3 What You May Need to Know About Custom Classes ... 5-47
5.11.3.1 About the Framework Base Classes for a View Object ... 5-47
5.11.3.2 You Can Safely Add Code to the Custom Component File ... 5-48
5.11.3.3 Attribute Indexes and InvokeAccessor Generated Code ... 5-48

6 Creating a Business Domain Layer Using Entity Objects

6.1 Introduction to Entity Objects ... 6-1
6.2 Creating Entity Objects and Associations ... 6-2
6.2.1 How to Create Entity Objects and Associations from Existing Tables ... 6-2
6.2.2 What Happens When You Create Entity Objects and Associations from Existing Tables 6-4
6.2.2.1 What Happens When a Table Has No Primary Key ... 6-5
6.2.3 Creating Entity Objects Using the Create Entity Wizard ... 6-6
6.2.4 Creating an Entity Object for a Synonym or View ... 6-6
6.2.5 Editing an Existing Entity Object or Association ... 6-6
6.2.6 Creating Database Tables from Entity Objects ... 6-6
6.2.6.1 Using Database Key Constraints for an Association ... 6-7
6.2.7 Synchronizing an Entity with Changes to Its Database Table ... 6-7
6.2.8 What You May Need to Know About Creating Entities ... 6-7
6.3 Creating and Configuring Associations ... 6-8
6.3.1 How to Create an Association ... 6-8
6.3.1.1 Changing Entity Association Accessor Names ... 6-10
6.3.1.2 Renaming and Moving Associations to a Different Package ... 6-10
6.3.2 What Happens When You Create an Association ... 6-11
6.3.3 What You May Need to Know About Composition Associations ... 6-11
6.4 Creating an Entity Diagram for Your Business Layer ... 6-12
6.4.1 How to Create an Entity Diagram ... 6-12
6.4.1.1 Publishing the Business Entity Diagram ... 6-13
6.4.2 What Happens When You Create an Entity Diagram ... 6-13
6.4.3 What You May Need to Know About Creating Entities On a Diagram ... 6-14
6.4.3.1 UML Diagram is Actively Synchronized with Business Components ... 6-14
6.4.3.2 UML Diagram Adds Extra Metadata to XML Component Descriptors ... 6-14

6.5 Defining Attribute Control Hints 6-15
6.5.1 How to Add Attribute Control Hints 6-15
6.5.2 What Happens When You Add Attribute Control Hints 6-16
6.5.3 Internationalizing the Date Format 6-16
6.6 Configuring Declarative Runtime Behavior 6-17
6.6.1 How To Configure Declarative Runtime Behavior 6-17
6.6.2 What Happens When You Configure Declarative Runtime Behavior 6-18
6.6.3 About the Declarative Entity Object Features 6-19
6.6.3.1 Legal Database and Java Data types for an Entity Object Attribute 6-19
6.6.3.2 Indicating Datatype Length, Precision, and Scale 6-20
6.6.3.3 Controlling the Updatability of an Attribute 6-20
6.6.3.4 Making an Attribute Mandatory 6-20
6.6.3.5 Defining the Primary Key for the Entity 6-21
6.6.3.6 Defining a Static Default Value 6-21
6.6.3.7 Synchronization with Trigger-Assigned Values 6-21
6.6.3.8 Trigger-Assigned Primary Key Values from a Database Sequence 6-22
6.6.3.9 Lost Update Protection 6-23
6.6.3.10 History Attributes 6-23
6.6.3.11 Setting the Discriminator Attribute for Entity Object Inheritance Hierarchies 6-24
6.6.3.12 Understanding and Configuring Composition Behavior 6-24
6.7 Using Declarative Validation Rules 6-26
6.7.1 How to Add a Validation Rule 6-26
6.7.2 What Happens When You Add a Validation Rule 6-27
6.7.3 What You May Need to Know About Validation Rules 6-28
6.7.3.1 Understanding the Built-in Entity-Level Validators 6-29
6.7.3.2 Understanding the Built-in Attribute-Level Validators 6-29
6.7.3.3 Caveat About the List Validator 6-29
6.8 Working Programmatically with Entity Objects and Associations 6-30
6.8.1 Finding an Entity Object by Primary Key 6-30
6.8.2 Accessing an Associated Entity Using the Accessor Attribute 6-31
6.8.3 Updating or Removing an Existing Entity Row 6-33
6.8.4 Creating a New Entity Row 6-34
6.8.5 Testing Using a Static Main Method 6-36
6.9 Generating Custom Java Classes for an Entity Object 6-38
6.9.1 How To Generate Custom Classes 6-38
6.9.1.1 Choosing to Generate Entity Attribute Accessors 6-39
6.9.2 What Happens When You Generate Custom Classes 6-40
6.9.3 Seeing and Navigating to Custom Java Files 6-40
6.9.4 What You May Need to Know About Custom Java Classes 6-41
6.9.4.1 About the Framework Base Classes for an Entity Object 6-41
6.9.4.2 You Can Safely Add Code to the Custom Component File 6-41
6.9.4.3 Configuring Default Java Generation Preferences 6-41
6.9.4.4 Attribute Indexes and InvokeAccessor Generated Code 6-41
6.9.5 Programmatic Example for Comparison Using Custom Entity Classes 6-44
6.10 Adding Transient and Calculated Attributes to an Entity Object 6-47
6.10.1 How to Add a Transient Attribute 6-47
6.10.2 What Happens When You Add Transient Attribute 6-48

6.10.3 Adding Java Code in the Entity Class to Perform Calculation 6-48

7 Building an Updatable Data Model With Entity-Based View Objects

7.1 Introduction to Entity-Based View Objects 7-1
7.2 Creating an Entity-Based View Object.. 7-2
7.2.1 How to Create an Entity-Based View Object.. 7-2
7.2.1.1 Creating a View Object Having All Attributes of an Entity Object 7-5
7.2.2 What Happens When You Create an Entity-Based View Object................................ 7-6
7.2.3 Editing an Existing Entity-Based View Object Definition 7-6
7.2.4 What You May Need to Know About View Objects 7-6
7.2.4.1 View Object Attributes Inherit Properties from Underlying Entity Object Attributes 7-6
7.3 Including Reference Entities in Join View Objects.. 7-7
7.3.1 How to Include Reference Entities in a View Object.. 7-8
7.3.1.1 Adding Additional Reference Entity Usages to the View Object......................... 7-8
7.3.1.2 Selecting Additional Attributes from Reference Entity Usages 7-9
7.3.1.3 Renaming Attributes from Reference Entity Usages 7-10
7.3.1.4 Removing Unnecessary Key Attributes from Reference Entity Usages........... 7-10
7.3.1.5 Hiding the Primary Key Attributes from Reference Entity Usages.................. 7-11
7.3.2 What Happens When You Reference Entities in a View Object.............................. 7-11
7.3.3 What You May Need to Know About Join View Objects... 7-12
7.3.3.1 Showing View Objects in a Business Components Diagram............................ 7-12
7.3.3.2 Modify Default Join Clause to Be Outer Join When Appropriate..................... 7-13
7.4 Creating an Association-Based View Link.. 7-13
7.4.1 How to Create an Association-Based View Link 7-14
7.4.2 What Happens When You Create an Association-Based View Link....................... 7-15
7.5 Testing Entity-Based View Objects Interactively 7-16
7.5.1 Overview of Business Component Browser Functionality for an Updatable Data Model 7-16
7.5.2 Adding View Object Instances to the Data Model.. 7-16
7.5.3 How to Test Entity-Based View Objects Interactively 7-17
7.5.4 What Happens When You Test Entity-Based View Objects Interactively 7-17
7.5.5 Simulating End-User Interaction with Your Application Module Data Model...... 7-19
7.5.5.1 Testing Master/Detail Coordination.. 7-19
7.5.5.2 Testing UI Control Hints 7-19
7.5.5.3 Testing View Objects That Reference Entity Usages... 7-19
7.5.5.4 Testing Business Domain Layer Validation... 7-20
7.5.5.5 Testing Alternate Language Message Bundles and Control Hints................... 7-20
7.5.5.6 Testing Row Creation and Default Value Generation 7-20
7.5.5.7 Testing New Detail Rows Have Correct Foreign Keys...................................... 7-20
7.6 Adding Calculated and Transient Attributes to an Entity-Based View Object 7-21
7.6.1 How to Add a SQL-Calculated Attribute.. 7-21
7.6.2 What Happens When You Add a SQL-Calculated Attribute 7-22
7.6.3 How to Add a Transient Attribute.. 7-22
7.6.3.1 Adding an Entity-Mapped Transient Attribute to a View Object..................... 7-23
7.6.4 What Happens When You Add a Transient Attribute.. 7-23
7.6.5 Adding Java Code in the View Row Class to Perform Calculation 7-24

7.6.6 What You May Need to Know About Transient Attributes 7-24
7.7 Understanding How View Objects and Entity Objects Cooperate at Runtime 7-24
7.7.1 Each View Row or Entity Row Has a Related Key 7-25
7.7.2 What Role Does the Entity Cache Play in the Transaction 7-26
7.7.3 Metadata Ties Together Cleanly Separated Roles of Data Source and Data Sink... 7-27
7.7.4 What Happens When a View Object Executes Its Query 7-28
7.7.5 What Happens When You Modify a View Row Attribute 7-29
7.7.6 What Happens When You Change a Foreign Key Attribute 7-31
7.7.7 What Happens When You Re-query Data 7-31
7.7.7.1 Unmodified Attributes in Entity Cache are Refreshed During Re-query 7-32
7.7.7.2 Modified Attributes in Entity Cache are Left Intact During Re-query 7-32
7.7.7.3 Overlapping Subsets of Attributes are Merged During Re-query 7-33
7.7.8 What Happens When You Commit the Transaction 7-33
7.7.9 Interactively Testing Multiuser Scenarios 7-34
7.8 Working Programmatically with Entity-Based View Objects 7-34
7.8.1 Example of Iterating Master/Detail/Detail Hierarchy 7-34
7.8.2 Example of Finding a Row and Updating a Foreign Key Value 7-36
7.8.3 Example of Creating a New Service Request 7-38
7.8.4 Example of Retrieving the Row Key Identifying a Row 7-39
7.9 Summary of Difference Between Entity-Based View Objects and Read-Only View Objects .. 7-40
7.9.1 Runtime Features Unique to Entity-Based View Objects 7-40
7.9.2 View Objects with No Entity Usage Are Read-Only 7-40
7.9.3 What You May Need to Know About Enabling View Object Key Management for Read-Only View Objects 7-41

8 Implementing Business Services with Application Modules

8.1 Introduction to Application Modules 8-1
8.2 Creating an Application Module 8-2
8.2.1 Creating an Application Module 8-3
8.2.2 What Happens When You Create an Application Module 8-3
8.2.3 Editing an Existing Application Module 8-3
8.2.4 Configuring Your Application Module Database Connection 8-3
8.2.4.1 Using a JDBC URL Connection Type 8-4
8.2.4.2 Using a JDBC Datasource Connection Type 8-4
8.2.5 Managing Your Application Module's Runtime Configurations 8-6
8.2.6 What You Might Need to Know About Application Module Connections 8-6
8.2.6.1 The Business Components Browser Requires a JDBC URL Connection 8-6
8.2.6.2 Testing the SRService Application Module in the Business Components Browser ... 8-6
8.3 Adding a Custom Service Method 8-6
8.3.1 How to Generate a Custom Class for an Application Module 8-7
8.3.2 What Happens When You Generate a Custom Class for an Application Module 8-7
8.3.3 What You May Need to Know About Default Code Generation 8-8
8.3.4 Debugging the Application Module Using the Business Components Tester 8-9
8.3.5 How to Add a Custom Service Method to an Application Module 8-9
8.4 Publishing Custom Service Methods to Clients 8-10

8.4.1 How to Publish Custom Service Methods to Clients ... 8-11
8.4.2 What Happens When You Publish Custom Service Methods to Clients ... 8-11
8.4.3 How to Generate Client Interfaces for View Objects and View Rows ... 8-12
8.4.4 What You May Need to Know About Method Signatures on the Client Interface 8-13
8.4.5 What You May Need to Know About Passing Information from the Data Model 8-13
8.5 Working Programmatically with an Application Module's Client Interface ... 8-14
8.5.1 How to Work Programmatically with an Application Module's Client Interface .. 8-14
8.5.2 What Happens When You Work with an Application Module's Client Interface.. 8-16
8.5.3 How to Access an Application Module Client Interface ... 8-16
8.5.3.1 How to Access an Application Module Client Interface in a JSF Web Application ... 8-17
8.5.3.2 How to Access an Application Module Client Interface in a JSP/Struts Web Application 8-19
8.5.3.3 How to Access an Application Module Client Interface in an ADF Swing Application 8-19
8.6 Overriding Built-in Framework Methods ... 8-19
8.6.1 How to Override a Built-in Framework Method ... 8-20
8.6.2 What Happens When You Override a Built-in Framework Method ... 8-20
8.6.3 How to Override prepareSession() to Set Up an Application Module for a New User Session 8-21
8.7 Creating an Application Module Diagram for Your Business Service ... 8-23
8.7.1 How to Create an Application Module Diagram... 8-23
8.7.2 What Happens When You Create an Application Module Diagram ... 8-23
8.7.3 What You May Need to Know About Application Module Diagrams ... 8-24
8.7.3.1 Using the Diagram for Editing the Application Module... 8-24
8.7.3.2 Controlling Display Options... 8-24
8.7.3.3 Filtering Method Names... 8-25
8.7.3.4 Show Related Objects and Implementation Files ... 8-25
8.7.3.5 Publishing the Application Module Diagram... 8-26
8.7.3.6 Testing the Application Module From the Diagram... 8-26
8.8 Supporting Multipage Units of Work... 8-26
8.8.1 Overview of Application Module Pooling and State Management... 8-26
8.8.2 Experimenting with State Management in the Business Components Browser 8-27
8.9 Deciding on the Granularity of Application Modules ... 8-28
8.9.1 Use Cases Assist in Planning Your Application Modules... 8-28
8.9.2 Application Modules Are Designed to Support Assembly... 8-29
8.9.3 Root Application Modules Versus Nested Application Module Usages ... 8-30

9 Implementing Programmatic Business Rules in Entity Objects

9.1 Introduction to Programmatic Business Rules ... 9-1
9.2 Understanding the Validation Cycle ... 9-2
9.2.1 Types of Entity Object Validation Rules ... 9-2
9.2.1.1 Attribute-Level Validation Rules ... 9-3
9.2.1.2 Entity-Level Validation Rules... 9-3
9.2.2 Understanding Commit Processing and Validation ... 9-3
9.2.3 Avoiding Infinite Validation Cycles ... 9-4
9.2.4 What Happens When Validations Fail ... 9-4

9.2.5 Understanding Entity Objects Row States 9-4
9.3 Using Method Validators..... 9-5
9.3.1 How to Create an Attribute-Level Method Validation 9-5
9.3.2 What Happens When You Create an Attribute-Level Method Validator..... 9-6
9.3.3 How to Create an Entity-Level Method Validator..... 9-7
9.3.4 What Happens When You Create an Entity-Level Method Validator 9-8
9.3.5 What You Might Need To Know About Translating Validation Rule Error Messages.... 9-8
9.3.6 What You May Need to Know About Referencing the Invalid Value in an Attribute-Level Validation Error Message 9-8
9.4 Assigning Programmatically-Derived Attribute Values..... 9-8
9.4.1 Defaulting Values for New Rows at Create Time..... 9-8
9.4.1.1 Choosing Between create() and initDefaults() Methods..... 9-9
9.4.1.2 Eagerly Defaulting an Attribute Value from a Database Sequence 9-9
9.4.2 Assigning Derived Values Before Saving 9-9
9.4.3 Assigning Derived Values When an Attribute Value is Set 9-10
9.5 Undoing Pending Changes to an Entity Using the Refresh Method 9-11
9.5.1 Controlling What Happens to New Rows During a Refresh..... 9-11
9.5.2 Cascading Refresh to Composed Children Entity Rows 9-11
9.6 Using View Objects for Validation..... 9-11
9.6.1 Creating View Objects at Runtime for Validation 9-11
9.6.2 Implementing an Efficient Existence Check 9-13
9.6.3 Validating Conditions Related to All Entities of a Given Type 9-14
9.7 How to Access Related Entity Rows Using Association Accessors..... 9-15
9.8 How to Reference Information About the Authenticated User 9-16
9.8.1 Referencing Role Information About the Authenticated User..... 9-16
9.8.2 Referencing the Name of the Authenticated User 9-17
9.9 How to Access Original Attribute Values 9-18
9.10 How to Store Information About the Current User Session..... 9-18
9.11 How to Access the Current Date and Time..... 9-19
9.12 How to Send Notifications Upon a Successful Commit..... 9-20
9.13 How to Conditionally Prevent an Entity Row from Being Removed 9-20
9.14 How to Implement Conditional Updatability for Attributes 9-20
9.15 Additional Resources 9-21

10 Overview of Application Module Data Binding

10.1 Overview of Data Controls and Declarative Bindings 10-1
10.1.1 Data Controls Abstract the Implementation Technology of a Business Service 10-1
10.1.2 Bindings Connect UI Controls to Data Collections and Operations 10-2
10.2 Understanding the Application Module Data Control 10-3
10.3 How an Application Module Appears in the Data Control Palette 10-3
10.3.1 Overview of the SRService Application Module 10-4
10.3.2 How to Change the Data Control Name Before You Begin Building Pages..... 10-4
10.3.3 How the Data Model and Service Methods Appear in the Data Control Palette ... 10-5
10.3.4 How to Change View Instance Names Before You Begin Building Pages..... 10-6
10.3.5 How Transaction Control Operations Appear in the Data Control Palette..... 10-7
10.3.6 How View Objects Appear in the Data Control Palette..... 10-7

10.3.6.1 Built-in Operations for View Object Data Collections 10-8
10.3.7 How Nested Application Modules Appear in the Data Control Palette................. 10-9
10.4 How to Add a Create Button on a Page.. 10-10
10.4.1 What Happens When You Drop a Create Button on a Web Page.......................... 10-10
10.4.2 What Happens When You Drop a Create Operation Onto a Swing Panel 10-11
10.4.3 When to Use CreateInsert Instead of Create.. 10-11
10.4.4 What You May Need to Know About Create and CreateInsert 10-11
10.5 Application Module Databinding Tips and Techniques... 10-12
10.5.1 How to Create a Record Status Display 10-12
10.5.2 How to Work with Named View Object Bind Variables 10-13
10.5.3 How to Use Find Mode to Implement Query-by-Example 10-15
10.5.4 How to Customize the ADF Page Lifecycle to Work Programmatically with Bindings .. 10-16
10.5.4.1 Globally Customizing the ADF Page Lifecycle 10-16
10.5.4.2 Customizing the Page Lifecycle for a Single Page 10-17
10.5.4.3 Using Custom ADF Page Lifecycle to Invoke an onPageLoad Backing Bean Method 10-17
10.5.5 How to Use Refresh Correctly for InvokeAction and Iterator Bindings 10-19
10.5.5.1 Correctly Configuring the Refresh Property of Iterator Bindings.................... 10-19
10.5.5.2 Refreshing an Iterator Binding Does Not Forcibly Re-Execute Query 10-20
10.5.5.3 Correctly Configuring Refresh Property of InvokeAction Executables 10-20
10.5.6 Understanding the Difference Between setCurrentRowWithKey and setCurrentRowWithKeyValue 10-21
10.5.7 Understanding Bundled Exception Mode 10-23
10.6 Overview of How SRDemo Pages Use the SRService 10-23
10.6.1 The SRList Page.. 10-23
10.6.1.1 Overview of Data Binding in the SRList Page 10-23
10.6.1.2 Business Service Notes for the SRList Page 10-24
10.6.2 The SRMain Page 10-24
10.6.2.1 Overview of Data Binding in the SRMain Page... 10-24
10.6.2.2 Business Service Notes for the SRMain Page 10-25
10.6.3 The SREdit Page 10-26
10.6.3.1 Overview of Data Binding in the SREdit Page... 10-26
10.6.3.2 Business Service Notes for the SREdit Page 10-27
10.6.4 The SRSearch Page.. 10-28
10.6.4.1 Overview of Data Binding in the SRSearch Page 10-28
10.6.4.2 Business Service Notes for the SRSearch Page... 10-29
10.6.5 The SRStaffSearch Page 10-29
10.6.5.1 Overview of Data Binding in the SRStaffSearch Page 10-29
10.6.5.2 Business Service Notes for the SRStaffSearch Page... 10-30
10.6.6 The SRManage Page 10-31
10.6.6.1 Overview of Data Binding in the SRManage Page.. 10-31
10.6.6.2 Business Service Notes for the SRManage Page 10-32
10.6.7 The SRSkills Page.. 10-32
10.6.7.1 Overview of Data Binding in the SRSkills Page... 10-32
10.6.7.2 Business Service Notes for the SRSkills Page 10-33
10.6.8 The SRCreate Page.. 10-35

10.6.8.1 Overview of Data Binding in the SRCreate Page 10-35
10.6.8.2 Business Service Notes for the SRCreate Page 10-36
10.6.9 The SRConfirmCreate Page 10-36
10.6.9.1 Overview of Data Binding in the SRConfirmCreate Page 10-36
10.6.9.2 Business Service Notes for the SRCreate Page 10-37

Part III Building Your Web Interface

11 Getting Started with ADF Faces

11.1 Introduction to ADF Faces 11-1
11.2 Setting Up a Workspace and Project 11-3
11.2.1 What Happens When You Use an Application Template to Create a Workspace . 11-4
11.2.1.1 Starter web.xml File 11-5
11.2.1.2 Starter faces-config.xml File 11-7
11.2.2 What You May Need to Know About the ViewController Project 11-8
11.2.3 What You May Need to Know About Multiple JSF Configuration Files 11-9
11.3 Creating a Web Page 11-10
11.3.1 How to Add a JSF Page 11-10
11.3.2 What Happens When You Create a JSF Page 11-12
11.3.3 What You May Need to Know About Using the JSF Navigation Diagram 11-13
11.3.4 What You May Need to Know About ADF Faces Dependencies and Libraries... 11-14
11.4 Laying Out a Web Page 11-14
11.4.1 How to Add UI Components to a JSF Page 11-15
11.4.2 What Happens When You First Insert an ADF Faces Component 11-16
11.4.2.1 More About the web.xml File 11-18
11.4.2.2 More About the faces-config.xml File 11-19
11.4.2.3 Starter adf-faces-config.xml File 11-20
11.4.3 What You May Need to Know About Creating JSF Pages 11-21
11.4.3.1 Editing in the Structure Window 11-22
11.4.3.2 Displaying Errors 11-23
11.4.4 Using the PanelPage Component 11-24
11.4.4.1 PanelPage Facets 11-26
11.4.4.2 Page Body Contents 11-29
11.5 Creating and Using a Backing Bean for a Web Page 11-30
11.5.1 How to Create and Configure a Backing Bean 11-31
11.5.2 What Happens When You Create and Configure a Backing Bean 11-32
11.5.3 How to Use a Backing Bean in a JSF Page 11-33
11.5.4 How to Use the Automatic Component Binding Feature 11-33
11.5.5 What Happens When You Use Automatic Component Binding in JDeveloper... 11-34
11.5.6 What You May Need to Know About Backing Beans and Managed Beans 11-35
11.5.7 Using ADF Data Controls and Backing Beans 11-37
11.6 Best Practices for ADF Faces 11-38

12 Displaying Data on a Page

12.1 Introduction to Displaying Data on a Page 12-1
12.2 Using the Data Control Palette 12-2

12.2.1 How to Understand the Items on the Data Control Palette 12-3
12.2.2 How to Use the Data Control Palette.. 12-6
12.2.3 What Happens When You Use the Data Control Palette 12-7
12.2.4 What Happens at Runtime 12-8
12.3 Working with the DataBindings.cpx File 12-9
12.3.1 How to Create a DataBindings.cpx File 12-9
12.3.2 What Happens When You Create a DataBindings.cpx File 12-10
12.4 Configuring the ADF Binding Filter 12-10
12.4.1 How to Configure the ADF Binding Filter.. 12-10
12.4.2 What Happens When You Configure an ADF Binding Filter................................ 12-11
12.4.3 What Happens at Runtime 12-12
12.5 Working with Page Definition Files 12-12
12.5.1 How to Create a Page Definition File 12-12
12.5.2 What Happens When You Create a Page Definition File 12-13
12.5.2.1 Binding Objects Defined in the parameters Element 12-14
12.5.2.2 Binding Objects Defined in the executables Element....................................... 12-15
12.5.2.3 Binding Objects Defined in the bindings Element... 12-18
12.5.3 What Happens at Runtime 12-19
12.5.4 What You May Need to Know About Binding Container Scope 12-19
12.6 Creating ADF Data Binding EL Expressions 12-20
12.6.1 How to Create an ADF Data Binding EL Expression... 12-20
12.6.2 How to Use the Expression Builder 12-21
12.6.3 What Happens When You Create ADF Data Binding Expressions 12-24
12.6.3.1 EL Expressions That Reference Attribute Binding Objects 12-24
12.6.3.2 EL Expressions That Reference Table Binding Objects.................................... 12-24
12.6.3.3 EL Expressions That Reference Action Binding Objects.................................. 12-25
12.6.4 What You May Need to Know About ADF Binding Properties............................ 12-27

13 Creating a Basic Page

13.1 Introduction to Creating a Basic Page.. 13-1
13.2 Using Attributes to Create Text Fields.. 13-2
13.2.1 How to Use the Data Control Palette to Create a Text Field 13-2
13.2.2 What Happens When You Use the Data Control Palette to Create a Text Field..... 13-3
13.2.2.1 Creating and Using Iterator Bindings 13-3
13.2.2.2 Creating and Using Value Bindings 13-4
13.2.2.3 Using EL Expressions to Bind UI Components 13-5
13.2.3 What Happens at Runtime: The JSF and ADF Lifecycles 13-6
13.3 Creating a Basic Form... 13-9
13.3.1 How to Use the Data Control Palette to Create a Form 13-9
13.3.2 What Happens When You Use the Data Control Palette to Create a Form........... 13-11
13.3.2.1 Using Facets.. 13-11
13.4 Incorporating Range Navigation into Forms.. 13-12
13.4.1 How to Insert Navigation Controls into a Form 13-13
13.4.2 What Happens When Command Buttons Are Created Using the Data Control Palette.. 13-14
13.4.2.1 Using Action Bindings for Built-in Navigation Operations............................ 13-14
13.4.2.2 Iterator RangeSize Attribute 13-14

13.4.2.3 Using EL Expressions to Bind to Navigation Operations 13-15
13.4.3 What Happens at Runtime: About Action Events and Action Listeners 13-16
13.4.4 What You May Need to Know About the Browser Back Button............................ 13-17
13.5 Creating a Form to Edit an Existing Record 13-18
13.5.1 How to Use the Data Control Palette to Create Edit Forms 13-18
13.5.2 What Happens When You Use Built-in Operations to Change Data 13-19
13.6 Creating an Input Form 13-21
13.6.1 How to Create an Input Form.. 13-21
13.6.2 What Happens When You Create an Input Form.. 13-22
13.6.3 What You May Need to Know About Displaying Sequence Numbers................. 13-23
13.6.4 What You May Need to Know About Create Forms and the RefreshCondition.. 13-24
13.7 Modifying the UI Components and Bindings on a Form 13-25
13.7.1 How to Modify the UI Components and Bindings.. 13-25
13.7.1.1 Changing the Value Binding for a UI Component 13-26
13.7.1.2 Changing the Action Binding for a UI Component.. 13-26
13.7.2 What Happens When You Modify Attributes and Bindings 13-27

14 Adding Tables

14.1 Introduction to Adding Tables 14-1
14.2 Creating a Basic Table 14-2
14.2.1 How to Create a Basic Table... 14-2
14.2.2 What Happens When You Use the Data Control Palette to Create a Table 14-4
14.2.2.1 Iterator and Value Bindings for Tables 14-5
14.2.2.2 Code on the JSF Page for an ADF Faces Table 14-5
14.3 Incorporating Range Navigation into Tables.. 14-7
14.3.1 How to Use Navigation Controls in a Table.. 14-8
14.3.2 What Happens When You Use Navigation Controls in a Table.............................. 14-8
14.3.3 What Happens at Runtime 14-9
14.3.4 What You May Need to Know About the Browser Back Button............................. 14-9
14.4 Modifying the Attributes Displayed in the Table 14-9
14.4.1 How to Modify the Displayed Attributes 14-10
14.4.2 How to Change the Binding for a Table.. 14-11
14.4.3 What Happens When You Modify Bindings or Displayed Attributes 14-12
14.5 Adding Hidden Capabilities to a Table 14-12
14.5.1 How to Use the DetailStamp Facet 14-12
14.5.2 What Happens When You Use the DetailStamp Facet 14-13
14.5.3 What Happens at Runtime 14-14
14.6 Enabling Row Selection in a Table 14-14
14.6.1 How to Use the TableSelectOne Component in the Selection Facet 14-16
14.6.2 What Happens When You Use the TableSelectOne Component 14-17
14.6.3 What Happens at Runtime 14-17
14.6.4 What You May Need to Know About Using Links Instead of the Selection Facet........... 14-17
14.6.5 How to Use the TableSelectMany Component in the Selection Facet 14-18
14.6.6 What Happens When You Use the TableSelectMany Component 14-19
14.6.7 What Happens at Runtime 14-21
14.7 Setting the Current Object Using a Command Component.. 14-22

14.7.1 How to Manually Set the Current Row 14-23
14.7.2 What Happens When You Set the Current Row 14-23
14.7.3 What Happens At Runtime 14-24

15 Displaying Master-Detail Data

15.1 Introduction to Displaying Master-Detail Data 15-1
15.2 Identifying Master-Detail Objects on the Data Control Palette 15-2
15.3 Using Tables and Forms to Display Master-Detail Objects 15-4
15.3.1 How to Display Master-Detail Objects in Tables and Forms 15-5
15.3.2 What Happens When You Create Master-Detail Tables and Forms 15-6
15.3.2.1 Code Generated in the JSF Page 15-6
15.3.2.2 Binding Objects Defined in the Page Definition File 15-7
15.3.3 What Happens at Runtime 15-8
15.3.4 What You May Need to Know About Master-Detail on Separate Pages 15-9
15.4 Using Trees to Display Master-Detail Objects 15-9
15.4.1 How to Display Master-Detail Objects in Trees 15-10
15.4.2 What Happens When You Create ADF Databound Trees 15-13
15.4.2.1 Code Generated in the JSF Page 15-13
15.4.2.2 Binding Objects Defined in the Page Definition File 15-14
15.4.3 What Happens at Runtime 15-15
15.4.4 What You May Need to Know About Adding Command Links to Tree Nodes.. 15-15
15.5 Using Tree Tables to Display Master-Detail Objects 15-17
15.5.1 How to Display Master-Detail Objects in Tree Tables 15-18
15.5.2 What Happens When You Create a Databound Tree Table 15-18
15.5.2.1 Code Generated in the JSF Page 15-19
15.5.2.2 Binding Objects Defined in the Page Definition File 15-19
15.5.3 What Happens at Runtime 15-19
15.6 Using an Inline Table to Display Detail Data in a Master Table 15-20
15.6.1 How to Display Detail Data Using an Inline Table 15-21
15.6.2 What Happens When You Create an Inline Detail Table 15-22
15.6.2.1 Code Generated in the JSF Page 15-23
15.6.2.2 Binding Objects Defined in the Page Definition File 15-23
15.6.3 What Happens at Runtime 15-24

16 Adding Page Navigation

16.1 Introduction to Page Navigation 16-1
16.2 Creating Navigation Rules 16-2
16.2.1 How to Create Page Navigation Rules 16-2
16.2.1.1 About Navigation Rule Elements 16-2
16.2.1.2 Using the Navigation Modeler to Define Navigation Rules 16-3
16.2.1.3 Using the JSF Configuration Editor 16-5
16.2.2 What Happens When You Create a Navigation Rule 16-8
16.2.3 What Happens at Runtime 16-10
16.2.4 What You May Need to Know About Navigation Rules and Cases 16-11
16.2.4.1 Defining Rules in Multiple Configuration Files 16-11
16.2.4.2 Overlapping Rules 16-11

16.2.4.3 Conflicting Navigation Rules ... 16-12
16.2.4.4 Splitting Navigation Cases Over Multiple Rules ... 16-12
16.2.5 What You May Need to Know About the Navigation Modeler ... 16-13
16.3 Using Static Navigation ... 16-14
16.3.1 How to Create Static Navigation ... 16-14
16.3.2 What Happens When You Create Static Navigation ... 16-15
16.4 Using Dynamic Navigation ... 16-16
16.4.1 How to Create Dynamic Navigation ... 16-17
16.4.2 What Happens When You Create Dynamic Navigation ... 16-18
16.4.3 What Happens at Runtime ... 16-19
16.4.4 What You May Need to Know About Using Default Cases ... 16-20
16.4.5 What You May Need to Know About Action Listener Methods ... 16-20
16.4.6 What You May Need to Know About Data Control Method Outcome Returns .. 16-21

17 Creating More Complex Pages

17.1 Introduction to More Complex Pages ... 17-1
17.2 Using a Managed Bean to Store Information ... 17-2
17.2.1 How to Use a Managed Bean to Store Information ... 17-2
17.2.2 What Happens When You Create a Managed Bean ... 17-3
17.3 Creating Command Components to Execute Methods ... 17-4
17.3.1 How to Create a Command Component Bound to a Service Method ... 17-5
17.3.2 What Happens When You Create Command Components Using a Method ... 17-5
17.3.2.1 Using Parameters in a Method ... 17-5
17.3.2.2 Using EL Expressions to Bind to Methods ... 17-6
17.3.3 What Happens at Runtime ... 17-7
17.4 Setting Parameter Values Using a Command Component ... 17-7
17.4.1 How to Set Parameters Using Command Components ... 17-7
17.4.2 What Happens When You Set Parameters ... 17-8
17.4.3 What Happens at Runtime ... 17-8
17.5 Overriding Declarative Methods ... 17-8
17.5.1 How to Override a Declarative Method ... 17-9
17.5.2 What Happens When You Override a Declarative Method ... 17-12

18 Creating a Search Form

18.1 Introduction to Creating Search Forms ... 18-1
18.2 Creating a EnterQuery/ExecuteQuery Search Form ... 18-3
18.2.1 How to Create an EnterQuery/ExecuteQuery Search Page ... 18-4
18.2.2 What Happens When You Create a Search Form ... 18-5
18.3 Creating a Web-type Search Form ... 18-6
18.3.1 How to Create a Search Form and Separate Results Page ... 18-6
18.3.2 What Happens When You Create A Web-type Search Form ... 18-7
18.3.3 What You May Need to Know ... 18-8
18.3.4 About Creating Search and Results on the Same Page ... 18-8
18.3.5 How To Create Search and Results on the Same Page ... 18-8
18.3.6 What Happens When Search and Results are on the Same Page ... 18-9
18.4 Creating Search Page Using Named Bind Variables ... 18-10
18.4.1 How to Create a Parameterized Search Form ... 18-11

18.4.2 What Happens When You Use Parameter Methods 18-11
18.4.3 What Happens at Runtime 18-13
18.5 Conditionally Displaying the Results Table on a Search Page 18-14
18.5.1 How to Add Conditional Display Capabilities 18-15
18.5.2 What Happens When you Conditionally Display the Results Table 18-16

19 Using Complex UI Components

19.1 Introduction to Complex UI Components 19-1
19.2 Using Dynamic Menus for Navigation 19-2
19.2.1 How to Create Dynamic Navigation Menus 19-3
19.2.1.1 Creating a Menu Model 19-3
19.2.1.2 Creating the JSF Page for Each Menu Item 19-13
19.2.1.3 Creating the JSF Navigation Rules 19-16
19.2.2 What Happens at Runtime 19-17
19.2.3 What You May Need to Know About Menus 19-18
19.3 Using Popup Dialogs 19-19
19.3.1 How to Create Popup Dialogs 19-21
19.3.1.1 Defining a JSF Navigation Rule for Launching a Dialog 19-22
19.3.1.2 Creating the JSF Page That Launches a Dialog 19-23
19.3.1.3 Creating the Dialog Page and Returning a Dialog Value 19-24
19.3.1.4 Handling the Return Value 19-27
19.3.1.5 Passing a Value into a Dialog 19-28
19.3.2 How the SRDemo Popup Dialogs Are Created 19-29
19.3.3 What You May Need to Know About ADF Faces Dialogs 19-33
19.3.4 Other Information 19-33
19.4 Enabling Partial Page Rendering 19-33
19.4.1 How to Enable PPR 19-35
19.4.2 What Happens at Runtime 19-36
19.4.3 What You May Need to Know About PPR and Screen Readers 19-36
19.5 Creating a Multipage Process 19-37
19.5.1 How to Create a Process Train 19-38
19.5.1.1 Creating a Process Train Model 19-39
19.5.1.2 Creating the JSF Page for Each Train Node 19-43
19.5.1.3 Creating the JSF Navigation Rules 19-45
19.5.2 What Happens at Runtime 19-46
19.5.3 What You May Need to Know About Process Trains and Menus 19-46
19.6 Providing File Upload Capability 19-47
19.6.1 How to Support File Uploading on a Page 19-49
19.6.2 What Happens at Runtime 19-52
19.6.3 What You May Need to Know About ADF Faces File Upload 19-53
19.6.4 Configuring File Uploading Initialization Parameters 19-53
19.6.5 Configuring a Custom Uploaded File Processor 19-54
19.7 Creating Selection Lists 19-55
19.7.1 How to Create a List with a Fixed List of Values 19-55
19.7.2 What Happens When You Create a List Bound to a Fixed List of Values 19-58
19.7.3 How to Create a List with a Dynamic List of Values 19-59
19.7.4 What Happens When You Create a List Bound to a Dynamic List of Values 19-61

19.7.5 How to Create a List with Navigation List Binding 19-63
19.7.6 What Happens When You Create a List With Navigation List Binding 19-64
19.8 Creating a Shuttle 19-65
19.8.1 How to Create a Shuttle 19-67
19.8.2 What Happens at Runtime 19-71

20 Using Validation and Conversion

20.1 Introduction to Validation and Conversion 20-1
20.2 Validation, Conversion, and the Application Lifecycle 20-2
20.3 Adding Validation 20-3
20.3.1 How to Add Validation 20-4
20.3.1.1 Adding ADF Faces Validation 20-4
20.3.1.2 Adding ADF Model Validation 20-8
20.3.2 What Happens When You Create Input Fields Using the Data Control Palette 20-9
20.3.3 What Happens at Runtime 20-10
20.3.4 What You May Need to Know 20-12
20.4 Creating Custom JSF Validation 20-12
20.4.1 How to Create a Backing Bean Validation Method 20-12
20.4.2 What Happens When You Create a Backing Bean Validation Method 20-13
20.4.3 How to Create a Custom JSF Validator 20-13
20.4.4 What Happens When You Use a Custom JSF Validator 20-16
20.5 Adding Conversion 20-16
20.5.1 How to Use Converters 20-17
20.5.2 What Happens When You Create Input Fields Using the Data Control Palette 20-18
20.5.3 What Happens at Runtime 20-19
20.6 Creating Custom JSF Converters 20-19
20.6.1 How to Create a Custom JSF Converter 20-19
20.6.2 What Happens When You Use a Custom Converter 20-21
20.7 Displaying Error Messages 20-21
20.7.1 How to Display Server-Side Error Messages on a Page 20-22
20.7.2 What Happens When You Choose to Display Error Messages 20-23
20.8 Handling and Displaying Exceptions in an ADF Application 20-23
20.8.1 How to Change Exception Handling 20-24
20.8.2 What Happens When You Change the Default Error Handling 20-26

21 Adding ADF Bindings to Existing Pages

21.1 Introduction to Adding ADF Bindings to Existing Pages 21-1
21.2 Designing Pages for ADF Bindings 21-2
21.2.1 Creating the Page 21-2
21.2.2 Adding Components to the Page 21-3
21.2.3 Other Design Considerations 21-4
21.2.3.1 Creating Text Fields in Forms 21-4
21.2.3.2 Creating Tables 21-4
21.2.3.3 Creating Buttons and Links 21-4
21.2.3.4 Creating Lists 21-5
21.2.3.5 Creating Trees or Tree Tables 21-5
21.3 Using the Data Control Palette to Bind Existing Components 21-5

21.3.1 How to Add ADF Bindings Using the Data Control Palette 21-5
21.3.2 What Happens When You Use the Data Control Palette to Add ADF Bindings 21-7
21.4 Adding ADF Bindings to Text Fields 21-7
21.4.1 How to Add ADF Bindings to Text Fields 21-7
21.4.2 What Happens When You Add ADF Bindings to a Text Field 21-8
21.5 Adding ADF Bindings to Tables 21-8
21.5.1 How to Add ADF Bindings to Tables 21-8
21.5.2 What Happens When You Add ADF Bindings to a Table 21-10
21.6 Adding ADF Bindings to Actions 21-11
21.6.1 How to Add ADF Bindings to Actions 21-11
21.6.2 What Happens When You Add ADF Bindings to an Action 21-12
21.7 Adding ADF Bindings to Selection Lists 21-12
21.7.1 How to Add ADF Bindings to Selection Lists 21-13
21.7.2 What Happens When You Add ADF Bindings to a Selection List 21-13
21.8 Adding ADF Bindings to Trees and Tree Tables 21-14
21.8.1 How to Add ADF Bindings to Trees and Tree Tables 21-14
21.8.2 What Happens When You Add ADF Bindings to a Tree or Tree Table 21-15

22 Changing the Appearance of Your Application

22.1 Introduction to Changing ADF Faces Components 22-1
22.2 Changing the Style Properties of a Component 22-2
22.2.1 How to Set a Component's Style Attributes 22-2
22.2.2 What Happens When You Format Text 22-3
22.3 Using Skins to Change the Look and Feel 22-3
22.3.1 How to Use Skins 22-5
22.3.1.1 Creating a Custom Skin 22-6
22.3.1.2 Configuring an Application to Use a Skin 22-9
22.4 Internationalizing Your Application 22-10
22.4.1 How to Internationalize an Application 22-14
22.4.2 How to Configure Optional Localization Properties for ADF Faces 22-18

23 Optimizing Application Performance with Caching

23.1 About Caching 23-1
23.2 Using ADF Faces Cache to Cache Content 23-2
23.2.1 How to Add Support for ADF Faces Cache 23-6
23.2.2 What Happens When You Cache Fragments 23-6
23.2.2.1 Logging 23-6
23.2.2.2 AFC Statistics Servlet 23-7
23.2.2.3 Visual Diagnostics 23-8
23.2.3 What You May Need to Know 23-8

24 Testing and Debugging Web Applications

24.1 Getting Started with Oracle ADF Model Debugging 24-1
24.2 Correcting Simple Oracle ADF Compilation Errors 24-2
24.3 Correcting Simple Oracle ADF Runtime Errors 24-4
24.4 Understanding a Typical Oracle ADF Model Debugging Session 24-6

24.4.1 Turning on Diagnostic Logging 24-7
24.4.2 Creating an Oracle ADF Debugging Configuration 24-7
24.4.3 Understanding the Different Kinds of Breakpoints 24-8
24.4.4 Editing Breakpoints to For Improved Control 24-9
24.4.5 Filtering Your View of Class Members 24-10
24.4.6 Communicating Stack Trace Information to Someone Else 24-10
24.5 Debugging the Oracle ADF Model Layer 24-10
24.5.1 Correcting Failures to Display Pages 24-12
24.5.1.1 Fixing Binding Context Creation Errors 24-12
24.5.1.2 Fixing Binding Container Creation Errors 24-13
24.5.2 Correcting Failures to Display Data 24-16
24.5.2.1 Fixing Executable Errors 24-16
24.5.2.2 Fixing Render Value Errors Before Submit 24-18
24.5.3 Correcting Failures to Invoke Actions and Methods 24-20
24.6 Tracing EL Expressions 24-23

Part IV Advanced Topics

25 Advanced Business Components Techniques

25.1 Globally Extending ADF Business Components Functionality 25-1
25.1.1 What Are ADF Business Components Framework Extension Classes? 25-2
25.1.2 How To Create a Framework Extension Class 25-2
25.1.3 What Happens When You Create a Framework Extension Class 25-3
25.1.4 How to Base an ADF Component on a Framework Extension Class 25-3
25.1.5 What Happens When You Base a Component on a Framework Extension Class.. 25-4
25.1.5.1 Basing an XML-Only Component on a Framework Extension Class 25-4
25.1.5.2 Basing a Component with a Custom Java Class on a Framework Extension Class ... 25-5
25.1.6 What You May Need to Know 25-6
25.1.6.1 Don't Update the Extends Clause in Custom Component Java Files By Hand 25-6
25.1.6.2 You Can Have Multiple Levels of Framework Extension Classes 25-7
25.1.6.3 Setting up Project-Level Preferences for Framework Extension Classes 25-7
25.1.6.4 Setting Up Framework Extension Class Preferences at the IDE Level 25-8
25.2 Creating a Layer of Framework Extensions 25-8
25.2.1 How to Create Your Layer of Framework Extension Layer Classes 25-8
25.2.2 How to Package Your Framework Extension Layer in a JAR File 25-9
25.2.3 How to Create a Library Definition for Your Framework Extension JAR File 25-9
25.3 Customizing Framework Behavior with Extension Classes 25-10
25.3.1 How to Access Runtime Metadata For View Objects and Entity Objects 25-10
25.3.2 Implementing Generic Functionality Using Runtime Metadata 25-11
25.3.3 Implementing Generic Functionality Driven by Custom Properties 25-12
25.3.4 What You May Need to Know 25-13
25.3.4.1 Determining the Attribute Kind at Runtime 25-13
25.3.4.2 Configuring Design Time Custom Property Names 25-13
25.3.4.3 Setting Custom Properties at Runtime 25-13
25.4 Creating Generic Extension Interfaces 25-14
25.5 Invoking Stored Procedures and Functions 25-16

25.5.1 Invoking Stored Procedures with No Arguments 25-16
25.5.2 Invoking Stored Procedure with Only IN Arguments 25-16
25.5.3 Invoking Stored Function with Only IN Arguments 25-18
25.5.4 Calling Other Types of Stored Procedures 25-19
25.6 Accessing the Current Database Transaction 25-21
25.7 Working with Libraries of Reusable Business Components 25-21
25.7.1 How To Create a Reusable Library of Business Components 25-22
25.7.2 How To Import a Package of Reusable Components from a Library 25-23
25.7.3 What Happens When You Import a Package of Reusable Components from a Library .. 25-24
25.7.4 What You May Need to Know 25-24
25.7.4.1 Adding Other Directories of Business Components to Project Source Path .. 25-24
25.7.4.2 Have to Close/Reopen to See Changes from a JAR 25-24
25.7.4.3 How to Remove an Imported Package from a Project 25-25
25.8 Customizing Business Components Error Messages 25-25
25.8.1 How to Customize Base ADF Business Components Error Messages 25-25
25.8.2 What Happens When You Customize Base ADF Business Components Error Messages 25-26
25.8.3 How to Customize Error Messages for Database Constraint Violations 25-27
25.8.4 How to Implement a Custom Constraint Error Handling Routine 25-27
25.8.4.1 Creating a Custom Database Transaction Framework Extension Class 25-27
25.8.4.2 Configuring an Application Module to Use a Custom Database Transaction Class . 25-28
25.9 Creating Extended Components Using Inheritance 25-29
25.9.1 How To Create a Component That Extends Another 25-30
25.9.2 What Happens When You Create a Component That Extends Another 25-30
25.9.2.1 Understanding an Extended Component's XML Descriptor 25-30
25.9.2.2 Understanding Java Code Generation for an Extended Component 25-31
25.9.3 What You May Need to Know 25-31
25.9.3.1 You Can Use Parent Classes and Interfaces to Work with Extended Components ... 25-31
25.9.3.2 Class Extends is Disabled for Extended Components 25-33
25.9.3.3 Interesting Aspects You Can Extend for Key Component Types 25-33
25.9.3.4 Extended Components Have Attribute Indices Relative to Parent 25-34
25.9.3.5 Design Time Limitations for Changing Extends After Creation 25-34
25.10 Substituting Extended Components In a Delivered Application 25-35
25.10.1 Extending and Substituting Components Is Superior to Modifying Code 25-35
25.10.2 How To Substitute an Extended Component 25-35
25.10.3 What Happens When You Substitute 25-36
25.10.4 Enabling the Substituted Components in the Base Application 25-37

26 Advanced Entity Object Techniques

26.1 Creating Custom, Validated Data Types Using Domains 26-1
26.1.1 What Are Domains? 26-2
26.1.2 How To Create a Domain 26-2
26.1.3 What Happens When You Create a Domain 26-2
26.1.4 What You May Need to Know 26-3

26.1.4.1 Using Domains for Entity and View Object Attributes 26-3
26.1.4.2 Validate Method Should Throw DataCreationException If Sanity Checks Fail 26-3
26.1.4.3 String Domains Aggregate a String Value... 26-4
26.1.4.4 Other Domains Extend Existing Domain Type... 26-4
26.1.4.5 Simple Domains are Immutable Java Classes 26-5
26.1.4.6 Creating Domains for Oracle Object Types When Useful................................. 26-5
26.1.4.7 Quickly Navigating to the Domain Class 26-6
26.1.4.8 Domains Get Packaged in the Common JAR.. 26-6
26.1.4.9 Entity and View Object Attributes Inherit Custom Domain Properties........... 26-7
26.1.4.10 Domain Settings Cannot Be Less Restrictive at Entity or View Level 26-7
26.2 Updating a Deleted Flag Instead of Deleting Rows.. 26-7
26.2.1 How to Update a Deleted Flag When a Row is Removed... 26-7
26.2.2 Forcing an Update DML Operation Instead of a Delete 26-8
26.3 Advanced Entity Association Techniques... 26-8
26.3.1 Modifying Association SQL Clause to Implement Complex Associations............. 26-8
26.3.2 Exposing View Link Accessor Attributes at the Entity Level 26-9
26.3.3 Optimizing Entity Accessor Access By Retaining the Row Set 26-9
26.4 Basing an Entity Object on a PL/SQL Package API 26-10
26.4.1 How to Create an Entity Object Based on a View.. 26-11
26.4.2 What Happens When You Create an Entity Object Based on a View 26-11
26.4.3 Centralizing Details for PL/SQL-Based Entities into a Base Class 26-11
26.4.4 Implementing the Stored Procedure Calls for DML Operations............................ 26-12
26.4.5 Adding Select and Lock Handling 26-13
26.4.5.1 Updating PLSQLEntityImpl Base Class to Handle Lock and Select............... 26-13
26.4.5.2 Implementing Lock and Select for the Product Entity..................................... 26-14
26.5 Basing an Entity Object on a Join View or Remote DBLink 26-17
26.6 Using Inheritance in Your Business Domain Layer... 26-17
26.6.1 Understanding When Inheritance Can be Useful... 26-18
26.6.2 How To Create Entity Objects in an Inheritance Hierarchy.................................... 26-19
26.6.2.1 Start By Identifying the Discriminator Column and Distinct Values 26-19
26.6.2.2 Identify the Subset of Attributes Relevant to Each Kind of Entity.................. 26-20
26.6.2.3 Creating the Base Entity Object in an Inheritance Hierarchy 26-20
26.6.2.4 Creating a Subtype Entity Object in an Inheritance Hierarchy 26-21
26.6.3 How to Add Methods to Entity Objects in an Inheritance Hierarchy 26-22
26.6.3.1 Adding Methods Common to All Entity Objects in the Hierarchy 26-22
26.6.3.2 Overriding Common Methods in a Subtype Entity 26-22
26.6.3.3 Adding Methods Specific to a Subtype Entity 26-22
26.6.4 What You May Need to Know.. 26-23
26.6.4.1 Sometimes You Need to Introduce a New Base Entity..................................... 26-23
26.6.4.2 Finding Subtype Entities by Primary Key 26-23
26.6.4.3 You Can Create View Objects with Polymorphic Entity Usages 26-23
26.7 Controlling Entity Posting Order to Avoid Constraint Violations................................ 26-24
26.7.1 Understanding the Default Post Processing Order 26-24
26.7.2 How Compositions Change the Default Processing Ordering............................... 26-24
26.7.3 Overriding postChanges() to Control Post Order.. 26-24
26.7.3.1 Observing the Post Ordering Problem First Hand.. 26-24
26.7.3.2 Forcing the Product to Post Before the ServiceRequest................................... 26-26

26.7.3.3 Understanding Associations Based on DBSequence-Valued Primary Keys .. 26-27
26.7.3.4 Refreshing References to DBSequence-Assigned Foreign Keys 26-28
26.8 Implementing Automatic Attribute Recalculation 26-29
26.9 Implementing Custom Validation Rules 26-31
26.9.1 How To Create a Custom Validation Rule 26-31
26.9.2 Adding a Design Time Bean Customizer for Your Rule 26-33
26.9.3 Registering and Using a Custom Rule in JDeveloper 26-34

27 Advanced View Object Techniques

27.1 Advanced View Object Concepts and Features 27-1
27.1.1 Using a Max Fetch Size to Only Fetch the First N Rows 27-1
27.1.2 Consistently Displaying New Rows in View Objects Based on the Same Entity.... 27-2
27.1.2.1 How View Link Consistency Mode Works 27-2
27.1.2.2 Understanding the Default View Link Consistency Setting and How to Change It.. 27-2
27.1.2.3 Using a RowMatch to Qualify Which New, Unposted Rows Get Added to a Row Set 27-3
27.1.2.4 Setting a Dynamic Where Clause Disables View Link Consistency 27-4
27.1.2.5 New Row from Other View Objects Added at the Bottom 27-4
27.1.2.6 New, Unposted Rows Added to Top of RowSet when Re-Executed 27-4
27.1.3 Understanding View Link Accessors Versus Data Model View Link Instances..... 27-4
27.1.3.1 Enabling a Dynamic Detail Row Set with Active Master/Detail Coordination 27-4
27.1.3.2 Accessing a Stable Detail Row Set Using View Link Accessor Attributes........ 27-5
27.1.3.3 Accessor Attributes Create Distinct Row Sets Based on an Internal View Object...... 27-5
27.1.4 Presenting and Scrolling Data a Page at a Time Using the Range 27-6
27.1.5 Efficiently Scrolling Through Large Result Sets Using Range Paging 27-7
27.1.5.1 Understanding How to Oracle Supports "TOP-N" Queries 27-7
27.1.5.2 How to Enable Range Paging for a View Object 27-8
27.1.5.3 What Happens When You Enable Range Paging 27-8
27.1.5.4 How are View Rows Cached When Using Range Paging? 27-9
27.1.5.5 How to Scroll to a Given Page Number Using Range Paging 27-9
27.1.5.6 Estimating the Number of Pages in the Row Set Using Range Paging 27-9
27.1.5.7 Accommodating Inserts and Deletes Using Auto Posting 27-9
27.1.5.8 Understanding the Tradeoffs of Using Range Paging Mode 27-9
27.1.6 Setting Up a Data Model with Multiple Masters 27-10
27.1.7 Understanding When You Can Use Partial Keys with findByKey() 27-11
27.1.8 Creating Dynamic Attributes to Store UI State 27-12
27.1.9 Working with Multiple Row Sets and Row Set Iterators 27-12
27.1.10 Optimizing View Link Accessor Access By Retaining the Row Set 27-13
27.2 Tuning Your View Objects for Best Performance 27-13
27.2.1 Use Bind Variables for Parameterized Queries 27-14
27.2.1.1 Use Bind Variables to Avoid Re-parsing of Queries 27-14
27.2.1.2 Use Bind Variables to Prevent SQL-Injection Attacks 27-14
27.2.2 Use Read-Only View Objects When Entity-Based Features Not Required 27-15
27.2.3 Use SQL Tracing to Identify Ill-Performing Queries 27-15

27.2.4 Consider the Appropriate Tuning Settings for Every View Object 27-16
27.2.4.1 Set the Database Retrieval Options Appropriately 27-16
27.2.4.2 Consider Whether Fetching One Row at a Time is Appropriate 27-16
27.2.4.3 Specify a Query Optimizer Hint if Necessary.. 27-17
27.2.5 Creating View Objects at Design Time.. 27-17
27.2.6 Use Forward Only Mode to Avoid Caching View Rows... 27-17
27.3 Using Expert Mode for Full Control Over SQL Query 27-18
27.3.1 How to Enable Expert Mode for Full SQL Control.. 27-18
27.3.2 What Happens When You Enable Expert Mode.. 27-18
27.3.3 What You May Need to Know... 27-19
27.3.3.1 You May Need to Perform Manual Attribute Mapping.................................. 27-19
27.3.3.2 Disabling Expert Mode Loses Any Custom Edits 27-20
27.3.3.3 Once In Expert Mode, Changes to SQL Expressions Are Ignored.................. 27-20
27.3.3.4 Don't Map Incorrect Calculated Expressions to Entity Attributes.................. 27-21
27.3.3.5 Expert Mode SQL Formatting is Retained... 27-22
27.3.3.6 Expert Mode Queries Are Wrapped as Inline Views....................................... 27-22
27.3.3.7 Disabling the Use of Inline View Wrapping at Runtime................................. 27-23
27.3.3.8 Enabling Expert Mode May Impact Dependent Objects 27-23
27.4 Working with Multiple Named View Criteria 27-23
27.4.1 Defining Named View Criteria.. 27-24
27.4.2 Applying One or More Named View Criteria... 27-25
27.4.3 Removing All Applied Named View Criteria... 27-25
27.4.4 Using the Named Criteria at Runtime.. 27-26
27.5 Performing In-Memory Sorting and Filtering of Row Sets.. 27-26
27.5.1 Understanding the View Object's Query Mode 27-27
27.5.2 Sorting View Object Rows In Memory 27-27
27.5.2.1 Combining setSortBy and setQueryMode for In-Memory Sorting................. 27-28
27.5.2.2 Extensibility Points for In-Memory Sorting... 27-30
27.5.3 Performing In-Memory Filtering with View Criteria.. 27-30
27.5.4 Performing In-Memory Filtering with RowMatch 27-32
27.5.4.1 Applying a RowMatch to a View Object.. 27-32
27.5.4.2 Using RowMatch to Test an Individual Row 27-34
27.5.4.3 How a RowMatch Affects Rows Fetched from the Database 27-34
27.6 Using View Objects to Work with Multiple Row Types... 27-34
27.6.1 What is a Polymorphic Entity Usage? 27-34
27.6.2 How To Create a View Object with a Polymorphic Entity Usage.......................... 27-35
27.6.3 What Happens When You Create a View Object with a Polymorphic Entity Usage........ 27-35
27.6.4 What You May Need to Know... 27-36
27.6.4.1 Your Query Must Limit Rows to Expected Entity Subtypes 27-36
27.6.4.2 Exposing Selected Entity Methods in View Rows Using Delegation 27-36
27.6.4.3 Creating New Rows With the Desired Entity Subtype.................................... 27-37
27.6.5 What are Polymorphic View Rows? 27-38
27.6.6 How to Create a View Object with Polymorphic View Rows................................. 27-39
27.6.7 What You May Need to Know... 27-40
27.6.7.1 Selecting Subtype-Specific Attributes in Extended View Objects................... 27-40
27.6.7.2 Delegating to Subtype-Specific Methods After Overriding the Entity Usage 27-41
27.6.7.3 Working with Different View Row Interface Types in Client Code 27-41

27.6.7.4 View Row Polymorphism and Polymorphic Entity Usage are Orthogonal... 27-42
27.7 Reading and Writing XML 27-43
27.7.1 How to Produce XML for Queried Data 27-43
27.7.2 What Happens When You Produce XML 27-44
27.7.3 What You May Need to Know........ 27-46
27.7.3.1 Controlling XML Element Names........ 27-46
27.7.3.2 Controlling Element Suppression for Null-Valued Attributes........ 27-46
27.7.3.3 Printing or Searching the Generated XML Using XPath 27-46
27.7.3.4 Using the Attribute Map For Fine Control Over Generated XML 27-47
27.7.3.5 Use the Attribute Map Approach with Bi-Directional View Links........ 27-48
27.7.3.6 Transforming Generated XML Using an XSLT Stylesheet........ 27-48
27.7.3.7 Generating XML for a Single Row 27-50
27.7.4 How to Consume XML Documents to Apply Changes........ 27-50
27.7.5 What Happens When You Consume XML Documents........ 27-50
27.7.5.1 How ViewObject.readXML() Processes an XML Document 27-50
27.7.5.2 Using readXML() to Processes XML for a Single Row........ 27-51
27.8 Using Programmatic View Objects for Alternative Data Sources 27-54
27.8.1 How to Create a Read-Only Programmatic View Object 27-54
27.8.2 How to Create an Entity-Based Programmatic View Object........ 27-54
27.8.3 Key Framework Methods to Override for Programmatic View Objects........ 27-55
27.8.4 How to Create a View Object on a REF CURSOR 27-56
27.8.4.1 The Overridden create() Method 27-56
27.8.4.2 The Overridden executeQueryForCollection() Method 27-57
27.8.4.3 The Overridden createRowFromResultSet() Method 27-57
27.8.4.4 The Overridden hasNextForCollectionMethod() 27-58
27.8.4.5 The Overridden releaseUserDataForCollection() Method 27-58
27.8.4.6 The Overridden getQueryHitCount() Method 27-58
27.8.5 Populating a View Object from Static Data 27-59
27.8.5.1 Basing Lookup View Object on SRStaticDataViewObjectImpl 27-61
27.8.5.2 Creating a View Object Based on Static Data from a Properties File........ 27-61
27.8.5.3 Creating Your Own View Object with Static Data 27-63
27.9 Creating a View Object with Multiple Updatable Entities 27-63
27.10 Declaratively Preventing Insert, Update, and Delete........ 27-65

28 Application Module State Management

28.1 Understanding Why State Management is Necessary 28-1
28.1.1 Examples of Multi-Step Tasks........ 28-1
28.1.2 Stateless HTTP Protocol Complicates Stateful Applications........ 28-2
28.1.3 How Cookies Are Used to Track a User Session........ 28-2
28.1.4 Performance and Reliability Impact of Using HttpSession........ 28-3
28.2 The ADF Business Components State Management Facility 28-4
28.2.1 Basic Architecture of the State Management Facility........ 28-5
28.2.2 Understanding When Passivation and Activation Occurs........ 28-6
28.2.3 How Passivation Changes When Optional Failover Mode is Enabled 28-7
28.3 Controlling the State Management Release Level........ 28-7
28.3.1 Supported Release Levels........ 28-8
28.3.2 Setting the Release Level at Runtime........ 28-9

28.3.2.1 Setting Release Level in a JSF Backing Bean 28-9
28.3.2.2 Setting Release Level in an ADF PagePhaseListener 28-10
28.3.2.3 Setting Release Level in an ADF PageController 28-11
28.3.2.4 Setting Release Level in an Custom ADF PageLifecycle 28-11
28.4 What State Is Saved and When is It Cleaned Up? 28-12
28.4.1 What State is Saved? 28-12
28.4.2 Where is the State Saved? 28-13
28.4.2.1 How Database-Backed Passivation Works 28-13
28.4.2.2 Controlling the Schema Where the State Management Table Resides 28-13
28.4.2.3 Configuring the Type of Passivation Store 28-14
28.4.3 When is the State Cleaned Up? 28-14
28.4.3.1 Previous Snapshot Removed When Next One Taken 28-14
28.4.3.2 Passivation Snapshot Removed on Unmanaged Release 28-15
28.4.3.3 Passivation Snapshot Retained in Failover Mode 28-15
28.4.4 Approaches for Timing Out the HttpSession 28-15
28.4.4.1 Configuring the Implicit Timeout Due to User Inactivity 28-15
28.4.4.2 Coding an Explicit HttpSession Timeout 28-16
28.4.5 Cleaning Up Temporary Storage Tables 28-16
28.5 Managing Custom User Specific Information 28-17
28.6 Managing State for Transient View Objects 28-19
28.7 Using State Management for Middle-Tier Savepoints 28-20
28.8 Testing to Ensure Your Application Module is Activation-Safe 28-20
28.8.1 Understanding the jbo.ampool.doampooling Configuration Parameter 28-20
28.8.2 Disabling Application Module Pooling to Test Activation 28-20
28.9 Caveats Regarding Pending Database State 28-21
28.9.1 Web Applications Should Use Optimistic Locking 28-21
28.9.2 Use PostChanges Only During the Current Request 28-21
28.9.3 Pending Database State Across Requests Requires Reserved Level 28-22
28.9.4 Connection Pooling Prevents Pending Database State 28-22

29 Understanding Application Module Pooling

29.1 Overview of Application Module Pooling 29-1
29.2 Lifecycle of a Web Page Request Using Oracle ADF and JSF 29-2
29.3 Understanding Configuration Property Scopes 29-4
29.4 Setting Pool Configuration Parameters 29-5
29.4.1 Setting Configuration Properties Declaratively 29-5
29.4.2 Setting Configuration Properties as System Parameters 29-7
29.4.3 Programmatically Setting Configuration Properties 29-7
29.5 How Many Pools are Created, and When? 29-9
29.5.1 Application Module Pools 29-9
29.5.2 Database Connection Pools 29-9
29.5.3 Understanding Application Module and Connection Pools 29-10
29.5.3.1 Single Oracle Application Server Instance, Single OC4J Container, Single JVM 29-10
29.5.3.2 Multiple Oracle Application Server Instances, Single OC4J Container, Multiple JVMs 29-11
29.6 Application Module Pool Parameters 29-12

29.6.1 Pool Behavior Parameters ... 29-12
29.6.2 Pool Sizing Parameters ... 29-13
29.6.3 Pool Cleanup Parameters ... 29-13
29.7 Database Connection Pool Parameters ... 29-15
29.8 How Database and Application Module Pools Cooperate ... 29-17
29.9 Database User State and Pooling Considerations ... 29-18
29.9.1 How Often prepareSession() Fires When jbo.doconnectionpooling = false ... 29-18
29.9.2 Setting Database User State When jbo.doconnectionpooling = true ... 29-18
29.9.3 Understanding How the SRDemo Application Sets Database State ... 29-19

30 Adding Security to an Application

30.1 Introduction to Security in Oracle ADF Web Applications ... 30-1
30.2 Specifying the JAZN Resource Provider ... 30-2
30.2.1 How To Specify the Resource Provider ... 30-2
30.2.2 What You May Need to Know About Oracle ADF Security and Resource Providers ... 30-3
30.3 Configuring Authentication Within the web.xml File ... 30-4
30.3.1 How to Enable J2EE Container-Managed Authentication ... 30-4
30.3.2 What Happens When You Use Security Constraints without Oracle ADF Security ... 30-8
30.3.3 How to Enable Oracle ADF Authentication ... 30-9
30.3.4 What Happens When You Use Security Constraints with Oracle ADF ... 30-11
30.4 Configuring the ADF Business Components Application to Use Container-Managed Security 30-12
30.4.1 How to Configure Security in an Oracle ADF Business Components Application ... 30-12
30.4.2 What Happens When You Configure Security in an ADF Business Components Application 30-14
30.4.3 What You May Need to Know About the ADF Business Components Security Property 30-14
30.5 Creating a Login Page ... 30-15
30.5.1 Wiring the Login and Error Pages ... 30-18
30.5.2 What Happens When You Wire the Login and Error Pages ... 30-19
30.6 Creating a Logout Page ... 30-19
30.6.1 Wiring the Logout Action ... 30-21
30.6.2 What Happens When You Wire the Logout Action ... 30-22
30.7 Implementing Authorization Using Oracle ADF Security ... 30-23
30.7.1 Configuring the Application to Use Oracle ADF Security Authorization ... 30-25
30.7.1.1 How to Configure Oracle ADF Security Authorization ... 30-25
30.7.1.2 What Happens When You Configure An Application to Use Oracle ADF Security . 30-25
30.7.1.3 What You May Need to Know About the Authorization Property ... 30-26
30.7.2 Setting Authorization on ADF Binding Containers ... 30-26
30.7.3 Setting Authorization on ADF Iterator Bindings ... 30-26
30.7.4 Setting Authorization on ADF Attribute and MethodAction Bindings ... 30-27
30.7.5 What Happens When Oracle ADF Security Handles Authorization ... 30-27
30.8 Implementing Authorization Programmatically ... 30-28
30.8.1 Making User Information EL Accessible ... 30-29

30.8.1.1 Creating a Class to Manage Roles 30-29
30.8.1.2 Creating a Managed Bean for the Security Information 30-31

31 Creating Data Control Adapters

31.1 Introduction to the Simple CSV Data Control Adapter 31-1
31.2 Overview of Steps to Create a Data Control Adapter 31-2
31.3 Implement the Abstract Adapter Class 31-3
31.3.1 Location of JAR Files 31-3
31.3.2 Abstract Adapter Class Outline 31-3
31.3.3 Complete Source for the SampleDCAdapter Class 31-4
31.3.4 Implementing the initialize Method 31-7
31.3.5 Implementing the invokeUI Method 31-7
31.3.6 Implementing the getDefinition Method 31-8
31.4 Implement the Data Control Definition Class 31-9
31.4.1 Location of JAR Files 31-9
31.4.2 Data Control Definition Class Outline 31-9
31.4.3 Complete Source for the SampleDCDef Class 31-10
31.4.4 Creating a Default Constructor 31-13
31.4.5 Collecting Metadata from the User 31-13
31.4.6 Defining the Structure of the Data Control 31-14
31.4.7 Creating an Instance of the Data Control 31-15
31.4.8 Setting the Metadata for Runtime 31-16
31.4.9 Setting the Name for the Data Control 31-17
31.5 Implement the Data Control Class 31-18
31.5.1 Location of JAR Files 31-18
31.5.2 Data Control Class Outline 31-19
31.5.3 Complete Source for the SampleDataControl Class 31-19
31.5.4 Implementing the invokeOperation Method 31-22
31.5.4.1 About Calling processResult 31-24
31.5.4.2 Return Value for invokeOperation 31-24
31.5.5 Implementing the getName Method 31-24
31.5.6 Implementing the release Method 31-25
31.5.7 Implementing the getDataProvider Method 31-25
31.6 Create any Necessary Supporting Classes 31-25
31.7 Create an XML File to Define Your Adapter 31-26
31.8 Build Your Adapter 31-27
31.9 Package and Deploy Your Adapter to JDeveloper 31-28
31.10 Location of Javadoc Information 31-30
31.11 Contents of Supporting Files 31-30
31.11.1 sampleDC.xsd 31-30
31.11.2 CSVHandler Class 31-30
31.11.3 CSVParser 31-37

32 Working Productively in Teams

32.1 Using CVS with an ADF Project 32-1
32.1.1 Choice of Internal or External CVS Client 32-1
32.1.2 Preference Settings 32-1

32.1.3 File Dependencies 32-1
32.1.4 Use Consistent Connection Definition Names 32-2
32.1.5 General Advice for Committing ADF Work to CVS 32-2
32.1.5.1 Other Version Control Tips and Techniques 32-2
32.1.6 Check Out or Update from the CVS Repository 32-3
32.1.7 Special Consideration when Manually Adding Navigation Rules to the faces-config.xml File 32-3
32.2 General Advice for Using CVS with JDeveloper 32-3
32.2.1 Team-Level Activities 32-3
32.2.2 Developer-Level Activities 32-4
32.2.2.1 Typical Workflow When Checking Your Work Into CVS 32-4
32.2.2.2 Handling CVS Repository Configuration Files 32-5
32.2.2.3 Advice for Merge Conflicts in ADF Business Components Projects 32-5

33 Working with Web Services

33.1 What are Web Services 33-1
33.1.1 SOAP 33-2
33.1.2 WSDL 33-2
33.1.3 UDDI 33-2
33.1.4 Web Services Interoperability 33-3
33.2 Creating Web Service Data Controls 33-4
33.2.1 How to Create a Web Service Data Control 33-4
33.3 Securing Web Service Data Controls 33-5
33.3.1 WS-Security Specification 33-5
33.3.2 Creating and Using Keystores 33-6
33.3.2.1 How to Create a Keystore 33-6
33.3.2.2 How to Request a Certificate 33-7
33.3.2.3 How to Export a Public Key Certificate 33-8
33.3.3 Defining Web Service Data Control Security 33-9
33.3.3.1 How to Set Authentication 33-9
33.3.3.2 How to Set Digital Signatures 33-12
33.3.3.3 How to Set Encryption and Decryption 33-13
33.3.3.4 How to Use a Key Store 33-13
33.4 Publishing Application Modules as Web Services 33-14
33.4.1 How to Enable the J2EE Web Service Option for an Application Module 33-14
33.4.2 What Happens When You Enable the J2EE Web Service Option 33-15
33.4.3 What You May Need to Know About Deploying an Application Module as a Web Service 33-16
33.4.4 What You May Need to Know About Data Types Supported for Web Service Methods 33-16
33.5 Calling a Web Service from an Application Module 33-16
33.5.1 Understanding the Role of the Web Services Description Language Document.. 33-16
33.5.2 Understanding the Role of the Web Service Proxy Class 33-17
33.5.3 How to Call a Web Service from an Application Module 33-17
33.5.3.1 Creating a Web Service Proxy Class for a Web Service 33-17
33.5.3.2 Understanding the Generated Web Service Proxy 33-18
33.5.3.3 Calling a Web Service Method Using the Web Service Proxy Class 33-19

33.5.4 What Happens When You Call a Web Service from an Application Module 33-19
33.5.5 What You May Need to Know 33-19
33.5.5.1 Use a Try/Catch Block to Handle Web Service Exceptions 33-19
33.5.5.2 Web Services are Do Not Share a Transaction with the Application Module 33-20
33.5.5.3 Setting Browser Proxy Information 33-20

34 Deploying ADF Applications

34.1 Introduction to Deploying ADF Applications 34-1
34.2 Deployment Steps 34-2
34.3 Deployment Techniques 34-8
34.4 Deploying Applications Using Ant 34-9
34.5 Deploying the SRDemo Application 34-9
34.6 Deploying to Oracle Application Server 34-9
34.6.1 Oracle Application Server Versions Supported 34-10
34.6.2 Oracle Application Server Release 2 (10.1.2) Deployment Notes 34-10
34.6.3 Oracle Application Server Deployment Methods 34-11
34.6.4 Oracle Application Server Deployment to Test Environments ("Automatic Deployment") 34-11
34.6.5 Oracle Application Server Deployment to Clustered Topologies 34-12
34.7 Deploying to JBoss 34-12
34.7.1 JBoss Versions Supported 34-12
34.7.2 JBoss Deployment Notes 34-12
34.7.3 JBoss Deployment Methods 34-14
34.8 Deploying to WebLogic 34-14
34.8.1 WebLogic Versions Supported 34-14
34.8.2 WebLogic Versions 8.1 and 9.0 Deployment Notes 34-14
34.8.3 WebLogic 8.1 Deployment Notes 34-15
34.8.4 WebLogic 9.0 Deployment Notes 34-15
34.8.5 WebLogic Deployment Methods 34-15
34.9 Deploying to WebSphere 34-16
34.9.1 WebSphere Versions Supported 34-16
34.9.2 WebSphere Deployment Notes 34-16
34.9.3 WebSphere Deployment Methods 34-16
34.10 Deploying to Tomcat 34-17
34.10.1 Tomcat Versions Supported 34-17
34.10.2 Tomcat Deployment Notes 34-17
34.11 Deploying to Application Servers That Support JDK 1.4 34-17
34.11.1 Switching Embedded OC4J to JDK 1.4 34-18
34.12 Installing ADF Runtime Library on Third-Party Application Servers 34-18
34.12.1 Installing the ADF Runtime Libraries from JDeveloper 34-18
34.12.2 Configuring WebSphere 6.0.1 to Run ADF Applications 34-21
34.12.2.1 Source for install_adflibs_1013.sh Script 34-22
34.12.2.2 Source for install_adflibs_1013.cmd Script 34-24
34.12.3 Installing the ADF Runtime Libraries Manually 34-26
34.12.3.1 Installing the ADF Runtime Libraries from a Zip File 34-28
34.12.4 Deleting the ADF Runtime Library 34-29
34.13 Verifying Deployment and Troubleshooting 34-29

34.13.1 How to Test Run Your Application 34-30
34.13.2 "Class Not Found" or "Method Not Found" Errors 34-30
34.13.3 Application Is Not Using data-sources.xml File on Target Application Server 34-30
34.13.4 Using jazn-data.xml with the Embedded OC4J Server 34-31
34.13.5 Error "JBO-30003: The application pool failed to check out an application module due to the following exception" 34-31

Part V Appendices

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-2
A.2.2 Oracle ADF Data Binding Files A-3
A.2.3 Oracle ADF Faces and Web Configuration Files A-3
A.3 ADF File Syntax Diagram A-4
A.4 bc4j.xcfg A-5
A.5 DataBindings.cpx A-6
A.5.1 DataBindings.cpx Syntax A-7
A.5.2 DataBindings.cpx Sample A-8
A.6 <pageName>PageDef.xml A-8
A.6.1 PageDef.xml Syntax A-9
A.6.2 PageDef.xml Sample for Attributes of a View Object A-18
A.6.3 PageDef.xml Sample for the Entire View Object A-19
A.7 web.xml A-20
A.7.1 Tasks Supported by the web.xml File A-21
A.7.1.1 Configuring for State Saving A-21
A.7.1.2 Configuring for Application View Caching A-22
A.7.1.3 Configuring for Debugging A-22
A.7.1.4 Configuring for File Uploading A-23
A.7.1.5 Configuring for ADF Model Binding A-24
A.7.1.6 Other Context Configuration Parameters for JSF A-24
A.7.1.7 What You May Need to Know A-25
A.8 j2ee-logging.xml A-25
A.8.1 Tasks Supported by the j2ee-logging.xml A-25
A.8.1.1 Change the Logging Level for Oracle ADF Packages A-25
A.8.1.2 Redirect the Log Output A-26
A.8.1.3 Change the Location of the Log File A-26
A.9 faces-config.xml A-26
A.9.1 Tasks Supported by the faces-config.xml A-27
A.9.1.1 Registering a Render Kit for ADF Faces Components A-27
A.9.1.2 Registering a Phase Listener for ADF Binding A-27
A.9.1.3 Registering a Message Resource Bundle A-28
A.9.1.4 Configuring for Supported Locales A-28
A.9.1.5 Creating Navigation Rules and Cases A-29
A.9.1.6 Registering Custom Validators and Converters A-30
A.9.1.7 Registering Managed Beans A-30

A.10 adf-faces-config.xml A-32
A.10.1 Tasks Supported by adf-faces-config.xml A-33
A.10.1.1 Configuring Accessibility Levels A-33
A.10.1.2 Configuring Currency Code and Separators for Number Groups and Decimals A-33
A.10.1.3 Configuring For Enhanced Debugging Output A-34
A.10.1.4 Configuring for Client-Side Validation and Conversion A-34
A.10.1.5 Configuring the Language Reading Direction A-34
A.10.1.6 Configuring the Skin Family A-35
A.10.1.7 Configuring the Output Mode A-35
A.10.1.8 Configuring the Number of Active ProcessScope Instances A-35
A.10.1.9 Configuring the Time Zone and Year Offset A-35
A.10.1.10 Configuring a Custom Uploaded File Processor A-36
A.10.1.11 Configuring the Help Site URL A-36
A.10.1.12 Retrieving Configuration Property Values From adf-faces-config.xml A-36
A.11 adf-faces-skins.xml A-37
A.11.1 Tasks Supported by adf-faces-skins.xml A-37

B Reference ADF Binding Properties

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

C ADF Equivalents of Common Oracle Forms Triggers

C.1 Validation & Defaulting (Business Logic) C-1
C.2 Query Processing C-2
C.3 Database Connection C-3
C.4 Transaction "Post" Processing (Record Cache) C-3
C.5 Error Handling C-4

D Most Commonly Used ADF Business Components Methods

D.1 Most Commonly Used Methods in the Client Tier D-1
D.1.1 ApplicationModule Interface D-2
D.1.2 Transaction Interface D-3
D.1.3 ViewObject Interface D-3
D.1.4 RowSet Interface D-5
D.1.5 RowSetIterator Interface D-5
D.1.6 Row Interface D-7
D.1.7 StructureDef Interface D-7
D.1.8 AttributeDef Interface D-8
D.1.9 AttributeHints Interface D-9
D.2 Most Commonly Used Methods In the Business Service Tier D-9
D.2.1 Controlling Custom Java Files For Your Components D-9
D.2.2 ApplicationModuleImpl Class D-10
D.2.2.1 Methods You Typically Call on ApplicationModuleImpl D-10
D.2.2.2 Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass ... D-11
D.2.2.3 Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass D-11

D.2.3 DBTransactionImpl2 Class D-12
D.2.3.1 Methods You Typically Call on DBTransaction..... D-13
D.2.3.2 Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass ... D-13
D.2.4 EntityImpl Class..... D-14
D.2.4.1 Methods You Typically Call on EntityImpl..... D-14
D.2.4.2 Methods You Typically Write in Your Custom EntityImpl Subclass D-15
D.2.4.3 Methods You Typically Override on EntityImpl..... D-15
D.2.5 EntityDefImpl Class D-17
D.2.5.1 Methods You Typically Call on EntityDefImpl D-17
D.2.5.2 Methods You Typically Write on EntityDefImpl..... D-17
D.2.5.3 Methods You Typically Override on EntityDefImpl D-18
D.2.6 ViewObjectImpl Class..... D-18
D.2.6.1 Methods You Typically Call on ViewObjectImpl..... D-18
D.2.6.2 Methods You Typically Write in Your Custom ViewObjectImpl Subclass D-19
D.2.6.3 Methods You Typically Override in Your Custom ViewObjectImpl Subclass D-20
D.2.7 ViewRowImpl Class..... D-20
D.2.7.1 Methods You Typically Call on ViewRowImpl..... D-20
D.2.7.2 Methods You Typically Write on ViewRowImpl D-21
D.2.7.3 Methods You Typically Override in Your Custom ViewRowImpl Subclass ... D-21
D.2.8 Setting Up Your Own Layer of Framework Base Classes D-22

E ADF Business Components J2EE Design Pattern Catalog

E.1 J2EE Design Patterns Implemented by ADF Business Components..... E-1

Index

Preface

Welcome to the Oracle Application Development Framework Developer's Guide for Forms/4GL Developers!

Audience

This manual is intended 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.

> **Note:** If you are already an advanced J2EE developer and maximally declarative development is not a top priority, Oracle offers a parallel developer's guide for Oracle ADF that may be more appropriate for you. In particular, if you prefer building your business service layer using EJB session beans, POJO classes, an O/R mapping layer, and your own J2EE design pattern code, this other guide helps you understand how to use Oracle ADF with these business service implementation technologies. You can access this alternative guide from JDeveloper 10g product center on OTN at `http://otn.oracle.com/jdev/`.

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 documents:

- *Oracle Application Development Framework Developer's Guide*, an alternative guide for advanced J2EE developers who wish to work with EJB seesion beans in their application's business service layer, among other technologies.
- *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, "Overview of Development Process with Oracle ADF and JSF"
- Chapter 3, "Oracle ADF Service Request Demo Overview"

1 Introduction to Oracle ADF Applications

This chapter describes the architecture and key functionality of the Oracle Application Development Framework (Oracle ADF).

This chapter includes the following sections:

- Section 1.1, "Introduction to Oracle ADF"
- Section 1.2, "Framework Architecture and Supported Technologies"
- Section 1.3, "Declarative Development with Oracle ADF and JavaServer Faces"
- Section 1.4, "Highlights of Additional ADF Features"

1.1 Introduction to Oracle ADF

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 10g 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.

1.2 Framework Architecture and Supported Technologies

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

- The model layer represents the data values related to the current page
- The view layer contains the UI pages used to view/modify that data
- The controller layer processes user input and determines page navigation
- The business service layer handles data access and encapsulates business logic

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

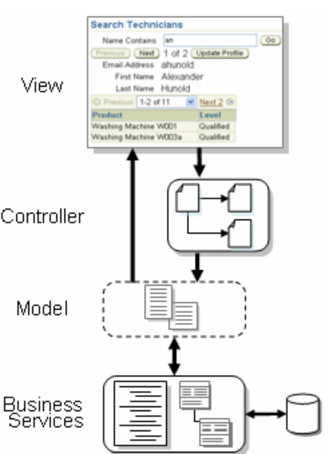
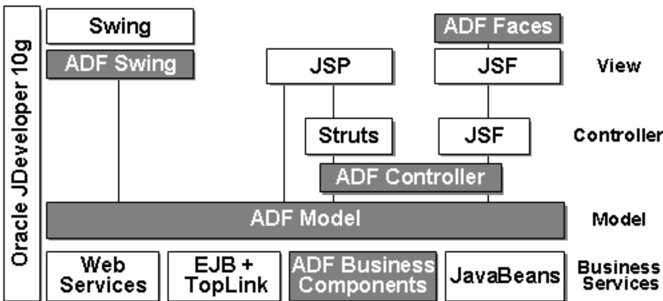


Figure 1–2 illustrates where each ADF module fits in this architecture. The core module in the framework is Oracle ADF Model, a declarative data binding facility that implements the JSR-227 specification. The Oracle ADF Model layer enables a unified approach to bind any user interface to any business service with no code. The other modules Oracle ADF comprises are:

- Oracle ADF Business Components, which simplifies building business services for developers familiar with 4GL tools like Oracle Forms, PeopleTools, SiebelTools, Visual Studio, and others
- Oracle ADF Faces, which offers a rich library of UI components for web applications built with JavaServer Faces (JSF)
- Oracle ADF Swing, which extends Oracle ADF Model to desktop applications built with Swing
- Oracle ADF Controller, which integrates Struts and JSF with Oracle ADF Model

Figure 1–2 Simple ADF Architecture



1.2.1 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 interfacing with web browsers and 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.

1.2.2 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.

1.2.3 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:

- ADF application modules

  These service components expose an updateable dataset of SQL query results with automatic business rules enforcement.

- 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, you need only supply the provider URL and optional parameters and you can work with the data.

- JavaBeans and Enterprise JavaBeans (EJB) Session Beans

  When necessary, you can easily work with any Java-based service classes as well, including EJBs that support transactions.

1.2.4 Recommended Technologies for Enterprise 4GL Developers

For enterprise 4GL developers building new web applications, Oracle recommends using JavaServer Faces for the view and controller layers, and ADF Business Components for the business service implementation. This combination offers you the same productive J2EE technology stack that over 4000 of Oracle's own enterprise 4GL developers use every day to build the Oracle E-Business Suite. Since its initial release in 1999, several thousand external customers and partners have built and deployed successful Oracle ADF-based applications as well. Both now and in the future, Oracle and others are betting their business on Oracle ADF with ADF Business Components.

1.3 Declarative Development with Oracle ADF and JavaServer Faces

Even as a seasoned user of rapid application development tools like Oracle Forms, PeopleTools, SiebelTools, or Visual Basic, you've likely already had *some* exposure to Java 2 Enterprise Edition. Maybe your interest in J2EE began as it did for Oracle's own E-Business Suite division, while evaluating standards-based architectures for self-service web applications to complement traditional desktop UIs for professional users. Or perhaps you've simply been studying in your spare time to expand your skill set and reinforce your resume.

Whatever your point of departure, initial impressions of your J2EE experience probably parallel those of fellow 4GL users. You understand how J2EE can increase flexibility, reuse, and choice, but its many "moving parts" leave you a little puzzled. Accustomed to banging out screen after screen with your familiar 4GL tools, you fear it's more likely your *head* you'll be banging if you have to write all that J2EE code by hand. You'll be glad to learn that all of the 4GL features you are familiar with have an analog in JDeveloper 10g with Oracle ADF, including:

- Declarative data access and validation
- Declarative user interface design and page navigation
- Declarative data binding
- Simple, event-driven approach to add custom logic where needed

When you use Oracle ADF's XML- and Java-based declarative development facilities in Oracle JDeveloper 10*g*, you also benefit from design-time error highlighting, context-sensitive editing assistance, and compile-time error checking.

1.3.1 Declarative Data Access and Validation with ADF Business Components

When building service-oriented J2EE applications, you implement your core business logic as one or more business services. These backend services provide clients a way to query, insert, update, and delete business data as required while enforcing appropriate business rules. ADF Business Components are prebuilt application objects that accelerate the job of delivering and maintaining high performance, richly-functional, data-centric services. They provide you a ready-to-use implementation of all the J2EE design patterns and best practices that Oracle's own application developers have found are needed to do the job right. By leveraging this road-tested code, you focus on your own application-specific logic instead of reinventing the wheel.

As illustrated in Figure 1–3, Oracle ADF provides the following key components to simplify building database-centric business services:

- Entity object

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

- Application module

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

- View object

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

Tip: Oracle Forms developers will immediately recognize this combined functionality as the same set of data-centric features provided by the form, data blocks, record manager, and form-level procedures/functions. The key difference in ADF is that the user interface is cleanly separated from data access and validation functionality. For more information, see Section 4.3.1, "Familiar Concepts for Oracle Forms Developers"

Figure 1–3 ADF Business Components Simplify Data Access and Validation

1.3.2 Declarative User Interface Design and Page Navigation with JavaServer Faces

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.

1.3.2.1 Declarative User Interface Design with JSF

The industry experts who collaborated on the JavaServer Faces standard incorporated numerous declarative development techniques into the design. For example, in JSF you use a simple *expression language* to work with the information you want to present. Example expressions look like `#{UserList.selectedUsers}` to reference a set of selected users, `#{user.name}` to reference a particular user's name, or `#{user.role == 'manager'}` to evaluate whether a user is a manager or not. At runtime, a generic expression evaluator returns the `List`, `String`, and `boolean` value of these

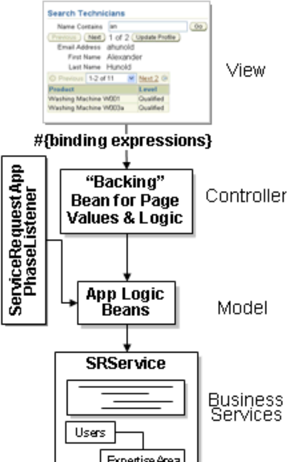
respective expressions, automating access to the individual objects and their properties without requiring code. This declarative expression language, nicknamed "EL," originally debuted as part of the JSTL tag library and an improved version is now incorporated in the current JSP and JSF standards.

At runtime, the value of a JSF UI 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}`. 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 set of objects you need to display is empty by using a boolean-valued expression like `#{not empty UserList.selectedUsers}` in the UI component's `rendered` attribute. If the list of selected users in the object named `UserList` is empty, 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 object creation mechanism. To use it, you configure the Java objects you need to use in a JSF `faces-config.xml` file. These objects are known as *managed beans* since they have properties that follow the JavaBeans specification and since the JSF runtime manages instantiating them on demand when any EL expression references them for the first time. JSF also offers a declarative mechanism to set the properties of these beans as they are first created. 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.

Figure 1–4 shows how JSF managed beans serve two primary roles.

Figure 1–4 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.

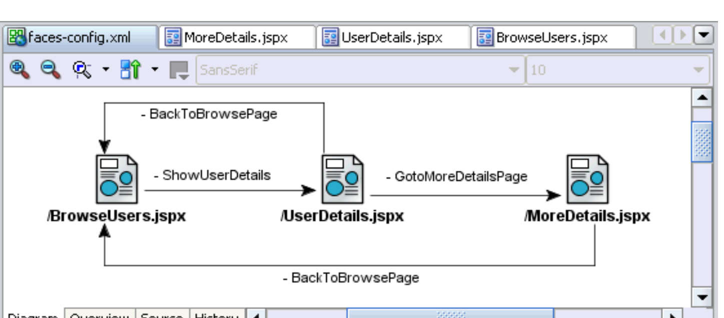
Think of managed beans that aren't playing the role of a page's backing bean simply as "application logic beans." They contain code and properties that are not specific to a single page. While not restricted to this purpose, they sometimes function as business service wrappers to cache method results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

1.3.2.2 Declarative Page Navigation with JSF

In addition to declarative UI design, JSF also provides a mechanism to declaratively define page navigation rules. Developers define these rules by specifying logical names for the legal navigation "outcomes" of a user's interaction with a page. For example, while on a `UserDetails.jspx` page modifying details of an account, an end user may interact with the page by clicking a **Save** button. The logical navigation outcomes of this interaction might be to go to a `MoreDetails.jspx` page to see more details, or else go back to a `BrowseUsers.jspx` page to see a list of user accounts. As shown in Figure 1–5, you might pick names like `GotoMoreDetailsPage` and `BackToBrowsePage` to describe these two outcomes. The navigation rule information is saved along with other configuration information in the `faces-config.xml` file, and at runtime JSF handles the page navigation based on these logical outcome names.

Figure 1–5 Visualizing JSF Navigation Rules in JDeveloper's Page Flow Diagram



1.3.3 Declarative Data Binding with Oracle ADF Model Layer

The Oracle ADF Model layer uses XML configuration files to drive generic data binding features. 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 the service's operations and data collections, including information about the properties, methods, and types involved. At design time, visual tools like JDeveloper can leverage the standard service metadata to simplify binding UI components to any data control operation or data collection. 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 business 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.

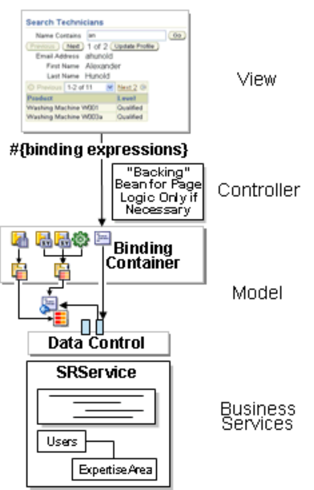
Declarative bindings abstract the details of accessing data from data collections in a data control and of invoking its operations. There are three basic kinds of declarative binding objects that automate the key aspects of data binding that all enterprise applications require:

- Iterator bindings to bind to an iterator that tracks the current row in a data collection
- Value bindings to connect UI components to attributes in a data collection
- Action bindings to invoke custom or built-it operations on a data control or its data collections

Iterator bindings simplify building user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information. UI components that display data use value bindings. Value bindings range from the most basic variety that work with a simple text field to more sophisticated list, table, and tree bindings that support the additional needs of list, table, and tree UI controls. An action binding is used by UI components like hyperlinks or buttons to invoke built-in or custom operations on data collections or a data control without writing code.

Figure 1–6 illustrates the architecture of a JSF application when you leverage ADF Model for declarative data binding. By combining Oracle ADF Model with JavaServer Faces, you avoid having to write a lot of the typical managed bean code (shown above in Figure 1–4) that would be required for real-world applications.

Figure 1–6 Architecture of a JSF Application Using ADF Model Data Binding



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 avoid having to write 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.

1.3.4 Simple, Event-Driven Approach to Add Custom Logic

Oracle ADF provides you a lot of declarative functionality and spares you from having to master and implement all the J2EE design patterns required for enterprise J2EE applications. The Java code you *do* write, as has always been the case in your familiar 4GL tools, is only the code that's unique to your specific business application or user interface interactions. The web-based applications you'll build using Oracle ADF have a cleanly layered architecture consisting of:

- JSF pages with UI components in the *view* layer
- JSF backing beans containing UI event handling and page flow logic in the *controller* layer
- ADF Model declarative data binding in *model* layer
- ADF Business Components implementing data access and validation in the *business services* layer

In practice, since the UI components are represented as nested tags in a JSP page, and declarative bindings are captured in the XML page definition file, the two layers in which you will write application-specific code are the controller layer and the business services layer.

1.3.4.1 Simple-to-Handle Events in the Controller Layer

When the end user interacts with JSF UI components in the browser, they raise events you can handle. For example, the user might:

- Click on a button or a hyperlink
- Change the selection in a dropdown list
- Expand or collapse a level in a tree display

You can write event-handling code in a JSF backing bean for a page that is triggered when these kind of UI events occur. These event handlers will look like the following:

Example 1–1 Handling a button click event in a backing bean

```
public String saveButton_onClick() {
  // Add event code here...
  if (userRequiresMoreDetailsPage()) {
    return "GotoMoreDetailsPage";
  else {
    return "BackToBrowsePage";
  }
}
```

Example 1–2 Handling a dropdown list's value change event in a backing bean

```
public void roleList_onListChanged(ValueChangeEvent event) {
  // Add event code here...
}
```

Example 1–3 Handling a tree control's expand/collapse event in a backing bean

```
public void mgmtChainTree_onExpandCollapse(DisclosureEvent event) {
  // Add event code here...
}
```

> **Tip:** These three JSF event handling methods are similar to the UI-related triggers in Oracle Forms named `WHEN-BUTTON-PRESSED`, `WHEN-LIST-CHANGED`, and `WHEN-TREE-NODE-EXPANDED` respectively.

The code you add inside these methods gets automatically triggered by the JSF runtime to handle the event when the user interacts with the related UI component. They will typically contain code that manipulates UI components on the page — hiding/showing them, changing colors, and so on — or that performs conditional page navigation logic. Should you want to handle some logic completely in the browser client, in addition to server-side JSF event-handling code, you can associate client-side JavaScript with UI components as well.

1.3.4.2 Simple-to-Handle Events in the Business Service Layer

When the user works with data in the UI, the ADF Model layer coordinates any changes with the appropriate data collection in the data control. For example, in a web-based customer service portal application like the one you'll study in this guide, end-users of the system with different roles might:

- Create a new service request to ask for assistance in resolving a problem
- Update a service request to assign it to a new technician
- Try to delete a service request

In response to these events, your business requirements might demand that:

- The request date of the service request be defaulted to the current time, rounded to the nearest ten-minute interval
- An appropriate technician be assigned based on availability and area of expertise when the service request is first saved to the database
- A technician assignment is validated to ensure that the technician has the appropriate expertise to work on the current request
- Only managers are allowed to delete a service request

Regardless of which view object the end user's action affects, if it relates to data from the `SERVICE_REQUESTS` table, then your central `ServiceRequest` entity object handles the validation and saving of those changes. The event handling code that you might write in this entity object's custom Java class get triggered at the appropriate time. These event handlers will look something like this:

Example 1–4 Handing the create event in an entity object

```
protected void create(AttributeList attrs) {
  // First perform the default "built-in" functionality
  super.create(attrs);
  // Add custom creation-time defaulting logic here...
  // Default the request date to the current date
  setRequestDate(currentTimeRoundedToNearestTenMinutes());
}
```

Example 1–5 Handling the DML event for INSERT in an entity object

```
protected void prepareForDML(int operation, TransactionEvent event) {
  // First perform the default "built-in" functionality
  super.prepareForDML(operation, event);
  // If we're doing an INSERT, then default the technican id
  if (operation == DML_INSERT) {
    // Auto-assign the new service request
    setAssignedTo(determineDefaultTechnicianId());
  }
}
```

Example 1–6 Handling an attribute validation event for AssignedTo

```
  public boolean validateAssignedTo(Number newTechnicianId) {
    // Add custom validation code here for AssignedTo attribute
    // return true if valid, and false if not.
    return doesTechnicianHaveAppropriateExpertise(newTechnicianId);
  }
```

Example 1–7 Handling the remove event for an entity object

```
public void remove() {
  // Add custom remove-time logic here...
  if (isUserAllowedToDeleteRequest() == false) {
    throw new JboException("You aren't allowed to remove a service request.");
  }
  super.remove();
}
```

Tip: These four event handler methods are similar to the data-related triggers in Oracle Forms named `WHEN-CREATE-RECORD`, `PRE-INSERT`, `WHEN-VALIDATE-ITEM`, and `WHEN-REMOVE-RECORD` respectively.

To ensure modular and encapsulated code, the helper functions called by these sample event handlers like `currentTimeRoundedToNearestTenMinutes()`, `determineDefaultTechnicianId()`, `doesTechnicianHaveAppropriateExpertise()`, and `isUserAllowedToDeleteRequest()` are written as private methods in the same entity object class.

1.3.4.3 Simple to Globally Extend Basic Framework Functionality

Since all of Oracle ADF is implemented itself in Java, it's not only possible but quite straightforward to extend the basic functionality of the framework or globally change the default behavior to be more like what your organization needs. In other words, rather than waiting for Oracle to implement your favorite enhancement request, you can just implement it yourself in a class that extends the appropriate base ADF class.

Consider the example above of defaulting a `Date` attribute to the current time rounded to the nearest ten minutes. The `create()` event handler method is what you would write when you encounter the first entity object that requires this functionality during your development. However, if you discover over time that your application is requiring this facility in many different entity objects, you can choose to *globally* add a new feature to the entity object base class. Example 1–8 shows a custom `OurCompanyEntityImpl` class that extends the base ADF entity object class (`EntityImpl`) and overrides the same `create()` event handling method, but this time in a *global* way.

After calling the `super.create()` method to perform the default functionality, the code performs the following steps:

1. Loop over all attribute definitions for this entity row.
2. If the type of an attribute is `Date` and a custom property named `TenMinuteDate` has been set on it by the developer, default the attribute value to the current time rounded to nearest 10 minutes.

Example 1–8 Extending the Oracle ADF Entity Object with a New Feature of Your Own

```
package com.yourcompany.fwkext;
public class OurCompanyEntityImpl extends EntityImpl {
  protected void create(AttributeList attrs) {
    super.create(attrs); // First perform the default "built-in" functionality
    // 1. Then loop over all attribute definitions for this entity row
    for (AttributeDef attr : getEntityDef().getAttributeDefs()) {
      // 2. If attr is a Date and "TenMinuteDate" custom property is set
      if (attr.getJavaType().equals(Date.class)
          && attr.getProperty("TenMinuteDate") != null) {
        // 3. Default attr value to current time rounded to nearest 10 min
        setAttribute(attr.getIndex(),currentTimeRoundedToNearestTenMinutes());
      }
    }
  }
}
```

This is an example in the business service layer, but you can perform similar kinds of global framework customizations in virtually any layer of the Oracle ADF architecture. In practice, most Oracle ADF customers create themselves a layer of classes that extend each of the base ADF classes, then they configure Oracle JDeveloper to use their customized framework base classes instead of the ADF default class names. By doing this once, it becomes a "set it and forget it" policy that JDeveloper enforces for you as you use all of the Oracle ADF editors. Even if you initially have no need to extend the framework, just setting up the layer of framework extension classes puts you in the position to add code to those classes at any time — to work around a bug you encounter, or to implement a new or augmented framework feature — without having to revisit all of your existing applications.

Note: Full source for Oracle ADF is available to supported customers through Oracle Worldwide Support. The full source code for the framework can be an important tool to assisting you in diagnosing problems and in correctly extending the base framework functionality for your needs.

1.4 Highlights of Additional ADF Features

ADF has additional functionality that can greatly improve your development productivity. These features include:

- Section 1.4.1, "Comprehensive JDeveloper Design-Time Support"
- Section 1.4.2, "Sophisticated AJAX-Style Web Pages Without Coding"
- Section 1.4.3, "Centralized, Metadata-Driven Functionality"
- Section 1.4.4, "Generation of Complete Web Tier Using Oracle JHeadstart"

1.4.1 Comprehensive JDeveloper Design-Time Support

The Studio Edition of JDeveloper includes all of the following facilities that simplify development of enterprise solutions using Oracle ADF and JavaServer Faces:

Facilities for Business Services Development

- Business Components wizards and editors

  Quickly create and modify the components that comprise your business services using productive wizards and editors. Reverse-engineer components from existing tables. Synchronize your components with changes that you (or the DBA) have made to the underlying database schema.

- Business Components Browser

  Interactively test your business service's data model even before you build a user interface. The Business Components Browser can also help isolate where problems in your application occur, as it runs your data model without running your user interface.

- Business Components Diagrammer

  Visualize, create, or modify your business service and business domain layer components using UML diagrams. Publish the diagrams to numerous formats for reference or inclusion in system documentation.

Facilities for Declarative Data Binding

- Data control wizards

  ADF application modules are automatically exposed as data controls. Should your needs call for working with web services, XML or CSV data from a URL, JavaBeans, or EJB session beans, handy data control wizards guide you step by step.

- Data Control Palette

  Visualize all application modules and other business services and drag their data collections, 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 on 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 that you can modify 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. Edit binding metadata with context-sensitive, XML schema-driven assistance on the structure and valid values.

- Service method invocation

  Configure business service method invocations with EL expression-based parameter passing. You can have methods invoked by the click 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

- 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.

Facilities for Visual Web Page Design

- Visual Web Page Designer

  Design your web pages visually using the visual web page designer. The designer is integrated with the Data Control Palette to support both drag and drop creation of user interfaces, and UI-first page design with subsequent data binding applied to an initial page mockup. The visual editor supports both JavaServer Faces and traditional JSP development.

- Page Flow Diagrammer

  Design your web page navigation visually using the visual page flow diagram. The diagrammer supports both JavaServer Faces as well as Apache Struts.

Facilities for Visual Design for Desktop-Fidelity UI's with Swing

- Visual Form Designer

  JDeveloper fully supports developing desktop-fidelity user interfaces using Forms and Panels that use the standard Swing controls provided as part of Java itself. All of the Oracle ADF declarative data binding and ADF Business Components facilities work to make building either client/server or three-tier Swing applications easy.

> **Note:** This edition of the developer's guide focuses exclusively on Web development with JSF. The development of desktop-fidelity user interfaces using Oracle ADF and ADF Swing will be covered in a separate, follow-on developer's guide dedicated to that subject.

1.4.2 Sophisticated AJAX-Style Web Pages Without Coding

The JSF reference implementation provides a bare-bones set of basic UI components which 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:

- Declarative partial page refreshing for interactive UIs

  For any UI component in your pages, you can indicate declaratively which other components should trigger its being "repainted" with fresh data without causing the entire browser page to refresh. This type of more interactive web experience is known popularly as "Web 2.0" or "AJAX"-style pages.

- 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. 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.

- 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.

1.4.3 Centralized, Metadata-Driven Functionality

Oracle ADF improves the reuse of several aspects of application functionality by allowing you to associate layered metadata with either your ADF Business Components or, for other data control types, the data control structure definitions that describe the data collections. These can then be reused by any page presenting their information. Examples of this functionality are:

- Easily localizable prompts, tooltips, and format masks

  JSF supports a simple mechanism to reference translatable strings in resource bundles, but it has no knowledge of what the strings are used for and no way to associate the strings with specific business domain objects. ADF Business Components improves on this by allowing your JSF pages to reference translatable prompts, tooltips, and format masks that you can associate with any attribute of any entity object or view object component. In this way, 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. These offer the ability to perform basic syntactic checks on the data value. However, JSF offers no mechanism to simplify enterprise, database-centric validation nor to easily validate the same business domain data in a consistent way on every screen where it's used. ADF Business Components improves on this by allowing you to associate an extensible set of validator objects with your entity objects, and supplement that with validation code you write in event handlers as shown in Section 1.3.4, "Simple, Event-Driven Approach to Add Custom Logic". In this way, the validations are 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 Business Components, you can associate user or role authorization information with each attribute in an entity object so that your JSF pages can easily display data only to users authorized to see it.

1.4.4 Generation of Complete Web Tier Using Oracle JHeadstart

As you'll learn throughout the rest of this guide, Oracle JDeveloper 10*g* and Oracle ADF give you a productive, visual environment for building richly functional, database-centric J2EE applications with a maximally declarative development experience. However, if you are used to working with tools like Oracle Designer that offer *complete* user interface generation based on a higher-level application structure definition, you may be looking for a similar facility for your J2EE development. If so, then the Oracle JHeadstart 10*g* application generator may be of interest to you. It is an additional extension for JDeveloper that stands on the shoulders of Oracle ADF's built-in features to offer complete web-tier generation for your application modules. Starting with the data model you've designed for your ADF business service, you use the integrated editors JHeadstart adds to the JDeveloper environment to iteratively refine a higher-level application structure definition. This controls the functionality and organization of the view objects' information in your generated web user interface. By checking boxes and choosing various options from dropdown lists, you describe a logical hierarchy of pages that can include multiple styles of search regions, List of Values (LOVs) with validation, shuttle controls, nested tables, and other features. These declarative choices use terminology familiar to Oracle Forms and Designer users, further simplifying web development. Based on the application structure definition, you generate a complete web application that automatically

implements the best practices described in this guide, easily leveraging the most sophisticated features that Oracle ADF and JSF have to offer.

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

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

Once you've generated a maximal amount of your application's web user interface, you can spend your time using JDeveloper's productive environment to tailor the results or to concentrate your effort on additional showcase pages that need special attention. Once you've modified a generated page, you can adjust a setting to avoid regenerating that page on subsequent runs of the application generator. Of course, since both the generated pages and your custom designed ones leverage the same ADF Faces UI components, all of your pages automatically inherit a consistent look and feel. For more information on how to get a fully-functional trial of JHeadstart for evaluation, including details on pricing, support, and additional services, see `http://otn.oracle.com/consulting/9iServices/JHeadstart.html`.

2 Overview of Development Process with Oracle ADF and JSF

This chapter highlights the typical development process for using Oracle JDeveloper 10g Release 3 (10.1.3) to build web applications using Oracle ADF and JSF, using Oracle ADF Business Components to implement the business service layer.

This chapter includes the following sections:

- Section 2.1, "Introduction to the Development Process"
- Section 2.2, "Creating an Application Workspace to Hold Your Files"
- Section 2.3, "Thinking About the Use Case and Page Flow"
- Section 2.4, "Designing the Database Schema"
- Section 2.5, "Creating a Layer of Business Domain Objects for Tables"
- Section 2.6, "Building the Business Service to Handle the Use Case"
- Section 2.7, "Dragging and Dropping Data to Create a New JSF Page"
- Section 2.8, "Examining the Binding Metadata Files Involved"
- Section 2.9, "Understanding How Components Reference Bindings via EL"
- Section 2.10, "Configuring Binding Properties If Needed"
- Section 2.11, "Understanding How Bindings Are Created at Runtime"
- Section 2.12, "Making the Display More Data-Driven"
- Section 2.13, "Adding the Edit Page and Finishing the Use Case"
- Section 2.14, "Considering How Much Code Was Involved"

2.1 Introduction to the Development Process

The simplest way to appreciate how easy J2EE development can be using JDeveloper and Oracle ADF is to walk through a typical, end-to-end development process. Using Oracle ADF, enterprise J2EE application development can be a visual, code free experience. Of course, you eventually write code in a real-world application, but importantly Oracle ADF allows you to focus your efforts on the code that is directly related to your application's business requirements.

In the Service Request tracking system used as the real-world example application throughout the remainder of this guide, external users log service 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 walkthrough highlights the typical steps involved in implementing a solution for one small use case related to that overall system's functionality. In the walkthrough you'll create a small system to managing the profiles of technician users and their areas of technical expertise.

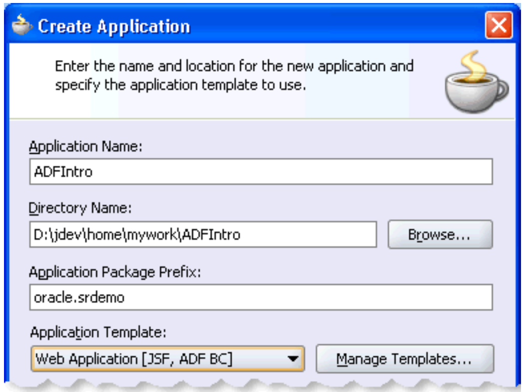
While this overview tries to give you the *essence* of the typical development process with Oracle ADF, it doesn't provide the click-by-click guidance of a tutorial. For a list of all the available Oracle ADF tutorials, go to the Oracle ADF product center on OTN at `http://otn.oracle.com/products/adf`.

Note: After reading the chapter, if you want to experiment with a *completed* version of the small application described here, download the `ADFIntro` workspace from the *Example Downloads* page at `http://otn.oracle.com/documentation/jdev/b25947_01/`.

2.2 Creating an Application Workspace to Hold Your Files

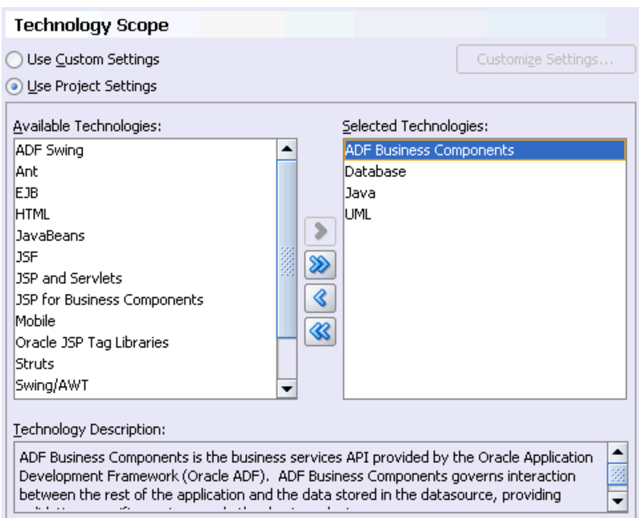
The first step in building a new application is to assign it a name and to specify the directory where its source files will be saved. Selecting **Application** from the JDeveloper **New Gallery** launches the Create Application dialog shown in Figure 2–1. Here you give the application a name like `ADFInfo`, set a working directory, and provide a package prefix for the classes you'll create in the application. You'll typically enter a package prefix like `oracle.srdemo` so that, by default, all of the components comprising the application will be created in packages whose names will begin with `oracle.srdemo.*`. Since you will be building a web application using JSF and ADF Business Components, Figure 2–1 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 2–1 Creating an Application Workspace to Hold Your Files



If you want to inspect the default technologies included in each project, you can open the Technology Scope page in the Project Properties dialog. Figure 2–2 shows what this page looks like after you add additional Database and UML technologies to the list in the `Model` project. By specifying exactly which technologies you intend to use in each project, you help JDeveloper simplify the available choices presented to the developer in the New Gallery and other dialogs to only those relevant to the current project's technologies.

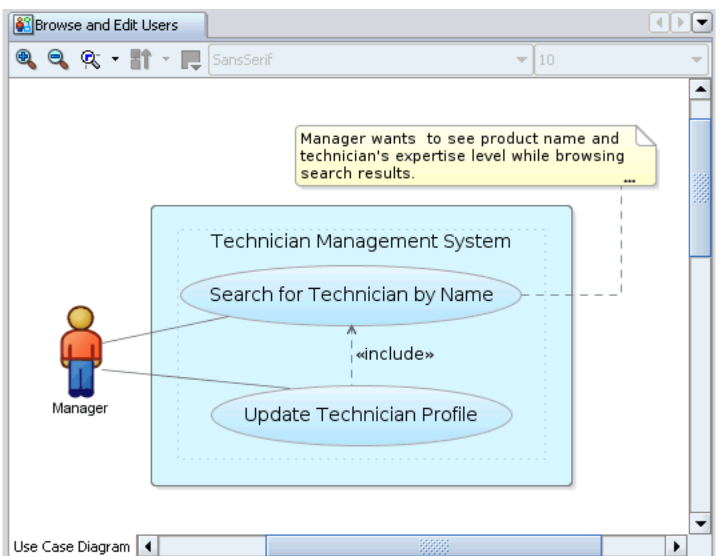
Figure 2–2 The Technology Scope Defines the Technologies Used in That Project



2.3 Thinking About the Use Case and Page Flow

After creating an application workspace, you might begin the development process by doing some use case modeling to capture and communicate end-user requirements for the system to be built. Figure 2–3 shows a simple diagram created the use case diagrammer, one of a number of built-in UML diagramming tools. The diagram represents the simple technician management system that your managers are asking you to build. It consists of two related use cases: "Search for Technician by Name" and "Update Technician Profile." Using diagram annotations, you can capture particular requirements about what end users might need to see on the screens that will implement the use case. For example, you might note that managers want to see product names and technician expertise levels while browsing the search results.

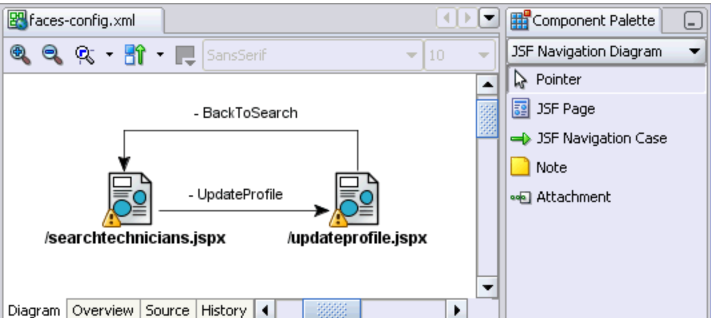
Figure 2–3 Use Case Diagram for a Simple Technician Management System



By modeling the use cases, you begin to understand the kinds of user interface pages that will be required to implement end-user requirements. As shown in Figure 2–4, using the JSF page flow diagrammer, you can create a skeleton page flow for the system. Since the page flow is related to the user interface, this work happens in the context of the `ViewController` project in the workspace.

Using the Component Palette, you drop pages and named navigation rules to connect them. You should be able to implement the requirements using the combination of a `searchtechnicians` page and an `updateprofile` page. After using the `searchtechnicians` page to find the technician to update, the manager will proceed to `updateprofile` page to modify that technician's profile. After saving the changes, she'll return back to the `searchtechnicians` page. The navigation lines on the page diagram reflect this flow. The warning symbols you see are no reason for alarm. They indicate that you still have to create the page that the page icon represents. You'll see that in a later step of the walkthrough.

Figure 2–4 Skeleton JSF Page Flow Diagram in the ViewController Project



Note: JDeveloper supports the ability to do "UI first" development to mock up web page displays using unbound UI components, and then bind them to data later. However, in this walkthrough you'll be following a more traditional bottom up approach that is most similar to the way traditional 4GL tools attack the problem. Both styles of development are possible, or you can develop simultaneously and meet in the middle.

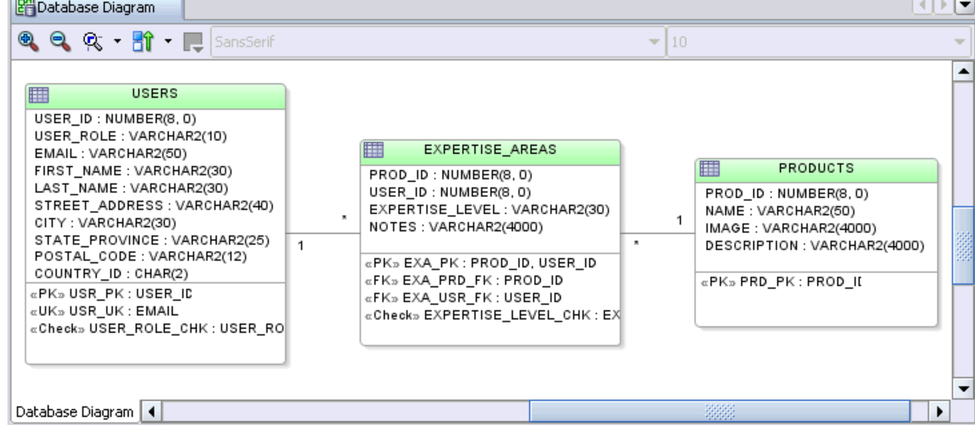
2.4 Designing the Database Schema

If you already have the database tables for `USERS`, `PRODUCTS`, and `EXPERTISE_AREAS` that store information about users with a technician role and the expertise they have for particular products, you can begin working with them immediately. Seeing them in a data diagram can help you understand whether the schema might require any changes to support the new use case.

After creating a named connection in the Connection Navigator to browse your database schema objects, you can see your existing tables under a Tables folder. Using the New Gallery, you can create a new database diagram, select the `USERS`, `PRODUCTS`, and `EXPERTISE_AREAS` tables from the Connection Navigator, and drop them onto the diagram. Figure 2–5 shows what the database diagram would look like.

Of course, if the tables didn't already exist, you could use the same diagrammer to design them and generate the DDL scripts to create the tables from scratch.

Figure 2–5 Database Diagram Showing Schema for Technician Management System



2.5 Creating a Layer of Business Domain Objects for Tables

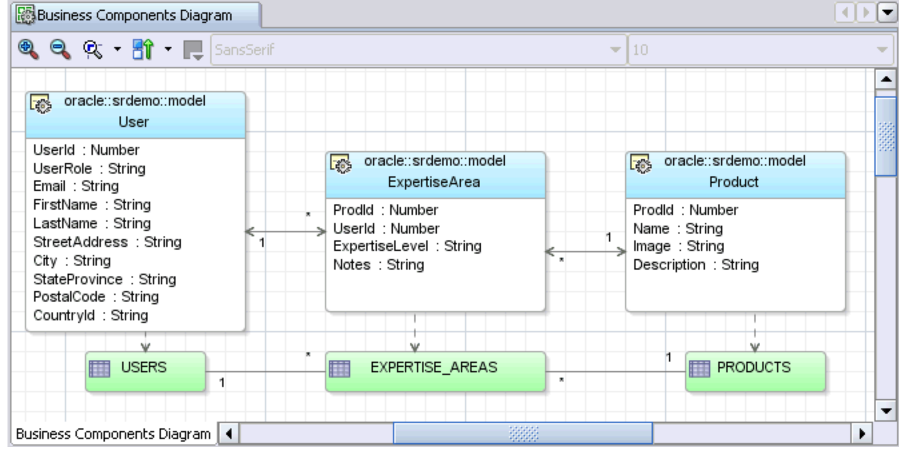
With the database tables in place, you can create a set of Java components that represents them and simplifies modifying the data they contain. Using entity objects to encapsulate data access and validation related to the tables, any pages you build today or in the future that work with these tables get consistently validated. Since this task is related to the data access and business logic for your application, you'll do it in the context of the `Model` project. As you work, JDeveloper automatically configures your project to reference any necessary Oracle ADF libraries your application will need at runtime.

2.5.1 Dragging and Dropping to Reverse-Engineer Entity Objects for Tables

You can create business components by using wizards or a diagram. On one hand, the wizards can be faster to use since you can create many components at the same time. For example, in a single step you can create entity objects, related view objects, and an application module to use as your business service with the Business Components from Tables wizard. On the other hand, creating business components on a diagram is often more intuitive and allows you to understand the roles each different kind of business component plays. Since creating business components using the wizards is covered in later chapters, for this walkthrough it is best to take things one steps at a time and work in a visual way.

To create a business components diagram, open the **New Gallery** and select **Business Components Diagram**. After creating a new diagram, just drop the tables from the Connection Navigator onto the diagram surface to create entity objects for them. By default, the entity objects get named `Users`, `ExpertiseAreas`, and `Products` based on the names of the tables. Since an entity object represents a single element of business domain data, it's best practice to name them with singular names. You can use the diagram's in-place editing support to rename the entity objects to `User`, `ExpertiseArea`, and `Product`. JDeveloper has extensive support for refactoring to automate changing any related files to use the new names. In addition to the entity objects that were created, the tool also created associations that reflect the foreign keys relating the underlying tables. To help communicate how your entity objects are related to the tables, you can add the tables themselves to the same diagram and draw dependency lines between them, as shown in Figure 2–6.

Figure 2–6 Business Components Diagram Showing Entity Objects and Related Tables



2.5.2 Adding Business Validation Rules to Your Entity Object

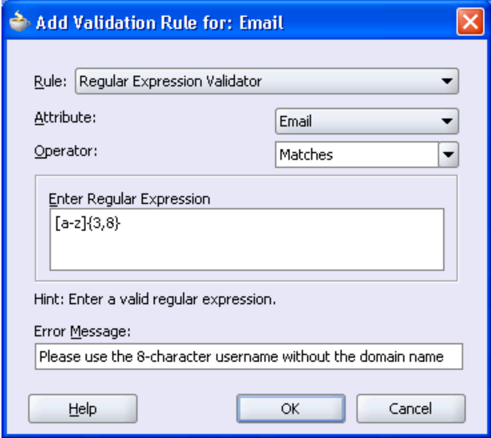
Your set of associated entity objects represents a reusable layer of business domain components. Any business validation rules you enforce at this level are enforced consistently throughout any applications you build that satisfy use cases related to `Users`, `ExpertiseAreas`, and `Products`. This applies regardless of the user interface technology that you use. This guide focuses attention on developing web applications with JSF, but validation rules encapsulated in your business domain layer work the same whether used through web, wireless, or Swing user interfaces, as well as through web services. The validation rules can range from the simplest syntactic value-checking to the most complicated enterprise database-backed programmatic rules.

By double-clicking the `Users` entity object in the diagram, you can access the Entity Object Editor to define declarative validation rules for users. Open the Validation page of the editor to add validation rules. Click **New** to add a new rule, and add a **UniqueKey Validator** that will ensure that the primary key for new `User` rows is unique. As shown in Figure 2–7, you can enter a validation error message that the manager will see if she enters a new user whose ID is already taken. JDeveloper saves the error message in a standard Java message bundle to simplify localizing it for different languages.

Figure 2–7 Adding a New UniqueKey Validator

Next, you could add a regular expression validator on the `Email` attribute to ensure that managers only enter email address in the expected 8-character, lowercase format. Selecting the `Email` attribute in the **Declarared Validation Rules** tree and clicking **New** again, you can add a Regular Expression Validator. Figure 2–8 shows the regular expression to match from three to eight lowercase characters, and the error message that will appear if the manager enters an email in the wrong format.

Figure 2–8 The Regular Expression Validator Ensures That Email Address Is Correctly Formatted



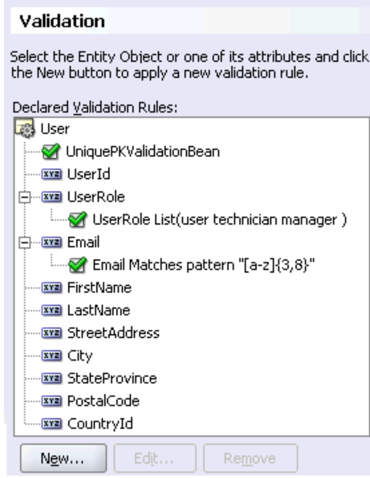
Select the `UserRole` attribute and click **New** to add a **List Validator** as shown in Figure 2–9 to ensure that the role of a new user can only be `user`, `technician`, or `manager`. Again, you can enter the error message the manager will see if she enters the data incorrectly.

Figure 2–9 The List Validator Ensures That UserRole Has One of Three Legal Values

Add Validation Rule for: UserRole
Rule: List Validator
Attribute: UserRole
Operator: In
List: Literal Values
Enter List of Values
user
technician
manager
Hint: Enter each value without quotes on a new line.
Error Message:
Role must be user, technician, or manager
Help OK Cancel

After doing this, you can see, as shown in Figure 2–10, that all the validation rules in effect are summarized on the Validation page of the Entity Object Editor.

Figure 2–10 The Validation Page Provides One Place to See All Validation Rules in Effect



In a real-world application, you would go on to add a number of method validators that invoke custom validation event handlers that you write in Java. Like the built-in validations, these can be attribute-level rules if they are related just to one specific attribute, or entity-level validation rules if they are more complex and involve consulting multiple attribute values or other entities to determine validity. The code you would write inside the validation events can easily access the properties of any entity object in your business domain layer, or easily perform any SQL query, as part of determining if the method-based validation rule should succeed or fail the validation check.

If you notice a pattern in the kind of custom Java-based validations your team is performing, you can extend the set of built-in validation rules in JDeveloper with custom, parameter-driven rules of your own. Custom rules allow other developers on your team to reuse the common validation patterns in their future work without writing code themselves. Instead they pick your custom validation rule from a list and set a few properties. Either way, whether business rules are implemented declaratively or using validation event handlers written in Java, they all appear in the Validation page of the Entity Object Editor, so that you have a single place to see all business rules in effect for that object.

2.5.3 Defining UI Control Hints for Your Entity Objects

In addition to business rules, other useful resources you can centralize in your entity object layer are the standard UI labels and format masks that your business information will use throughout all applications you build. By opening the Entity Object Editor and selecting individual attributes, you can use the **Control Hints** tab to define the **Label Text**, **Tooltip Text**, **Format Type**, **Format** mask and other hints. Since these user-visible labels and format masks need to be sensitive to the preferred language of the end user, JDeveloper manages them for you in a standard Java resource bundle. This way, they are straightforward to translate into other locales for creating multilingual applications.

Assume you've edited the `User`, `ExpertiseArea`, and `Product` entities to define label text hints for all the attributes. You'll see later in the walkthrough how these centralized UI label hints are automatically leveraged in the pages you build that work with data related to these entity objects.

2.6 Building the Business Service to Handle the Use Case

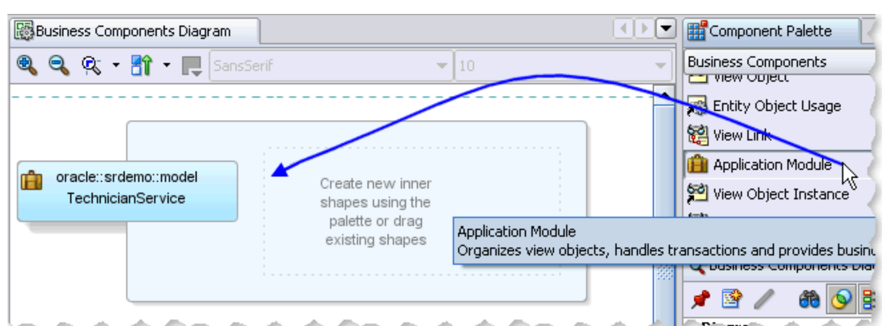
Once the reusable layer of entity objects is created, you can implement the specific business service to satisfy the needs of the use case at hand. Since this task is related to the data access and business logic for your application, you'll do it in the context of the `Model` project.

2.6.1 Creating a Application Module to Manage Technicians

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

After adjusting the diagram's visual properties to turn off the grid display, you can use the Component Palette to drop a new application module onto the business component diagram, as shown in Figure 2–11. Since it will be a service concerned with technicians, you could call it `TechnicianService` to reflect this role.

Figure 2–11 Creating a New Application Module on the Business Components Diagram



The application module's data model is composed of SQL queries that define what data the end user will see on the screen. To work with SQL query results, you'll define view object components that encapsulate the necessary queries. When the end user needs to update the data, your view objects reference entity objects in your reusable business domain layer. These entity objects ensure any data the end user modifies is validated and saved in a consistent way.

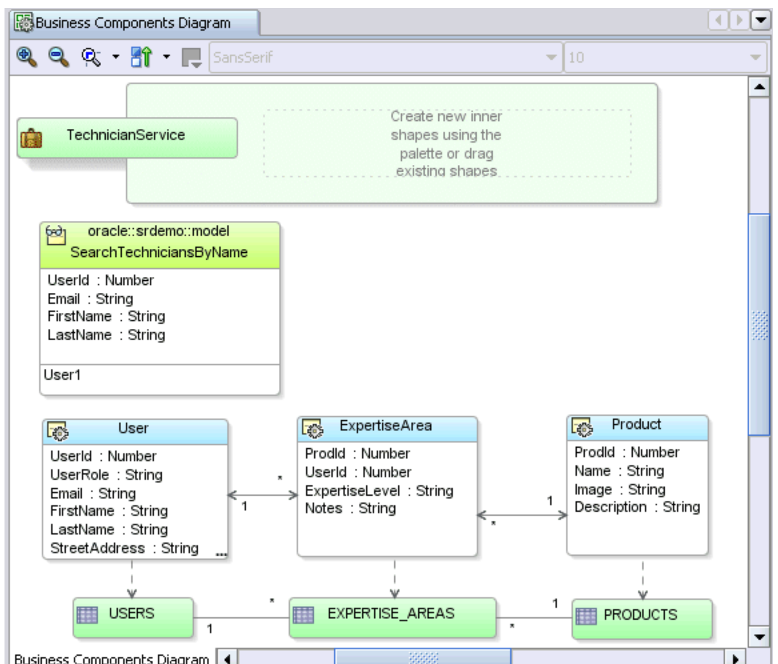
2.6.2 Creating View Objects to Query Appropriate Data for the Use Case

Start by considering the data required by the "Search for Technician by Name" use case. For certain, you'll need a query that involves `User` information, since technicians are a kind of user whose data resides in the `USERS` table. So, drop a new view object component onto the diagram and name it `SearchTechniciansByName`. Drag the `User` entity object and drop it onto the new `SearchTechniciansByName` view object component in the diagram. By creating a view object in this manner, JDeveloper updates the view object's SQL query to include all of that entity object's data.

For the search page, imagine that you only need to display the `UserId`, `Email`, `FirstName`, and `LastName` attributes. You can use the diagram to select the other attributes in the `SearchTechniciansByName` view object and remove them using the Delete key. By including only the data you need in the query, you make your application more efficient by retrieving only the information required by the task at hand.

To save space, you can display a view object in standard mode, and you can adjust the colors to distinguish different types of business components on the diagram. Figure 2–12 shows how a well organized business components diagram might look for the steps you've seen so far.

Figure 2–12 Business Components Diagram Including SearchTechniciansByName View Object



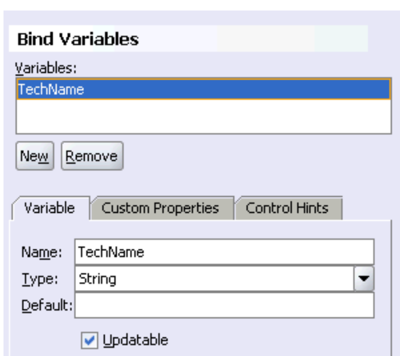
You want the `SearchTechniciansByName` view object to:

- Only find users with the '`technician`' role
- Be searchable by a user-supplied name parameter

To accomplish this you need to edit the view object to configure these features. Double-clicking the view object on the diagram to opens the View Object Editor.

To add a bind variable to support searching technicians by name, you can open the **Bind Variables** page of the editor as shown in Figure 2–13, and define one called `TechName` of type `String` with `null` as its default value. On the **Control Hints** tab you can define a **Label Text** hint for the bind variable of "Name Contains". This label will appear in any pages you build using this bind variable for searching.

Figure 2–13 Adding a Named Bind Variable to Search by Name



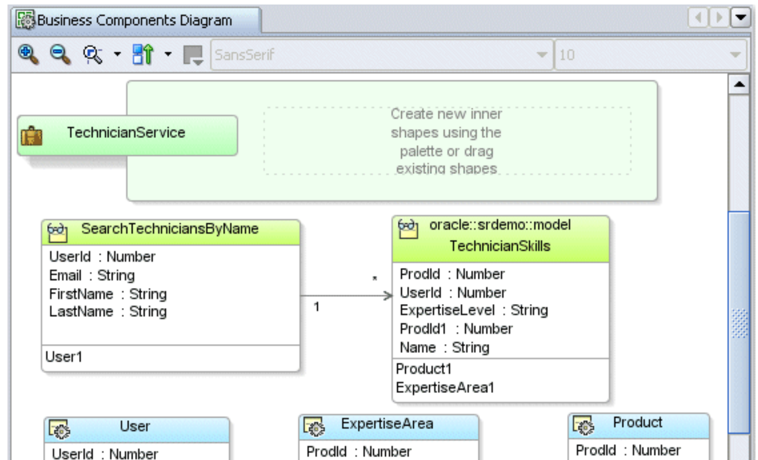
Next, you can open the SQL Statement page and fine-tune the `SELECT` statement. As shown in Figure 2–14, you can enter an appropriate WHERE clause to search only for technicians whose first name or last name matches the value of the `TechName` bind variable in a case-insensitive way. You can also enter an ORDER BY clause to sort the results first by last name, then by first name.

Figure 2–14 Adjusting the Query's Where and Order By Clauses

Recall from the use case diagram that the managers asked to see the related set of technician expertise information while browsing search results. They also indicated that they wanted to see the product name information related to the expertise level. You can drop a second view object onto the diagram and call it `TechnicianSkills`. Since it will need to include information from both the `ExpertiseArea` and `Product` entity objects, you can multiselect both of these entities and drop them together onto the new view object. Double-clicking the `TechnicianSkills` view object, you can see that the join query has been automatically determined, and you can use the **Attributes** page to remove the attributes you don't need.

Lastly, since you'll need a master/detail display between the results in the `SearchTechniciansByName` query and the `TechnicianSkills` query, you can use the **View Link** tool in the palette to connect the master view object to the detail view object. Figure 2–15 shows the results of having created the new detail `TechnicianSkills` view object and linking it master/detail with `SearchTechniciansByName`.

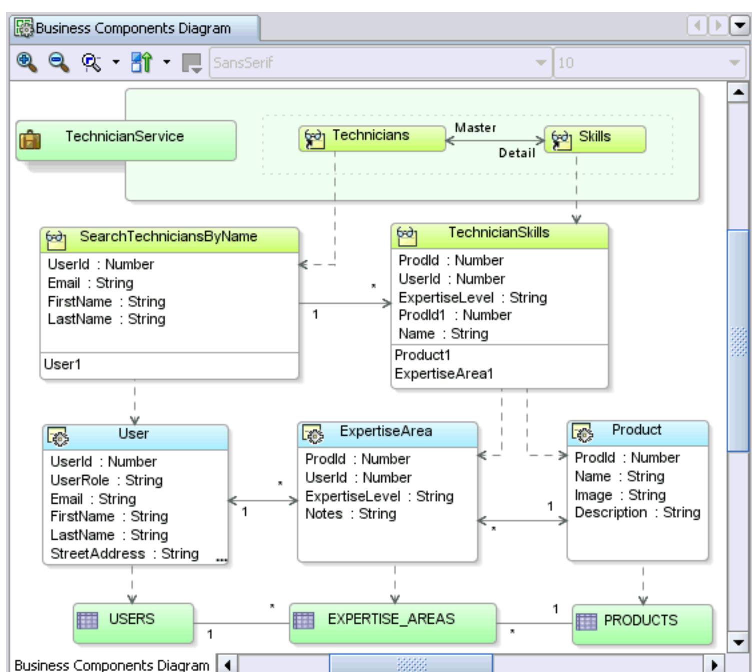
Figure 2–15 Master/Detail View Objects in the Business Components Diagram



2.6.3 Using View Objects in the Application Module's Data Model

The last step to complete the `TechnicianService` application module is to use your view objects to define its data model. Like entity objects, view objects are reusable components that you can use in multiple application modules when their queries make sense in multiple use cases. Selecting both the `SearchTechniciansByName` and `TechnicianSkills` view objects in the diagram and dropping them onto the application module accomplishes the task. You can rename the view object instances used in the data model to have shorter names. These names will be used by the client to identify the data collections produced by the view object's queries at runtime. In Figure 2–16, you can see what it would look like if you chose the shorter names `Technicians` and `Skills` to name the master/detail collections in the `TechnicianService`'s data model. The dotted lines are dependency lines you can add to further clarify how components on a UML diagram depend on each other.

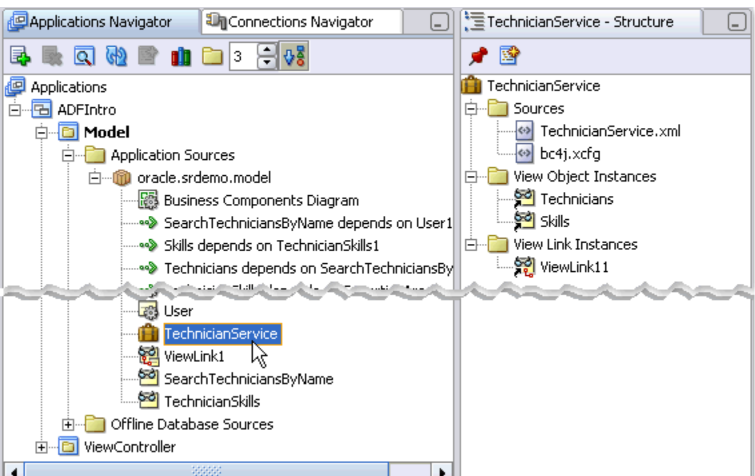
Figure 2–16 TechnicianService Application Module Containing Named Instances of Two View Objects



Whether you use wizards and editors or the visual diagrammers to work on your application components, as shown in Figure 2–17, the Application Navigator displays all of the components you've created. Selecting a particular component like the `TechnicianService` application module, you can see additional information about it in the Structure window.

You'll notice in the Structure window that the `TechnicianService` component has only an XML file in its `Sources` folder. Using ADF Business Components for your business service layer, each component has a related XML component definition file. If you don't have any *need* to write custom Java code to extend the behavior of a given component or to handle its events, you can just use the component in an XML-only mode. A class provided by the base framework gets used at runtime instead, and its behavior is determined by the metadata in its XML component definition file.

Figure 2–17 The Application Navigator Displays the Components Comprising Your Application

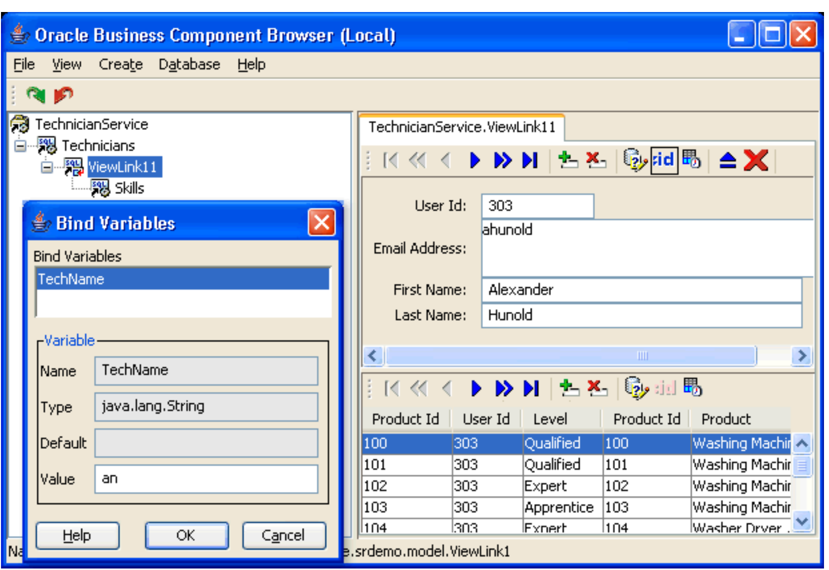


2.6.4 Testing Your Service

Even before building a user interface, you can interactively test your `TechnicianService` component using the integrated Business Components Browser. To launch this testing tool, select the application module in either the Application Navigator or the diagram and choose **Test...** from the context menu. When the Business Components Browser Connect dialog appears, click **Connect**.

As shown in Figure 2–18, the `Technicians` and `Skills` master/detail view object instances that comprise the data model of the `TechnicianService` component appear in the tree at the left. Double-click the `ViewLink11` node that links the two in the tree and a Bind Variables dialog appears to prompt you for a value for the `TechName` bind variable. Entering a value like "an" and clicking **OK** executes the `Technicians` view object query, and the tester tool shows you the results. Any technicians whose first name or last name contains the letters "an" in the name appear. Notice that the UI controls appear with the labels that you defined in the UI control hints for the entity objects in your business domain layer.

Figure 2–18 Interactively Testing the TechnicianService Application Module



Using the toolbar buttons you can scroll through the master data to observe the automatic master/detail coordination. If you try updating the email address of Alexander Hunold to `ahunold@srdemo.com` and click into a different field, you'll receive the validation exception shown in Figure 2–19. Recall that this was one of the business rules that you encapsulated inside your `User` entity object in the business domain layer. The exception verifies the automatic coordination that occurs between updatable rows in view objects and the underlying entity objects that they reference.

Figure 2–19 The Business Components Browser Showing Failed Entity Object Validation Rule

While doing development with Oracle ADF, you will find the Business Components Browser is very useful. It allows you to exercise all of the data model of your business service component without having to build — or without having to use — your end-user UI pages. This can save you a lot of time when trying to diagnose problems, or just test out the latest queries or business rules you've added to your business service layer.

2.6.5 The Data Control for Your Application Module Enables Data Binding

With the business service in place, it's time to think about creating the user interface. Application modules you build are automatically exposed as JSR-227-compliant data controls to enable drag-and-drop data binding using JDeveloper's rich design-time support for this specification. Figure 2–20 shows the Data Control Palette and the `TechnicianServicesDataControl` that is automatically kept in sync with your application module definition as you create and modify it. You can see the `Technicians` and `Skills` data collections that represent the view object instances in the `TechnicianService`'s data model. The `Skills` data collection appears as a child of `Technicians`, reflecting the master/detail relationship you set up while building the business service. The attributes available in each row of the respective data collections appear as child nodes. The data collection level **Operations** folders, shown collapsed in the figure, contains the built-in operations that the ADF Model layer supports on data collections like Previous, Next, First, Last, and so on.

Figure 2–20 The Data Control Palette Displays Business Services for Declarative Data Binding

---

Note: If you create other kinds of data controls for working with web services, XML data retrieved from a URL, JavaBeans, or EJBs, they would also appear in the *Data Control Palette* with an appropriate display. The first time you create one of these data controls in a project, JDeveloper creates a `DataControls.dcx` file that contains configuration information about them. In addition, it creates XML structure definition files for each data type involved in the service interface.

These additional files are not needed when you are working with application modules since application modules are already metadata-driven components whose XML component definition files contain all the information necessary to be exposed automatically as JSR 227 data controls.

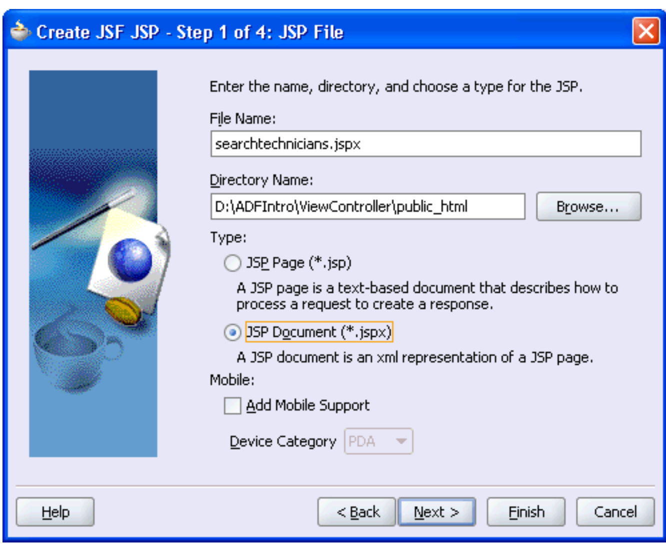
2.7 Dragging and Dropping Data to Create a New JSF Page

Now that you're familiar with the basics of the Data Control Palette, you can begin doing drag-and-drop data binding to create your page. Since you'll be using ADF Faces components in your page, 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, as shown in Figure 2–21. If the **ADF Faces Components** and **ADF Faces HTML** libraries are missing, you can add them from here.

Figure 2–21 Configuring ViewController Project Tag Libraries to Use ADF Faces

Next, you can select the `ViewController` project in the Application Navigator and choose **Open JSF Navigation** from the context menu to return to the skeleton JSF page flow you created earlier. Double-click the `/searchtechnicians.jspx` page icon in the diagram to launch the Create JSF JSP wizard shown in Figure 2–22, which you use to create the file representing the `searchtecnicians.jspx` web page.

Figure 2–22 Creating a New JSF JSP Page

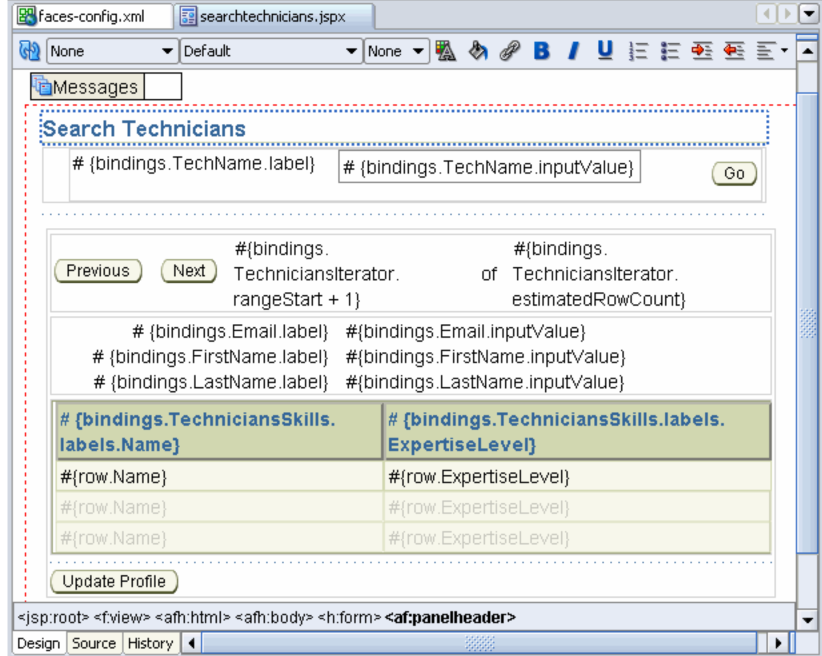


You may be more familiar working with JSP pages that have a `*.jsp` extension, but using a standard XML-based JSP "Document" instead is a best practice for JSF development since it:

- 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

You can click **Finish** on the first page of the wizard, taking all defaults. After completing the Create JSF JSP wizard, a new page opens in the visual editor. From there, creating the databound page shown in Figure 2–23 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 2–23 Browse Users JSF Page in the Visual Designer



The basic steps to create this page are:

1. Drop a `panelHeader` component from the ADF Faces Core page of the Component Palette onto the page and set its `text` attribute in the Property Inspector to "Search Technicians".

2. Drop the `ExecuteWithParams` built-in operation for the `Technicians` data collection in the Data Control Palette to create an ADF parameter form. This step creates a `panelForm` component containing the label, field, and button to collect the value of the TechName bind variable and pass it to the view object when the button is clicked. Use the Property Inspector to set the text on the command button created to "*Go*".

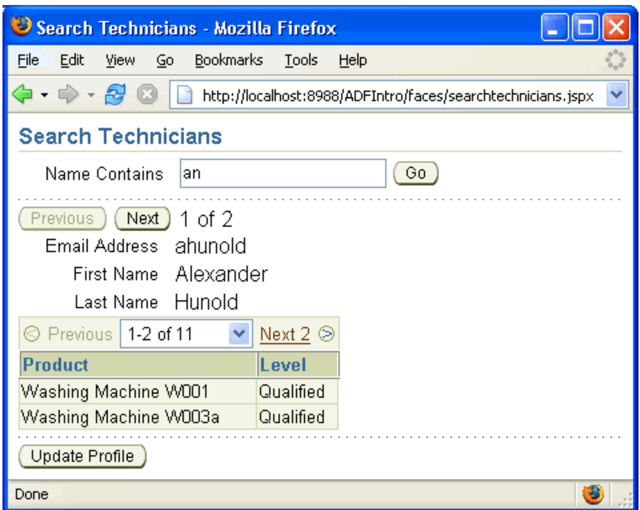
3. Drop the `Technicians` data collection from the Data Control Palette to create an ADF read-only form. This operation creates a `panelForm` component containing the label and fields for the attributes in a row of the `Technicians` data collection. In the Edit Form Fields dialog that appears, you can delete the `UserId` attribute from the list so that it doesn't appear on the page.

4. Again from the **Operations** folder of the `Technicians` data collection, drop the built-in `Previous` operation to the page as a command button. Repeat to drop a **Next** button to the right of it for the built-in `Next` operation.
5. Drop the detail `Skills` data collection in the Data Control Palette as an ADF read-only table. In the Edit Table Columns dialog that appears, include columns in the table only for the `ExpertiseLevel` and `Name` attributes, and use the **Up** and **Down** buttons to reorder them so that `Name` comes first. Select the **Enable sorting** checkbox to enable sorting the data by clicking the column headers.
6. Drop a `commandButton` from the **Component Palette** at the bottom of the page and change its `Text` property to "Update Profile". Set its `Action` property to "UpdateProfile" by picking it from the list to have the button follow the "UpdateProfile" navigation rule you created in the JSF page flow diagram earlier.

> **Note:** Producing the *exact* look of the page in Figure 2–23 involves a few additional drags and drops to group the default components, add an object separator between them, and dropping some read-only text fields bound to the current row count and total rows using EL. These steps are explained in a later section of this walkthrough.

At any time you can run or debug your page to try out the user interface that you've built. Notice that the UI control hints you set up on your entity object attributes in the business domain layer automatically appear in the user interface. Searching by name finds technicians whose last name or first name contains the string you've entered (case-insensitively). The **Previous** and **Next** buttons navigate among the technicians found by the search, and each technician's related set of skills display, along with the product name to which the expertise applies. A simple master/detail search page is illustrated in Figure 2–24.

Figure 2–24 Simple Search Technicians Page with Master/Detail Data

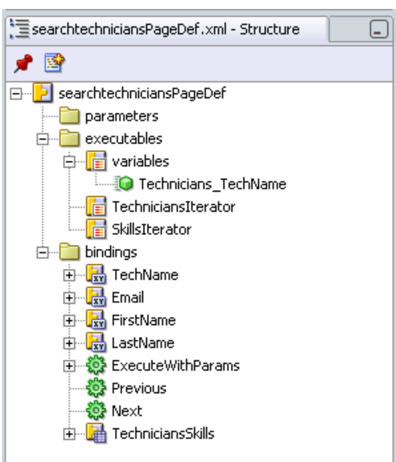


2.8 Examining the Binding Metadata Files Involved

The group of bindings supporting the UI components on a page are described in a page-specific XML file called the *page definition file*. The 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 2–25 shows the contents of the `searchtechniciansPageDef.xml` file in the Structure window. Each time you add components to the page using the Data Control Palette JDeveloper adds appropriate declarative binding entries into this page definition file.

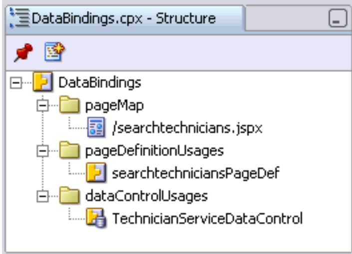
For example, after performing the steps in the previous section to create the databound UI components on the `searchtechnicians.jspx` page, you can see in the figure that JDeveloper added an action binding named `ExecuteWithParams` to invoke the built-in data control operation of the same name. Iterator bindings named `TechniciansIterator` and `SkillsIterator` were added to handle the data collection of rows from the `Technicians` and `Skills` view object instances, respectively. Action bindings named `Next` and `Previous` were added to support the buttons that were dropped. And, finally, value bindings of appropriate names were added to support the read-only `outputText` fields and the table.

Figure 2–25 Page Definition XML File for searchtechnicians.jspx



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 that lists the data controls that are in use in the project. Figure 2–26 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 2–26 Structure of DataBindings.cpx



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".

2.9 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 file and creates tags representing the JSF UI 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 2–27 summarizes how the page's UI components reference the bindings in its page definition. At runtime, these bindings are contained in a *binding container* related to the page.

Figure 2–27 EL Expressions Relate UI Components in a Page 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}"`

The first EL expression "wires" the button to execute the action binding named `Previous` in the binding container bound to the built-in `Previous` operation of the related data collection. The second EL expression automatically disables the button when the `Previous` operation does not make sense, such as when the user has navigated to the first row in the data collection.

Studying another example in the page, like the read-only `outputText` field that displays the user's email address and the `panelLabelAndMessage` component that contains it, you would see that JDeveloper sets up the following properties on these components to refer to the `Email` value binding:

- `value="#{bindings.Email.inputValue}"`

  on the outputText component

- `label="#{bindings.Email.label}"`

  on panelLabelAndMessage component

The combination of these settings "wires" the `outputText` component to pull its value from the `Email` binding, and the `panelLabelAndMessage` component to use the `Email` binding's `label` property as a display label. Since you configured UI controls hints for the attributes of your entity objects in the business domain layer, the control hints are inherited by the view object's attributes. The bindings expose this information at runtime so that components can easily refer to the control hints, such as labels and format masks using EL expressions.

The drag-and-drop data binding just completed did not account for how the current record display (for example "N of M") appeared on the page. To create it, you will need to reference properties for:

- The current range of visible rows
- The starting row in the range
- The total number of rows in the collection

Since the bindings expose properties for these, it is easy to create a current record display like this. Just drop three `outputText` components from the Component Palette and set each component's `value` attribute to an appropriate EL expression. The first one needs to show the current row number in the range of results from the `Technicians` data collection, so set its value attribute to an EL expression that references the (zero-based) `rangeStart` property on the `TechniciansIterator` binding.

```
#{bindings.TechniciansIterator.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, make a reference to the iterator binding's `estimatedRowCount` property:

```
#{bindings.TechniciansIterator.estimatedRowCount}
```

2.10 Configuring Binding Properties If Needed

Any time you want to see or set properties on bindings in the page definition, you can choose **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. As shown in Figure 2–27, after opening the page definition, the Property Inspector was used to set the `RangeSize` of the `TechniciansIterator` binding to `1` and the same property of the `SkillsIterator` to `2`. Setting the `RangeSize` property for each iterator this way causes one user and two expertise areas to display at a time on the page.

2.11 Understanding How Bindings Are Created at Runtime

To round out this basic introduction of ADF Model binding for JSF pages, it is good to understand how your data controls and declarative bindings are created at runtime. As part of configuring your project for working with Oracle ADF data binding, JDeveloper registers an additional handler that is triggered whenever a client requests a JSP page. This handler is listed in the standard J2EE web application configuration file (`web.xml`) of your `ViewController` project. It sets up the correct binding container for the current page based on the related ADF Model XML configuration files, and makes it accessible using the EL expression `#{bindings}`. During the subsequent handling of the web page request, the JSF standard dictates a predictable set of processing steps known as the page "lifecycle". Oracle ADF uses standard mechanisms to plug into these processing steps to automate preparing data for display, invoking built-in operations or custom service methods, as well as updating and validating data against the view objects in your application module's data model.

Note: If your curiosity already craves step-by-step details on how a page request is handled when using JSF and Oracle ADF, see the annotated sequence diagram in Section 29.2, "Lifecycle of a Web Page Request Using Oracle ADF and JSF"

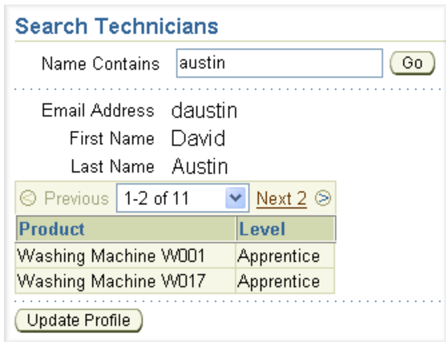
2.12 Making the Display More Data-Driven

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 reference the properties of ADF bindings to hide or show groups of components or to toggle between alternative sets of components.

2.12.1 Hiding and Showing Groups of Components Based on Binding Properties

If the manager enters a name in the `searchtechnicians.jspx` page that matches a single user, the disabled **Next** and **Previous** navigation buttons the "1 of 1" record counter are superfluous. Instead, you might want a result like what you see in Figure 2–28, where these components disappear when only a single row is returned.

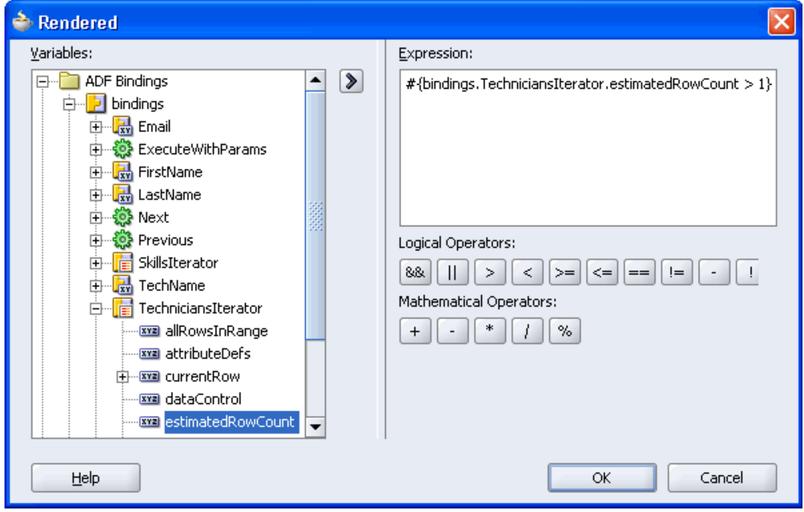
Figure 2–28 Hiding Panel with Navigation Buttons When Not Relevant



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 in the Structure window to place the existing controls inside the new container component. 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 data collection can be obtained using its `estimatedRowCount` property. Figure 2–29 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 **ADF Bindings** folder as well as the **bindings** node to see the bindings in the current page's binding container, you will see the `TechniciansIterator`. You can then expand this iterator binding 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.

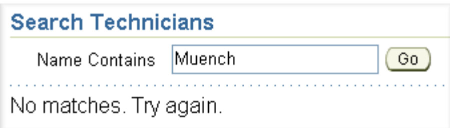
Figure 2–29 Setting a Panel's Rendered Attribute Based on Binding Properties



2.12.2 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 2–30.

Figure 2–30 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, you can use the switcher component to switch between showing that panel and a simple message, depending on the number of rows returned.

Figure 2–31 shows the Structure window for the `searchtechnicians.jspx` 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 technician and experience area information. In the `notfound` facet, you want just an `outputText` component that displays the "No matches. Try again" message. Finally, you set the `DefaultFacet` property on the switcher component to `found` so that facet displays by default.

Figure 2–31 Switcher Component Containing 'found' and 'notfound' Facets

Set the `facetName` attribute of switcher to the following EL expression so that 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.TechniciansIterator.estimatedRowCount > 0 ? 'found' : 'notfound'}
```

The combination of Oracle ADF declarative bindings, ADF Faces components, and EL expressions is another situation in which tedious, repetitive coding can be handled with ease.

2.13 Adding the Edit Page and Finishing the Use Case

With the page for the "Search Technicians by Name" use case implemented, you can turn your attention to the "Update Technician Profile" use case. Assuming you want the manager to be able to view *all* the attributes of the technician user when they go to update the profile, you'll need to create a view object that includes all the `User` entity object's attributes to support the `updateprofile.jspx` page. Once you add this new view object to the `TechnicianService`'s data model, you'll see it appear in the Data Control Palette, from where you can then drop it onto the `updateprofile.jspx` page to create an edit form in one step.

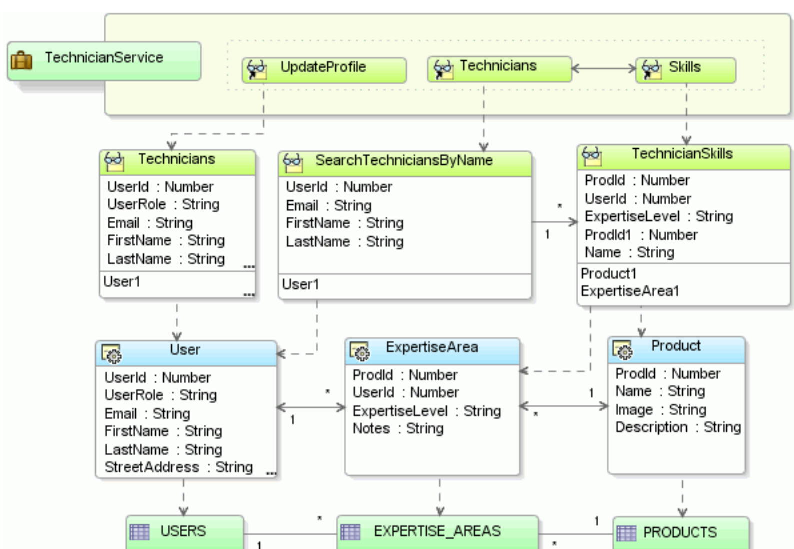
2.13.1 Adding Another View Object to the Data Model

To add a view object with all of the `User` entity object's attribute, open the business component diagram again in the `Model` project and drop a new view object from the Component Palette. Name the new view object `Technicians` since it will be used to show all the data about technicians. Drop the User entity object onto the new view object in the diagram, and double-click the view object to open the View Object Editor. Using the editor you can:

- Add the same WHERE clause you did for the `SearchTechniciansByName` view object to query rows only where `user_role = 'technician'`
- Set the `UserId` attribute to be **Updatable** only **While New** to prevent the end user from updating the values of `UserId` values in existing rows.

Back on the diagram drop the `Technicians` view object onto the `TechnicianService` to use it in the data model, and name the view object instance `UpdateProfile`. Figure 2–32 shows the updated business service model. You'll notice that the corresponding `UpdateProfile` data collection appears immediately in the *Data Control Palette*.

Figure 2–32 Updated Business Service Diagram



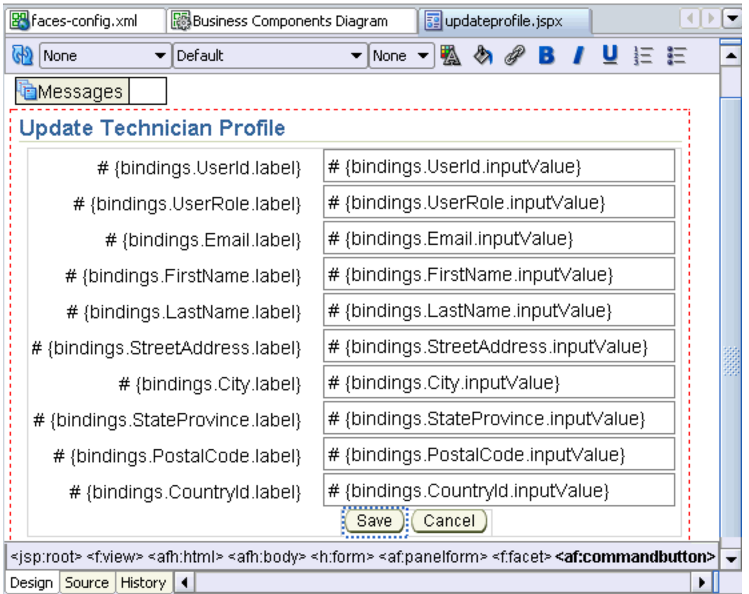
2.13.2 Creating the Edit Page

To open the visual editor for the `updateprofile.jspx` page, open the JSF page flow diagram, double-click the `/updateprofile.jspx` page and click **Finish** in the Create JSF JSP wizard.

You can build the edit form shown in Figure 2–33 with the following steps:

- Drop a `panelHeader` component from the **ADF Faces Core** page of the Component Palette onto the page and set its `text` attribute in the Property Inspector to "Update Technician Profile".
- Drop the `UpdateProfile` data collection from the Data Control Palette and drop it onto the page as an ADF Form.
- Expand the **Operations** folder of the **TechnicianService** data control in the Data Control Palette and drop its `Commit` built-in operation in the "footer" facet folder of the `panelForm` in the Structure window. Change its `Text` property to "Save". Set its `Disabled` property to "false". Set its `Action` property to "BackToSearch" by picking it from the list to have the button follow the "BackToSearch" navigation rule you created in the JSF page flow diagram earlier.
- Drop a `Rollback` built-in operation to the "footer" facet of the `panelForm` in the Structure window. Change its `Text` property to "Cancel". Set its `Disabled` property to "false". As with the **Save** button, set this button's `Action` property to "BackToSearch".

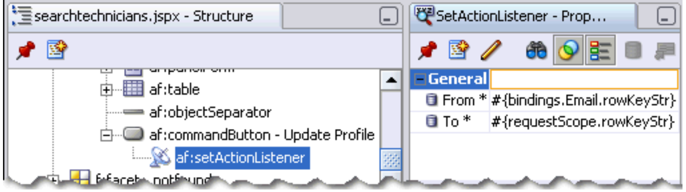
Figure 2–33 Update Technician Profile



2.13.3 Synchronizing the Search and Edit Page

When a manager finds the technician whose profile she wants to update, clicking the **UpdateProfile** button will take her to the `updateprofile.jspx` page. In order for the `updateprofile.jspx` page to know which technician to edit, you can use a declarative technique to set an attribute to the value of key that represents the current technician's row. Figure 2–34 shows the Structure window after a declarative `setActionListener` component is added as a child of the **Update Profile** command button. This `setActionListener` component evaluates one EL expression to determine the value to copy and another EL expression to determine the property to which to assign this value. The Property Inspector shows the EL expressions indicating that it should copy the value from the `rowKeyStr` property of the `Email` binding, and set it into the value of an attribute named rowKeyStr in the request scope.

Figure 2–34 Declaratively Setting a Request Attribute



In the page definition of the `updateprofile.jspx` page, you can use another declarative technique to invoke an action binding for the built-in `setCurrentRowWithKey` operation whenever the page is rendered. This action binding accepts a parameter whose value you can provide using the `#{requestScope.rowKeyStr}` expression. In this way, without writing code you can use the key of the current row from the browse page to set the current row for editing in the `UpdateProfile` data collection. These steps allow you to synchronize the two pages without writing code.

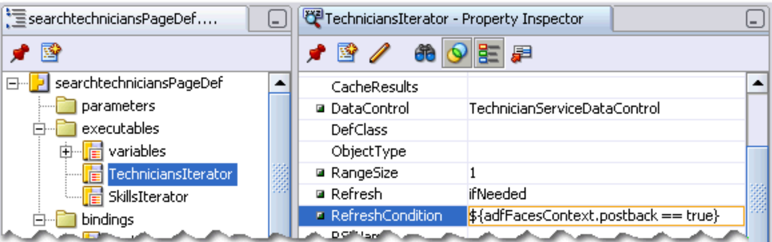
2.13.4 Controlling Whether Data Appears Initially

For a page like `searchtechnicians.jspx`, sometimes you want the end user to immediately see results from the view object's query when they see the page for the first time. On other occasions, you might want the user to first have a chance to enter some search criteria before performing the query. Assuming you want the `searchtechnicians.jspx` page to behave like this, you can conditionalize the initial display of data based on an EL expression. The expression that will come in handy for this tasks allows you to detect whether the page is being displayed for the first time, or whether the end-user has subsequently clicked a button or link on the page, causing the browser to post a request back to the server to handle the event. As shown in Figure 2–35, by setting the `RefreshCondition` property on the `TechniciansIterator` binding to the EL expression:

```
${adfFacesContext.postback == true}
```

you cause the iterator to present data only after the user has interacted with the page. When the page first renders, this postback property will be false, and the related data collection will be empty.

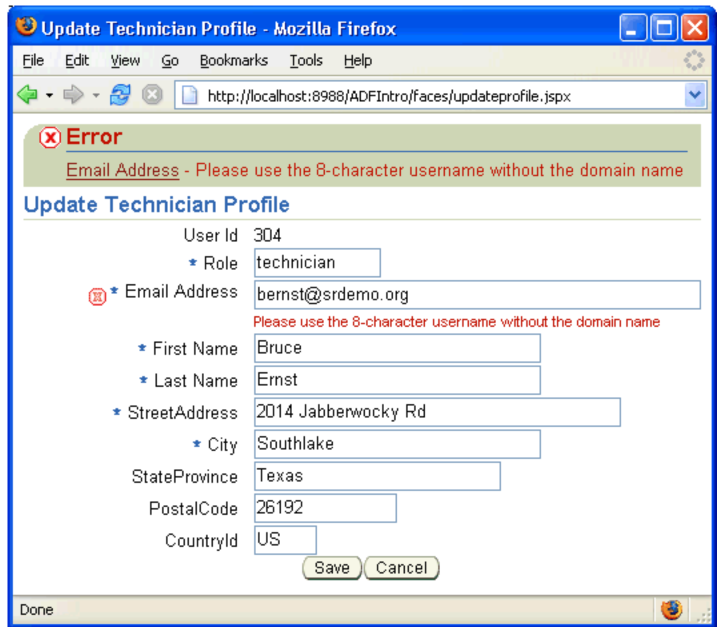
Figure 2–35 Declaratively Controlling When an Iterator Displays Data



2.13.5 Running the Final Result

If you run the pages now you will see that they are working. A manager can search for technicians, browse their skills, update a technician's profile, and save changes. If the manager accidentally updates a piece of data that violates a business rule that you've encapsulated inside your domain business layer of entity objects, she will automatically see the error messages. Figure 2–36 shows the results of trying to update the email address to a value that contains the domain name. Similar errors would appear if you tried to violate the other business rules you added to the `User` entity, or if you had additional view objects that referenced the same entity objects. When multiple managers use the application simultaneously, the ADF components comprising the business service automatically handle multi-user concurrency through row locking, lost-update protection, and application module pooling.

Figure 2–36 Automatic Presentation of Failed Entity Object Business Rules



2.14 Considering How Much Code Was Involved

In this walkthrough, you've seen the steps to build a simple, yet complete J2EE application with an MVC architecture using:

- Oracle ADF Model for declarative data binding for the model layer
- JavaServer Faces with Oracle ADF Faces components for the view layer
- JavaServer Faces page flow and declarative handlers for the controller layer, and
- Oracle ADF Business Components for the business service layer

The experience was both visual and declarative: you didn't have to write any Java code to achieve the results shown here. In fact, if you were to search the workspace directory for all `*.java` files created during the course of the walkthrough, you would see the results shown in Figure 2–37. Assuming you follow the recommended practices for ADF beginners of only generating Java code when you need it, you see that JDeveloper only created four `*.java` files. And these files don't even count as "code", since all they contain are the translatable strings related to your ADF business components.

Figure 2–37 Only Java Files Required for This Walkthrough Are Java Message Bundles

```
D:\ADFIntro>dir /s *.java
 Volume in drive D is Drive D
 Volume Serial Number is 087B-55EF

 Directory of D:\ADFIntro\Model\src\oracle\srdemo\model\common

04/07/2006  14:33               836 ExpertiseAreaImplMsgBundle.java
04/07/2006  14:33               825 ProductImplMsgBundle.java
04/08/2006  08:56               804 SearchTechniciansByNameRowImplMsgBundle.java
04/07/2006  14:33             1,268 UserImplMsgBundle.java
               4 File(s)          3,733 bytes

     Total Files Listed:
               4 File(s)          3,733 bytes
               0 Dir(s)  12,721,233,920 bytes free

D:\ADFIntro>
```

If you were to peek into one of the message bundle files like `UserImplMsgBundle.java`, Example 2–1 shows what you would see. It contains the UI control hints you defined for the attributes of the `User` entity object, along with the error message strings for the business rule validators. By automatically maintaining these standard Java message bundle files for you, JDeveloper simplifies creating multilingual applications. JavaServer Faces provides a similar mechanism to cleanly separate your translatable strings into separate resource files. At runtime, both JSF and ADF use the most appropriate version of the message resources based on the user's browser language settings.

Example 2–1 UserImplMsgBundle Translatable Control Hints and Errors Messages

```
public class UserImplMsgBundle extends JboResourceBundle {
  static final Object[][] sMessageStrings = {
    { "UserRole_LABEL", "Role" },
    { "PostalCode_TOOLTIP", "Zip" },
    { "UserRole_Rule_0",
       "Role must be user, technician, or manager" },
    { "Email_Rule_0",
      "Please use the 8-character username without the domain name" },
    { "User_Rule_0",
      "This user id is already taken. Please choose another." },
    { "CountryId_TOOLTIP", "Country" },
    { "City_TOOLTIP", "City" },
    { "LastName_LABEL", "Last Name" },
    { "StateProvince_TOOLTIP", "State" },
    { "FirstName_LABEL", "First Name" },
    { "Email_LABEL", "Email Address" },
    { "StreetAddress_TOOLTIP", "Address" },
    { "UserId_LABEL", "User Id" }
  };
// etc.
}
```

So discounting these message bundle files, you developed the entire walkthrough without writing or generating a line of Java code. This is a testament to the depth and breadth of declarative functionality that the combination of JSF and Oracle ADF can offer you, especially when using the ADF Business Components technology for your business service layer. 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, ADF Business Components, and JSF.

3

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 3.1, "Introduction to the Oracle ADF Service Request Demo"
- Section 3.2, "Setting Up the Oracle ADF Service Request Demo"
- Section 3.3, "Quick Tour of the Oracle ADF Service Request Demo"

3.1 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:

- Oracle ADF Business Components application module to encapsulate the business service interface and data model
- Oracle ADF Business Components entity objects representing 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, "Introduction to 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 4GL developers.

3.1.1 Requirements for Oracle ADF Service Request Application

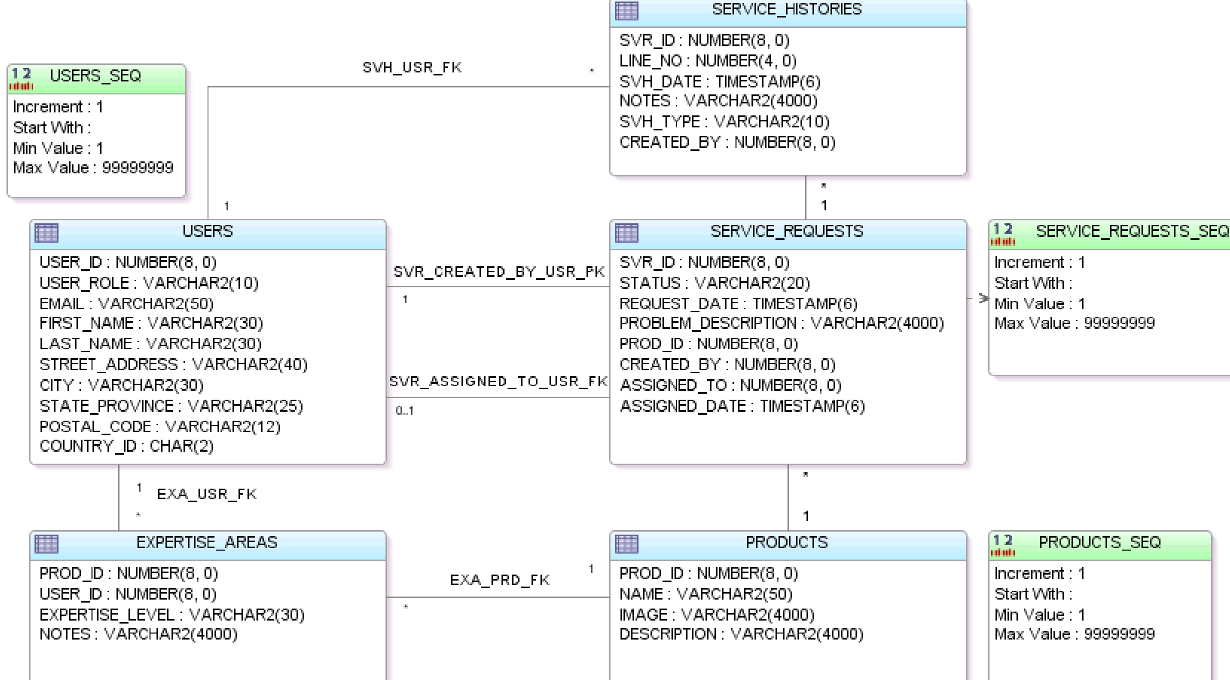
The SRDemo application has the following basic requirements:

- 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 3.2.3, "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 3.2.5, "Running the Oracle ADF Service Request Demo Unit Tests in JDeveloper").

3.1.2 Overview of the Schema

Figure 3–1 shows the schema for the SRDemo application.

Figure 3–1 Schema Diagram for the SRDemo Application



The schema consists of five tables, three database sequences, and one PL/SQL package. 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.

The PL/SQL package `CONTEXT_PKG` contains a procedure `SET_APP_USER_NAME()` and a function `APP_USER_NAME()`, used to illustrate a simple example of how to set per-user database state from inside an application module.

3.2 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.ht
ml
```

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.

3.2.1 Downloading and Installing the Oracle ADF Service Request Application

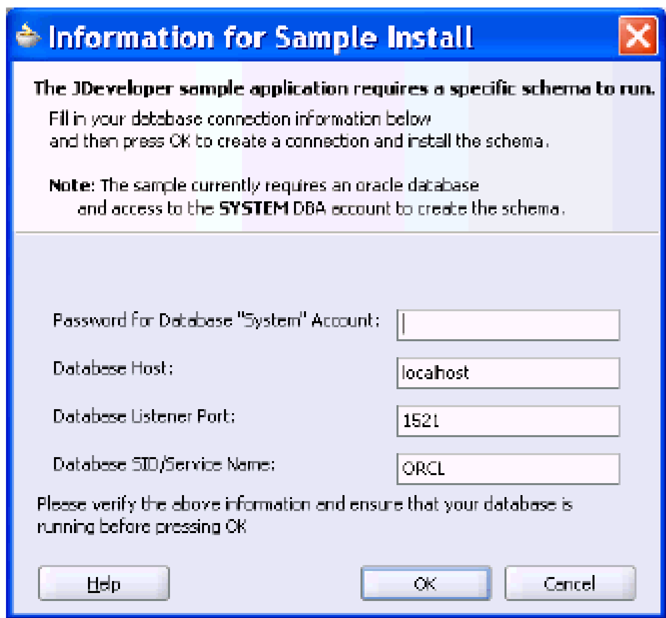
The SRDemo application is available for you to install as a JDeveloper extension. In JDeveloper, you use the Check for Updates wizard to begin the process of installing the extension.

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 (ADF BC Version)** and select it. Click **Next** to initiate the download.

   Note that the Update Center may display two versions of the SRDemo application. Only the one labeled "ADF BC" uses the technology stack described in this guide.
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 3–2 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/SRDemoSampleADFBC`. The Update Center also installs the extension JAR file `<JDEV_INSTALL>/jdev/extensions/oracle.jdeveloper.srdemo.bc.10.1.3.jar` which allows JDeveloper to create the SRDemo application workspace.

3.2.2 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/SRDemoSampleADFBC/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 3.2.1, "Downloading and Installing the Oracle ADF Service Request Application".

Follow these instructions to manually create the SRDemo schema.

To manually install the SRDemo schema:

- In JDeveloper, open the ANT build file `build.xml` in the BuildAndDeploy project of the SRDemoSampleADFBC 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.

To manually refresh data in the SRDemo schema:

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

If the schema already exists, you can use the above tasks to refresh the SRDemo sample data. You can either recreate the schema and data entirely (using the `setupDBOracle` task) or you can avoid recreating the schema and merely repopulate the SRDemo data (`resetData` task). Repopulating the schema is the easiest way to refresh the sample data since you will not need to log in as a SYS user. If you decide to reinstall the schema manually, you will be required to enter the SYSTEM DBA account password before the task recreates the schema.

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.

3.2.3 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 3.1.2, "Overview of the Schema".

3.2.4 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 3–3.

Figure 3–3 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**.

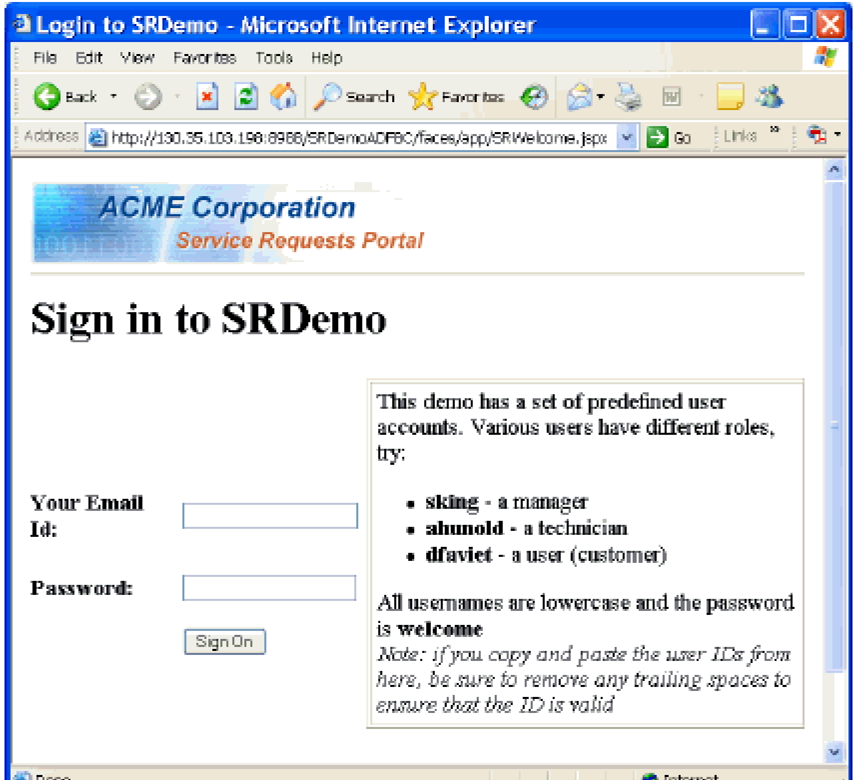
Running the `index.jspx` page from inside JDeveloper will start the embedded Oracle Application Server 10*g* Oracle Containers for J2EE (OC4J) server, launch your default browser, and cause it to request the following URL:

```
http://130.35.103.198:8988/SRDemoADFBC/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 3–4.

> **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 3–4 SRWelcome.jspx: SRDemo Login Page



See Section 3.3, "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.

3.2.5 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.

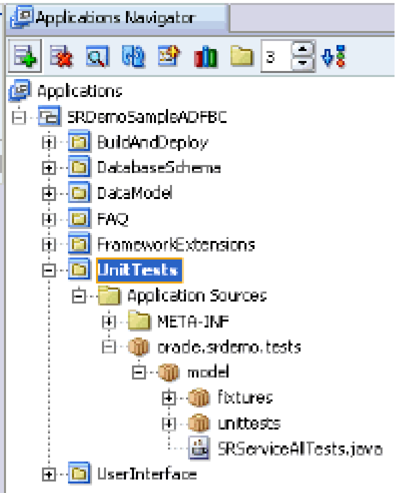
Note: You may skip the following procedure if you installed the SRDemo application using the Update Center in JDeveloper. In that case, the JUnit extension will automatically be installed when you download the application.

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 `SRServiceAllTests.java` class shown in Figure 3–5. To run the regression test suite, select the `SRServiceAllTests.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 3–5 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 testiest that is passing.

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

3.3 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 the chapters listed in Part II of this guide.

For example, the SRDemo application implements the following business logic:

ServiceRequest Entity Object

- Assigned Date must be in the current month
- Assigned Date must be after Request Date
- Request Date defaults to current date
- Status attribute Must be 'Open', 'Pending', or 'Closed'
- ProdId value must be a valid product id in the PRODUCTS table
- Cannot remove a service request if you are not a manager
- Service Request status transitions to "Open" when customer adds a note
- Service Request status transitions to "Pending" when technician adds a note

ServiceHistory Entity Object

- Line Number and Service Request ID defaulted for new service request notes
- Only staff can mark a service request note hidden
- Type of service request note defaults based on current user role

3.3.1 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.

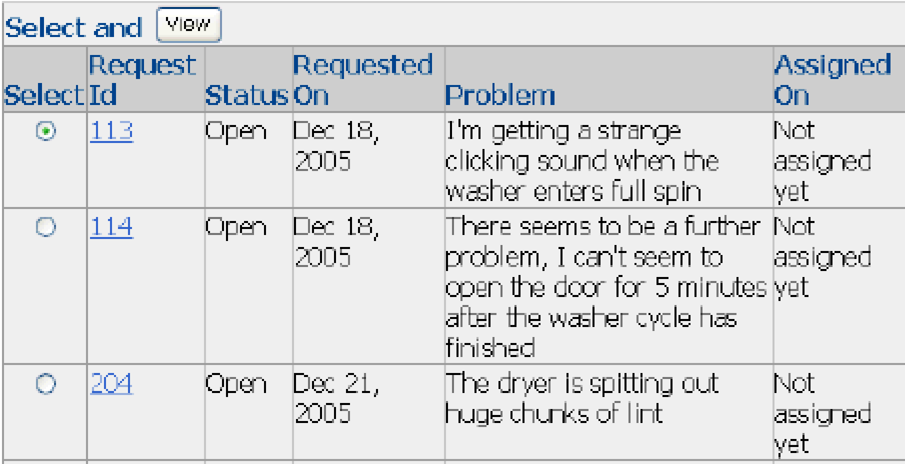
Sign On

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

Start

This action displays the customer's list of open service requests, as shown in Figure 3–6.

Figure 3–6 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** 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 3–7 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 3–7 SRMain.jspx: Main Page for a Customer

Logged in as **dfaviet**

Service Request Information for SR # 100

Product: Washing Machine W001
Requested By: Daniel Faviet
Assigned To: Alexander Hunold
Problem: I have noticed that every time I do a wash there is a pool

Add a note | Upload a document

Note:

Add | Cancel | Clear Fields

Date	Type	Note
Dec 15, 2005	Technician	Had customer check hoses to see if they are leaking
Dec 16, 2005	Customer	Everything works it was the hose to washing machine connector

Where to Find Implementation Details

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

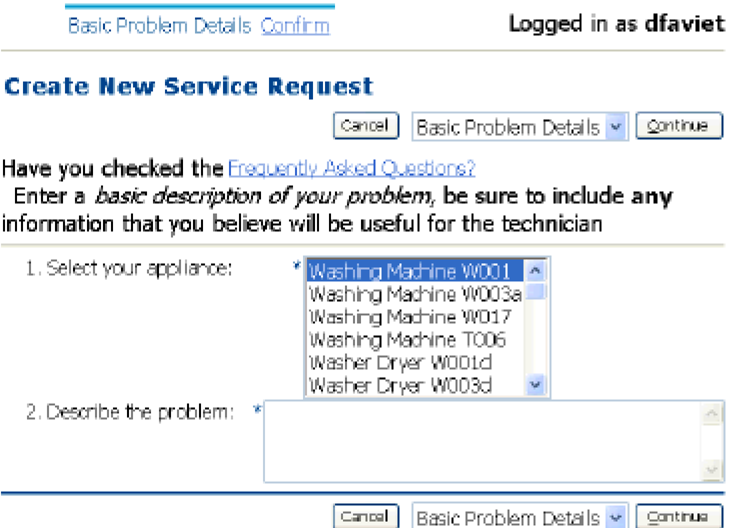
- Using dynamic navigation menus: The menu tabs and subtabs which let the user access the desired pages of the application, are created declaratively by binding each menu component to a menu model object and using the menu model to display the appropriate menu items. See Section 19.2, "Using Dynamic 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 14.2, "Creating a Basic Table".
- 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 15.3, "Using Tables and Forms to Display Master-Detail Objects".

3.3.2 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 3–8 shows the first page.

Figure 3–8 SRCreate.jspx: Step One, Create-Service-Request Page

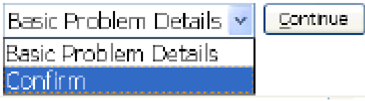


You can see how the application handles validation errors by clicking the **Continue** button before entering a problem description.

Figure 3–9 shows the validation error that displays within a dialog when the problem description is not entered.

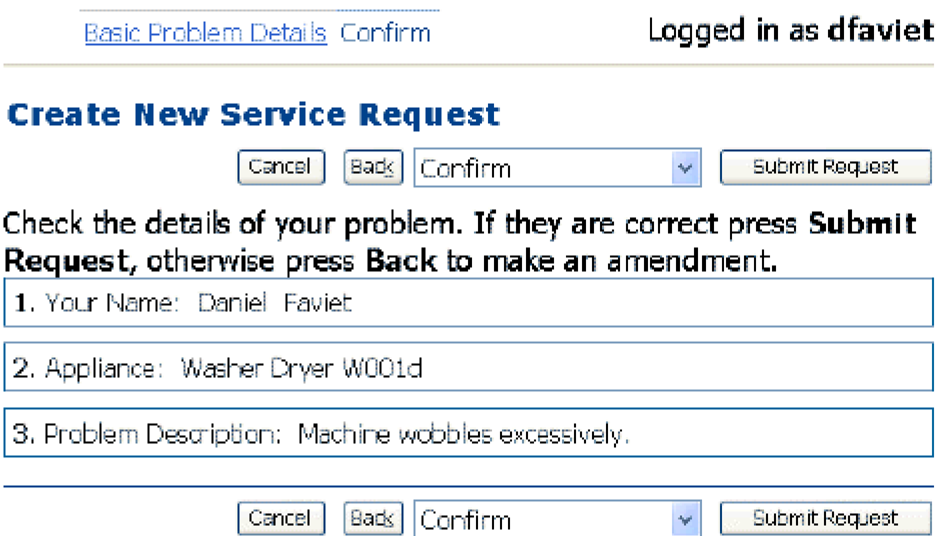
Figure 3–9 SRCreate.jspx: Step One Validation Error in Separate Dialog

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 3–10 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 3–10 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.

Logout

Where to Find Implementation Details

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

- Creating a new record: The user adds a note to a service request using a form that commits the data to the data source. JDeveloper lets you invoke the built-in Create operation on the application module as an easy way to drop record creation forms. See Section 13.6, "Creating an Input Form".
- Multipage process: The ADF Faces components `processTrain` and `processChoiceBar` guide the user through the process of creating a new service request. See Section 19.5, "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 20.3, "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 16.1, "Introduction to Page Navigation".

3.3.3 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.

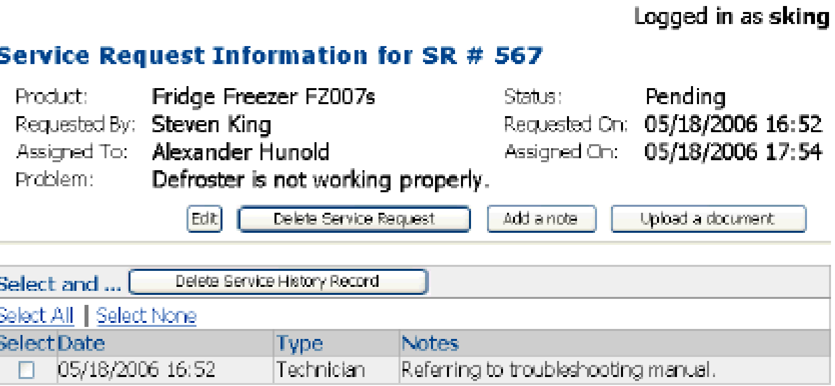
Click the **Start** button.

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 3–11 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.

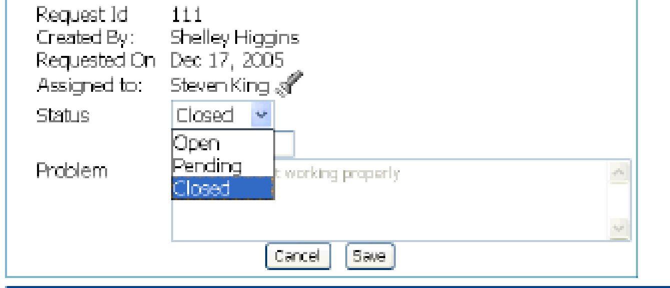
Figure 3–11 SRMain.jspx: Main Page for a Manager



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

Figure 3–12 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.

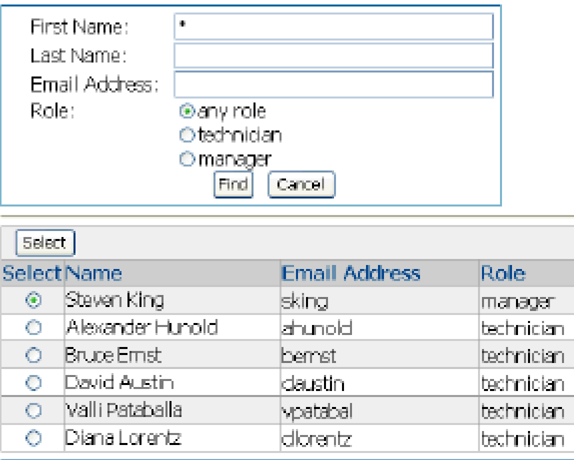
Figure 3–12 SREdit.jspx: Edit Page for a Manager



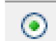
To find another technician to assign, click the symbol next to the assigned person's name.

Figure 3–13 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 3–13 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 19.7, "Creating Selection 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. Wildcard search is supported. See Section 18.3, "Creating a Web-type Search Form".
- 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 20.7, "Displaying Error Messages" and Section 19.3, "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 19.4, "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 17.2, "Using a Managed Bean to Store Information".

3.3.4 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 3–14 shows the tree table with an expanded technician node.

Figure 3–14 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:

Logged in as **sking**

Management Reporting

Staff with Open/Pending issues

- Steven King
- Alexander Hunold
  - [Open] Defroster is not working properly
  - [Open] Dryer is spitting out lots of lint.
- Bruce Ernst
- David Austin
- Valli Pataballa

Problem History for Service Request 111

Open - Created on *December 17, 2005 8:59 AM*

Complete Problem Text:

Defroster is not working properly

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:

Logged in as **sking**

Management Reporting

Staff with Open/Pending issues

- Steven King
- Alexander Hunold
  - [Open] Defroster is not working properly
  - [Open] Dryer is spitting out lots of lint.
- Bruce Ernst
- David Austin
- Valli Pataballa

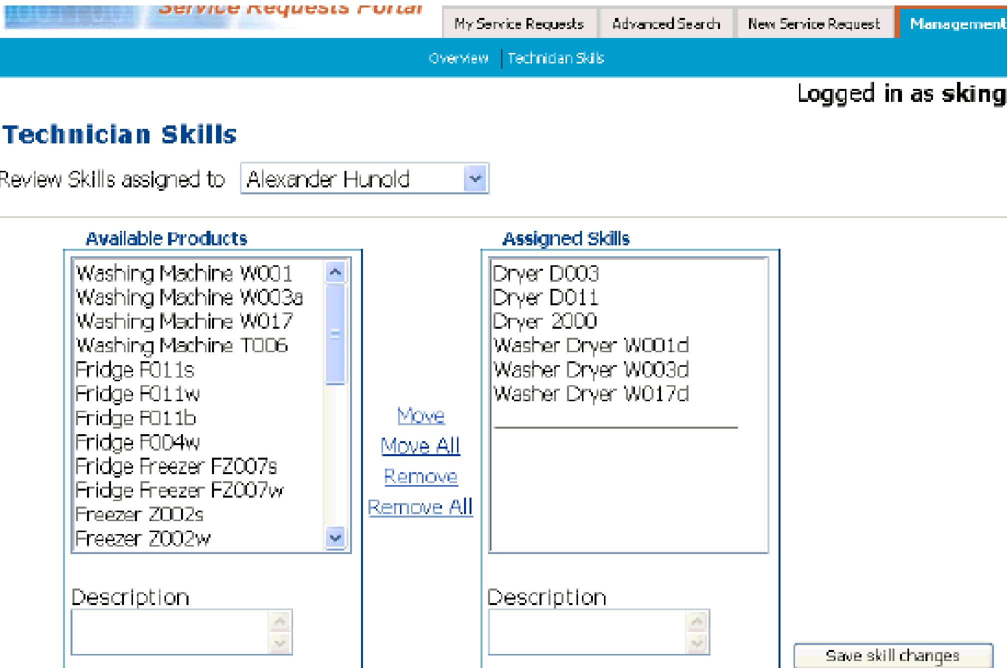
Skills information for Alexander Hunold

Product Area	Expertise Level
Dryer D003	Qualified
Dryer D011	Qualified
Dryer 2000	Qualified
Washer Dryer W001d	Expert
Washer Dryer W003d	Qualified
Washer Dryer W017d	Qualified

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 3–15 shows the shuttle component populated with the skills of the selected staff member.

Figure 3–15 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 19.8, "Creating a 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 30.8, "Implementing Authorization Programmatically".

3.3.5 Technician Logs In and Updates a Service Request

Enter the log in information for a technician:

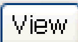
- **User name**: `ahunold`
- **Password**: `welcome`

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**.

Upload a document

Figure 3–16 shows the file upload window. (Please note the SRDemo application currently provides no means to view the contents of the uploaded document.)

Figure 3–16 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 19.6, "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 22.3, "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 22.4, "Internationalizing Your Application".

Part II

Building Your Business Services

Part II contains the following chapters:

- Chapter 4, "Overview of ADF Business Components"
- Chapter 5, "Querying Data Using View Objects"
- Chapter 6, "Creating a Business Domain Layer Using Entity Objects"
- Chapter 7, "Building an Updatable Data Model With Entity-Based View Objects"
- Chapter 8, "Implementing Business Services with Application Modules"
- Chapter 9, "Implementing Programmatic Business Rules in Entity Objects"
- Chapter 10, "Overview of Application Module Data Binding"

4 Overview of ADF Business Components

This chapter provides an overview of the ADF Business Components layer of Oracle ADF, including a description of the key features they provide for building your business services.

This chapter includes the following sections:

- Section 4.1, "Prescriptive Approach and Reusable Code for Business Services"
- Section 4.2, "What are ADF Business Components and What Can They Do?"
- Section 4.3, "Relating ADF Business Components to Familiar 4GL Tools"
- Section 4.4, "Overview of ADF Business Components Implementation Architecture"
- Section 4.5, "Understanding the Active Data Model"
- Section 4.6, "Overview of ADF Business Components Design Time Facilities"

4.1 Prescriptive Approach and Reusable Code for Business Services

The J2EE platform defines a server-side model for development and deployment of services. However, the task of writing, reusing, and customizing the robust functionality needed for real-world business applications is left as an exercise for the members of each development team to figure out for themselves. In particular, the J2EE specifications do not prescribe an approach for:

- writing and enforcing business application logic in a central way
- reusing business logic in multiple applications
- accessing updatable views of business data tailored specifically to the task at hand
- maintaining and customizing the business functionality once the application is delivered.

From years of experience in building the E-Business Suite applications on the J2EE platform, Oracle knows that *these* are the activities where you'll spend the bulk of your time and effort when you build your own J2EE solutions. ADF Business Components is designed to provide a prescriptive approach to address the these challenging tasks, and offers a reusable library of software components and design time "plugins" to JDeveloper that make it easy to follow the time-tested approach they prescribe.

The ADF Business Components technology in Oracle ADF is the culmination of years of joint design and development work between the Oracle Applications, Oracle Tools, and Oracle Server Technologies divisions to pragmatically implement Oracle's vision for how well-architected, database-centric enterprise J2EE applications will be built now and in the future.

Along with the other layers of the overall Oracle ADF, ADF Business Components is the technology used daily by over 4000 of Oracle's own internal applications developers, and by several thousand external customers including Oracle partners. This means it is a proven solution you can count on, too.

4.2 What are ADF Business Components and What Can They Do?

ADF Business Components are "building-blocks" that provide the most productive way to create, deploy, and maintain a business service. ADF Business Components dramatically simplifies the development, delivery, and customization of enterprise J2EE business applications by providing you with a set of intelligent software building-blocks that save development time by making many of the most typical development task declarative. They cooperate "out-of-the-box" to manage all of the common facilities required to:

- Productively author and test business logic in components which automatically integrate with databases,
- Flexibly reuse business logic through multiple SQL-based views of data, supporting different application tasks,
- Efficiently access and update the views from browser, desktop, mobile, and web service clients
- Easily customize application functionality in layers without requiring modification of the delivered application.

By eliminating the substantial coding and testing work related to common "application plumbing" facilities, ADF Business Components lets application developers focus full-time on implementing business solutions. ADF Business Components provides a foundation of Java classes that your business-tier application components extend to leverage a robust implementation of the numerous design patterns you need in the following areas:

Simplifying Data Access

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

Enforcing Business Domain Validation and Business Logic

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

Supporting Sophisticated UIs with Multi-Page Units of Work

- Automatically reflect changes made by business service application logic in the user interface,
- Retrieve reference information from related tables, and automatically maintain the information when user changes foreign-key values
- Simplify multi-step web-based business transactions with automatic web-tier state management
- Handle images, video, sound, and documents with no code
- Synchronize pending data changes across multiple views of data
- Consistently apply prompts, tooltips, format masks, and error messages in any application
- Define custom metadata for any business components to support metadata-driven user interface or application functionality.
- Add dynamic attributes at runtime to simplify per-row state management.

Implementing Best Practice, High-Performance Service-Oriented Architecture

- Enforce best-practice interface-based programming style
- Simplify application security with automatic JAAS integration and audit maintenance
- "Write once, deploy any": use the same business service as plain Java class, EJB session bean, or web service
- Switch from 2-tier to 3-tier deployment with no client code changes
- Reduce network traffic for remote clients through efficient batch operations

Streamlining Application Customization

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

All of these features can be summarized by saying that using ADF Business Components for your J2EE business service layer makes your life a lot easier. The key ADF Business Components components that cooperate to provide the business service implementation are:

- *Entity Object*

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

- *Application Module*

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

- *View Object*

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

While the base components handle all the common cases with their sophisticated built-in behavior, taking advantage of their benefits does **not** compromise your ability to have it your way whenever necessary. Since any automatic behavior provided by the base components can be easily overridden with a few strategic lines of code, you're never locked in to a certain way of doing things for all cases.

4.3 Relating ADF Business Components to Familiar 4GL Tools

ADF Business Components provides components that implement functionality similar to what you are used to in the enterprise 4GL tools you have used prior to embarking on J2EE development. This section will assist you in understanding how the key components in ADF Business Components map conceptually to the ones you have used in other 4GL tools.

4.3.1 Familiar Concepts for Oracle Forms Developers

ADF Business Components implements all of the data-centric aspects of the familiar Oracle Forms runtime functionality in a way that they can be used with any kind of user interface. In Oracle Forms, each *form* contains both visual objects like canvases, windows, alerts, and LOVs, as well as non-visual objects like data blocks, relations, and record groups. Individual data block *items* have both visual properties like `Foreground Color` and `Bevel` as well as non-visual properties like `Data Type` and `Maximum Length`. Even the different event-handling triggers that Forms defines fall into visual and non-visual categories. For example, it's clear that triggers like `WHEN-BUTTON-PRESSED` and `WHEN-MOUSE-CLICKED` are visual in nature, relating to the front-end UI, while triggers like `WHEN-VALIDATE-ITEM` and `ON-INSERT` are more related to the back-end data processing. While merging visual and non-visual aspects definitely simplifies the learning curve, the flip side is that it can complicate reuse. With a cleaner separation of UI-related and data-related elements, it would be easier to redesign the user interface without disturbing back-end business logic and easier to repurpose back-end business logic in multiple different forms.

In order to imagine this separation of UI and data, consider lopping a form as you know it today in half, keeping only its non-visual, data-related aspects. What's left would be a container of data blocks, relations, and record groups. This container would continue to provide a database connection for the data blocks to share and be responsible for coordinating transaction commits or rollbacks. Of course, you could still use the *non-visual* validation and transactional triggers to augment or change the default data-processing behavior as well. This non-visual object your considering is a kind of a "smart data model" or a generic *application module*, with data and business logic, but no user interface elements. The goal of separating this application module from anything visual is to allow any kind of user interface you need in the future to use it as a data service.

Focus a moment on the role the *data blocks* would play in this application module. They would query rows of data from the database using SQL, coordinate master/detail relationships with other data blocks, validate user data entry with `WHEN-VALIDATE-RECORD` and `WHEN-VALIDATE-ITEM` triggers, and communicate valid user changes back to the database with `INSERT`, `UPDATE`, and `DELETE` statements when you commit the data service's transaction.

Experience tells you that you need to filter, join, order, and group data for your end-users in many different ways depending on the task at hand. On the other hand, the *validation rules* that you apply to your business domain data remain basically the same over time. Given these observations, it would be genuinely useful to write business entity validation exactly once, and leverage it consistently anywhere that data is manipulated by users in your applications.

Enabling this flexibility requires further "factoring" of your data block functionality. You need one kind of "SQL query" object to represent each of the many different views of data your application requires, and another kind of "business entity" object that will enforce business rules and communicate changes to your base table in a consistent way. By splitting things again like this, you can have multiple different "view objects" working with the same underlying "entity object" when their SQL queries present the same business data.

Oracle ADF breathes life into the UI/data split you imagined above by providing ready-to-use Java components that implement the Forms-inspired functionality you're familiar with in the Java world, with responsibilities divided along the cleanly-separated functional lines you just hypothesized.

Application Module is a "headless" Form Module

The `ApplicationModule` component is the "data half of the form" you considered above. It's a smart data service containing a *data model* of master/detail-related queries that your client interface needs to work with. It also provides a transaction and database connection used by the components it contains. It can contain form-level procedures and functions called service methods, that are encapsulated within the service implementation. You can decide which of these procedures and functions should be private and which ones should be public.

The Entity Object validates and saves rows like the Forms Record Manager

The `EntityObject` implements the "validation and database changes half" of the data block functionality from above. In the Forms runtime, this duty is performed by the record manager. It is responsible for keeping track of which of the rows in a data block have changed, for firing the validation triggers in a data block and its data items, and for coordinating the saving of changes to the database. This is exactly what an entity object does for you. Related to an underlying base table, it's a component that represents your business domain entity and gives you a single place to encapsulate business logic related to validation, defaulting, and database modification behavior for that business object.

The View Object queries data like a Data Block

The `ViewObject` component performs the "data retrieval half" of the data block functionality above. Each view object encapsulates a SQL query, and at runtime each one manages its own query result set. If you connect two or more view objects in master/detail relationships, that coordination is handled automatically. While defining a view object, you can link any of its query columns to underlying entity objects. By capturing this information, the view object and entity object can cooperate automatically for you at runtime to enforce your domain business logic regardless of the "shape" of the business data needed by the user for the task at hand.

4.3.2 Familiar Concepts for PeopleTools Developers

If you have developed solutions in the past with PeopleTools, you are familiar with the PeopleTools Component structure. ADF Business Components implement the data access functionality you are familiar with from PeopleTools.

Application Module is a “headless” Component

ADF adheres to an MVC pattern and separates the Model from the View. Pages as you are familiar with in the PeopleTools Component are defined in the view layer, using standard technologies like with JSF and ADF Faces components for web-based applications or Swing for desktop-fidelity client displays.

The ADF Application Module defines the data structure, just like the PeopleTools Component Buffer. By defining master/detail relationships between ADF query components that produce row sets of data, you ensure that any Application Module that works with the data can reuse the natural hierarchy as required, similar to the scroll levels in the Component Buffer.

Similar to the Component Interface you are familiar with, the Application Module is a service object that provides access to standard methods as well as additional developer-defined business logic. In order to present a "headless" data service for a particular user interface, the Component Interface restricts a number of PeopleTools functions that are related to UI interaction. The Application Module is similar to the Component Interface in that it provides a "headless" data service, but in contrast it does not do this by wrapping a restricted view of an existing user interface. Instead, the Application Module is architected to deal exclusively with business logic and data access. Rather than building a Component Interface on top of the Component, with ADF you first build the Application Module service that is independent of user interface, and then build one or more pages on top of this service to accomplish some end-user task in your application.

The Application Module is associated with a Transaction object in the same way that the PeopleTools Component Buffer is. The Application Module also provides a database connection for the components it contains. Any logic you associate today with the transaction as Component PeopleCode, in ADF you would define as logic on the Application Module.

Logic associated with records in the transaction, that today you write as Component Record PeopleCode or Component Record Field PeopleCode, should probably *not* be defined on the Application Module. ADF has View Objects (see below) that allow for better re-use when the same Record appears in different Components.

Entity Object is a Record Definition

The Entity Object is the mapping to the underlying data structure, just like the PeopleTools Record Definition maps to the underlying table or view. You'll often create one Entity Object for each of the tables that you need to manipulate your application.

Similar to how you declare a set of valid values for fields like 'Customer Status' using PeopleTools' translate values, in ADF you can add declarative validations to the individual Attributes of an Entity Object. Any logic you associate with the record that applies throughout your applications, which today you write as Record PeopleCode or Record Field PeopleCode, can be defined in ADF on the Entity Object.

View Object Queries Data Like a Row Set

Just like a PeopleTools row set, a View Object can be populated by a SQL query. Unlike a row set, a View Object definition can contain business logic.

Any logic, which you would find in Component Record PeopleCode, is a likely candidate to define on the View Object. Component Record PeopleCode is directly tied to the Component, but a View Object can be associated with different Application Modules. While you can use the same Record Definition in many PeopleTools Components, Oracle ADF allows you to reuse the business logic across multiple applications.

The View Object queries data in exactly the "shape" that is useful for the current application. Many View Objects can be built on top of the same Entity Object.

You can define relationships between View Objects to create master-detail structures just like you find them in the scroll levels in the PeopleTools Component.

4.3.3 Familiar Concepts for SiebelTools Developers

If you have developed solutions in the past with SiebelTools version 7.0 or earlier, you will find that ADF Business Components implements all of the familiar data access functionality you are familiar with, with numerous enhancements.

Entity Object is a Table Object with Encapsulated Business Logic

Like the Siebel Table object, the ADF Entity Object describes the physical characteristics of a single table, including column names and physical data types. Both objects contain sufficient information to generate the DDL to create the physical tables in the DB. In ADF you define Associations between Entity Objects to reflect the foreign keys present in the underlying tables, and these associations are used automatically join business information in the view object queries used by user interface pages or screens. List of values objects that you reference from data columns today are handled in ADF through a combination of declarative entity validation rules and view object queries. You can also encapsulate other declarative or programmatic business logic with these entity object "table" handlers that is automatically reused in any view of the data you create.

View Object is a Business Component

Like the Siebel Business Component, the ADF View Object describes a logical mapping on top of the underlying physical table representation. They both allow you to provide logical field names, data, and calculated fields that match the needs of the user interface. As with the Business Component, you can define View Objects that join information from various underlying tables. The related ADF View Link is similar to the Siebel Link object and allows you to define master/detail relationships. In ADF, your view object definitions can exploit the full power of the SQL language to shape the data as required by the user interface.

Application Module is a Business Object With Connection and Transaction

The Siebel Business Object lets you define a collection of Business Components. The ADF Application Module performs a similar task, allowing you to create a collection of master/detail View Objects that act as a "data model" for a set of related user interface pages. In addition, the Application Module provides a transaction and database connection context for this group of data views. You can make multiple requests to objects obtained from the Application Module and these participate in the same transaction.

4.3.4 Familiar Functionality for ADO.NET Developers

If you have developed solutions in the past with Visual Studio 2003 or 2005, you are familiar with using the ADO.NET framework for data access. ADF Business Components implements all of the data access functionality you are familiar with from ADO.NET, with numerous enhancements.

Application Module is an enhanced DataSet

The `ApplicationModule` component plays the same role as the ADO.NET `DataSet`. It is a strongly-typed service component that represents a collection of row sets called view objects, which as described below, are similar to ADO.NET `DataTables`. The application module works with a related `Transaction` object to provide the context for queries the SQL queries the view objects execute and the modifications saved to the database by the entity objects, which play the role of the ADO.NET `DataAdapter`.

The Entity Object is an enhanced, strongly-typed DataAdapter

The `EntityObject` is like a strongly-typed ADO.NET `DataAdapter`. It represents the rows in a particular table and handles the find-by-primary-key, insert, update, delete, and lock operations for those rows. In ADF, you don't have to specify these statements yourself, but you can override them if you need to. The entity object encapsulates validation or other business logic related to attributes or entire rows in the underlying table. This validation is enforced when data is modified and saved by the end-user using any view object query that references the underlying entity object.

The View Object is an enhanced DataTable

The `ViewObject` component encapsulates a SQL query and manages the set of resulting rows. It can be related to an underlying entity object to automatically coordinate validation and saving of modifications made by the user to those rows. This cooperation between a view object's queried data and an entity objects encapsulated business logic offers all of the benefits of the DataTable with the clean encapsulation of business logic into a layer of business domain objects. Like ADO.NET data tables, you can easily work with a view object's data as XML or have a view object read XML data to automatically insert, update, or delete rows based on the information it contains.

4.4 Overview of ADF Business Components Implementation Architecture

Before diving into each of the key components in subsequent chapters, it's good at the outset to understand a few guiding principles that have gone into the design and implementation of this layer of Oracle ADF.

4.4.1 Based on Standard Java and XML

Like the rest of Oracle ADF, the ADF Business Components technology is implemented in Java. The base components implement a large amount of generic, metadata-driven functionality to save you development time by standing on the shoulders of a rich layer of working, tested code. The metadata for ADF Business Components follow J2EE community best practice of using cleanly-separated XML files to hold the metadata that configures each component's runtime behavior.

Since ADF Business Components is often used for bet-your-business applications, it's important to understand that full source for Oracle ADF, including the ADF Business Components layer, is available to supported customers through Oracle Worldwide Support. The full source code for the framework can be an important tool to assist you in diagnosing problems and in correctly extending the base framework functionality for your needs.

4.4.2 Works with Any Application Server or Database

Because your business components are implemented using plain Java classes and XML files, you can use them in any runtime environment where a Java Virtual Machine is present. This means that services built using ADF Business Components are easy to use both inside a J2EE server — known as the "container" of your application at runtime — as well as outside. Customers routinely use application modules in such diverse configurations as command-line batch programs, web services, custom servlets, JSP pages, desktop-fidelity clients built using Swing, and others.

Applications built using ADF Business Components can run on any Java-capable application server, including any J2EE-compliant application server. As described in Section 4.6.1, "Choosing a Connection, SQL Flavor, and Type Map", in addition to building applications that target Oracle databases with numerous optimizations, you can also build applications that work with non-Oracle databases.

4.4.3 Implements All of the J2EE Design Patterns You Need

The ADF Business Components layer implements all of the popular J2EE design patterns that you would normally need to understand, implement, and debug yourself to create a real-world enterprise J2EE application. If it is important to you to cross-reference the names of some of these design patterns you might have read about in J2EE literature with how they are implemented by ADF Business Components, you can refer to Appendix E, "ADF Business Components J2EE Design Pattern Catalog".

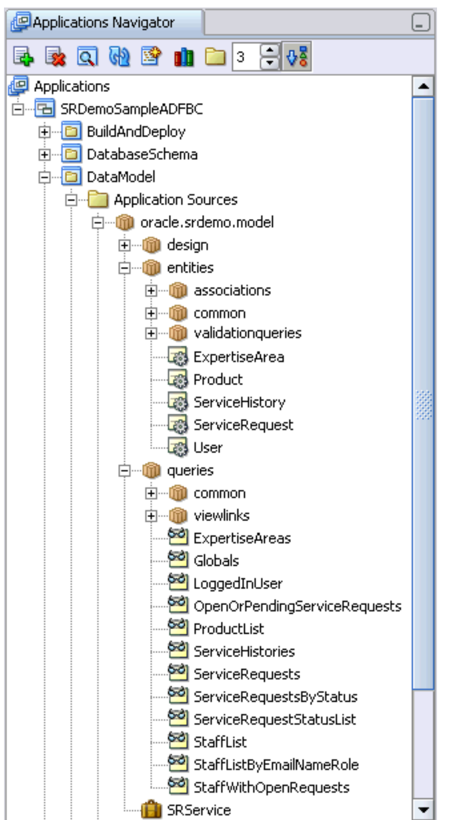
4.4.4 Components are Organized into Packages

Since ADF Business Components is implemented in Java, it is implemented in Java classes and interfaces that are organized into packages. Java packages are identified by dot-separated names that developers use to arrange code into a hierarchical naming structure. To ensure your code won't clash with reusable code from other organizations, best practice dictates choosing package names that begin with your organization's name or web domain name. So, for example, the Apache organization chose `org.apache.tomcat` for a package name related to its Tomcat web server, while Oracle picked `oracle.xml.parser` as a package name for its XML parser. Components you create for an your own applications will live in a packages with names like `com.`*`yourcompany`*`.`*`yourapp`* and subpackages of these.

As a specific example, the ADF Business Components that make up the main business service for the SRDemo application are organized into the `oracle.srdemo.model` package, and subpackages. As shown in Figure 4–1, these components reside in the `DataModel` project in the workspace, and are organized broadly as follows:

- `oracle.srdemo.model` contains the `SRService` application module
- `oracle.srdemo.model.queries` contains the view objects
- `oracle.srdemo.model.entities` contains the entity objects
- `oracle.srdemo.model.design` contains UML diagrams documenting the service

Figure 4–1 Organization of ADF Business Components in the SRDemo Application



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

Due to JDeveloper's support for refactoring, you can easily rename or move components to a different package structure at any time. In other words, you don't need to necessarily get the structure right the first time. Your business service's package structure will almost certainly evolve over time as you gain more experience with the ADF environment.

There is no "magic" number that describes the optimal number of components in a package. However, with experience, you'll realize that the correct structure for your team falls somewhere between the two extremes of:

- All components in a single package
- Each component in its own, separate package

As described in more detail in Section 25.7, "Working with Libraries of Reusable Business Components", since a package of ADF Business Components is the unit of granularity that JDeveloper supports importing for reuse in other projects, sometimes you'll also factor this consideration into how you choose to organize components.

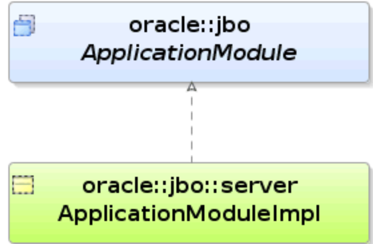
4.4.5 Architecture of the Base ADF Business Components Layer

The classes and interfaces that comprise the pre-built code provided by the ADF Business Components layer live in the `oracle.jbo` package and numerous subpackages, however in your day to day work with ADF Business Components you'll mostly be working with classes and interfaces in the two key packages `oracle.jbo` and `oracle.jbo.server`. The `oracle.jbo` package contains all of the interfaces that are designed for the business service client to work with, while the `oracle.jbo.server` package contains the classes that implement these interfaces.

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

Figure 4–2 shows a concrete example of the application module component. The client interface for the application module is the `ApplicationModule` interface in the `oracle.jbo` package. This interface defines the names and signatures of methods that clients can use while working with the application module, but it does not include any specifics about the implementation of that functionality. The class that implements the base functionality of the application module component lives in the `oracle.jbo.server` package and is named `ApplicationModuleImpl`.

Figure 4–2 Oracle ADF Business Components Separate Interface and Implementation



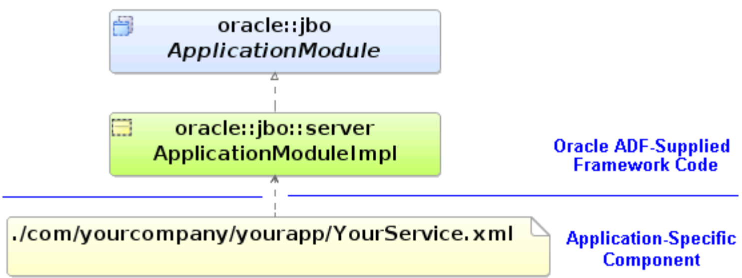
4.4.6 Components Are Metadata-Driven With Optional Custom Java Code

Each kind of component in ADF Business Components comes with built-in runtime functionality that you control through declarative settings. These settings are stored in an XML component definition file with the same name as the component that it represents. When you need to write custom code for a component, you can enable an optional custom Java class for the component in question.

4.4.6.1 Example of an XML-Only Component

Figure 4–3 illustrates the XML component definition file for an application-specific component like an application module named `YourService` that you create in a package named `com.yourcompany.yourapp`. The corresponding XML component definition resides in a `./com/yourcompany/yourapp` subdirectory of the JDeveloper's project's source path root directory. That XML file records the name of the Java class it should use at runtime to provide the application module implementation. In this case, the XML records the name of the base `oracle.jbo.server.ApplicationModuleImpl` class provided by Oracle ADF.

Figure 4–3 XML Component Definition File for an Application Module



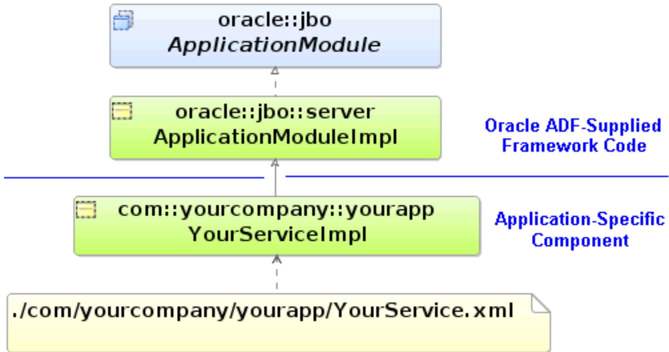
If you have no need to extend the built-in functionality of a component in ADF Business Components, and no need to write any custom code to handle its built-in events, you can use the component in this XML-only fashion. This means your component is completely defined by its XML component definition and be fully-functionality without requiring any custom Java code or even a Java class file related to the component at all.

4.4.6.2 Example of a Component with Custom Java Class

When you need to add custom code to extend the base functionality of a component or to handle events, you can enable a custom Java class for any of the key types of ADF Business Components you create. You enable custom classes for a component on the **Java** panel of its respective component editor in JDeveloper. This creates a Java source file for a custom class related to the component whose name follows a configurable naming standard. This class, whose name is recorded in the component's XML component definition, provides a place where you can write the custom Java code required by that component. Once you've enabled a custom Java class for a component, you can navigate to it at any time using a corresponding **Go To... Class** option in the component's Application Navigator context menu.

Figure 4–4 illustrates what occurs when you enable a custom Java class for the `YourService` application module considered above. A `YourServiceImpl.java` source code file is created in the same directory in the source path as your component's XML component definition file. The `YourService.xml` file is updated to reflect the fact that at runtime the component should use the `com.yourcompany.yourapp.YourServiceImpl` class instead of the base `ApplicationModuleImpl` class.

Figure 4–4 Component with Custom Java Class



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

4.4.7 Recommendations for Configuring ADF Business Components Design Time Preferences

You can configure whether JDeveloper generates custom Java files by default for each component type that supports it, as well as whether JDeveloper maintains a list of Oracle ADF business components in each package using a package XML file. This section describes Oracle's recommendations to developers getting started with ADF Business Components on how to configure these options.

4.4.7.1 Recommendation for Initially Disabling Custom Java Generation

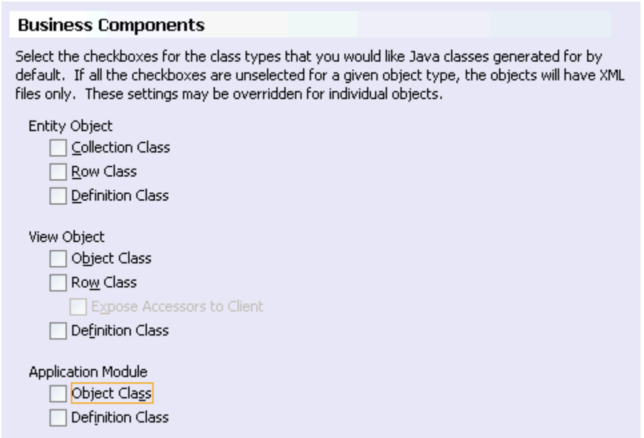
Your applications can freely mix XML-only components with components that have related custom Java files. For example, you can define a completely functional, updatable data model with declaratively enforced business rules using XML-only components. On the other end of the spectrum, some developers prefer to proactively generate Java classes for each component they create as part of their team's coding style.

For developers getting started with ADF Business Components, Oracle recommends initially configuring JDeveloper to not generate any custom Java classes by default. This way, you learn the reasons why custom Java is needed and you consciously enable it for the components that require it in your application. Over time, you will develop a personal preference of your own.

Note that this recommended setting is not the default, so you need to perform the following steps to configure the Java generation preferences as recommended here:

- Choose **Tools** | **Preferences...** from the JDeveloper main menu
- Select the **Business Components** preference category in the tree at the left
- Ensure all of the checkboxes are unchecked as shown in Figure 4–5, then click **OK**.

Figure 4–5 Setting Business Components Preferences to Generate No Java By Default



4.4.7.2 Recommendation for Disabling Use of Package XML File

By default, for upward compatibility with previously releases of Oracle ADF, JDeveloper maintains an XML file in each directory containing the names of the Oracle ADF business components that reside in that package. While previously required by the ADF runtime classes, this package XML file is optional in this version. Since maintaining this "package XML" file can complicate team development, Oracle recommends you disable the use of any package XML files by setting the **Copy Package XML Files to Class Path** option off in the **Business Components: General** panel of the IDE preferences as shown in Figure 4–6.

Figure 4–6 Disabling the Use of the Optional Package XML File for ADF Business Components

Business Components: General

☐ Generate Comments for Custom Code Location

☐ Copy Package XML Files to Class Path

Note: To disable the use of package XML files in an existing project containing ADF Business Components, you can visit the **Project Properties** dialog, select the **Business Components: Options** panel, and uncheck the same checkbox as shown above.

4.4.8 Basic Datatypes

The Java language provides a number of built-in data types for working with strings, dates, numbers, and other data. When working with ADF Business Components, you can use these types, but by default you'll use an optimized set of types in the `oracle.jbo.domain` and `oracle.ord.im` packages. These types, shown in Table 4–1, improve the performance of working with data from the Oracle database by allowing the data to remain in its native, internal format avoiding costly type conversions when they are not necessary. For working with string-based data, by default ADF Business Components uses the regular `java.lang.String` type.

Table 4–1 Basic Data Types in the oracle.jbo.domain and oracle.ord.im Packages

Data Type	Represents
`Number`	Any numerical data
`Date`	Date with optional time
`DBSequence`	Sequential integer assigned by a database trigger
`RowID`	Oracle database ROWID
`Timestamp`	Timestamp value
`TimestampTZ`	Timestamp value with Timezone information
`BFileDomain`	Binary File (BFILE) object
`BlobDomain`	Binary Large Object (BLOB)
`ClobDomain`	Character Large Object (CLOB)
`OrdImageDomain`	Oracle Intermedia Image (ORDIMAGE)
`OrdAudioDomain`	Oracle Intermedia Audio (ORDAUDIO)

Table 4–1 (Cont.) Basic Data Types in the oracle.jbo.domain and oracle.ord.im Packages

Data Type	Represents
`OrdVideoDomain`	Oracle Intermedia Video (ORDVIDEO)
`OrdDocDomain`	Oracle Intermedia Document (ORDDOC)
`Struct`	User-defined object type
`Array`	User-defined collection type (e.g. `VARRAY`)

Note: The `oracle.jbo.domain.Number` class has the same class name as the built-in `java.lang.Number` type. Since the Java compiler *implicitly* imports `java.lang.*` into every class, you need to explicitly import the `oracle.jbo.domain.Number` class into any class that references this. Typically, JDeveloper will do this automatically for you, but when you begin to write more custom code of your own, you'll learn to recognize compiler or runtime errors related to "Number is an abstract class" mean that you are inadvertently using `java.lang.Number` instead of `oracle.jbo.domain.Number`. Adding the:

```
import oracle.jbo.domain.Number;
```

line at the top of your class, after the `package` line, avoids these kinds of errors.

4.4.9 Generic Versus Strongly-Typed APIs

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

```
Row row = serviceRequestVO.getCurrentRow();
Date requestDate = (Date)row.getAttribute("RequestDate");
```

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

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

```
ServiceRequestsRow row = (ServiceRequestRow)serviceRequestVO.getCurrentRow();
Date requestDate = row.getRequestDate();
```

In this case, you work with generated method names whose return type is known at compile time, instead of passing string names and having to cast the results. Subsequent chapters explain how to enable this strongly-typed style of working if you prefer it.

4.4.10 Client-Accessible Components Can Have Custom Interfaces

By design, the entity objects in the business domain layer of business service implementation are not designed to be referenced directly by clients. Instead, clients work with the data queried by view objects as part of an application module's data model. Behind the scenes, as you'll learn in Section 7.7, "Understanding How View Objects and Entity Objects Cooperate at Runtime", the view object cooperates automatically with entity objects in the business domain layer to coordinate validating and saving the data the user changes.

Therefore, the client-visible components of your business service are the:

- Application Module — representing the service itself
- View Objects — representing the query components
- View Rows — representing each row in a given query component's results

4.4.10.1 Framework Client Interfaces for Components

The `oracle.jbo` package provides client-accessible API for your business service as a set of Java interfaces. In line with the design mentioned above, this package does not contain any `Entity` interface, or any methods that allow the client to directly work with entity objects. Instead, client code works with interfaces like:

- `ApplicationModule` — to work with the application module
- `ViewObject` — to work with the view object
- `Row` — to work with the view rows

4.4.10.2 Custom Client Interfaces for Components

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

- Custom Application Module Interface

  ```
  SRService extends ApplicationModule
  ```

- Custom View Object Interface

  ```
  StaffListByEmailNameRole extends ViewObject
  ```

- Custom View Row Interface

  ```
  StaffListRowClient extends Row
  ```

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

4.5 Understanding the Active Data Model

One of the key simplifying benefits of using ADF Business Components for your business service implementation is the application module's support for an "active data model" of row sets. For developers coming from a Forms/4GL background, it works just like you are used to in previous tools.

4.5.1 What is an Active Data Model?

Using a typical J2EE business service implementation puts the burden on the client layer developer to be responsible for:

- Invoking service methods to return data to present,
- Tracking what data the client has created, deleted, or modified, and
- Passing the changes back to one or more different service methods to validate and save them.

The architects that designed the ADF application module recognized that this retrieve, create, edit, delete, and save cycle is so common in enterprise business applications that a smarter, more generic solution was required. Using the application module for your business service, you simply bind client UI controls like fields, tables and trees to the active view object instances in the application module's data model. Your UI displays automatically update to reflect any changes to the rows in the view object row sets in that data model. This includes displays you create using JSP or JSF pages for the web or mobile devices, as well as desktop-fidelity UI's comprising windows and panels using Swing. This "active" data notification includes changes to the data model that are the result of work performed directly or indirectly by your custom business service methods, too.

Under the covers the application module component implements a set of *generic* service methods to enable the active data model facility in a Service Oriented Architecture (SOA). The client layer simply uses the ADF Business Components interfaces in the `oracle.jbo` package. These interfaces provide a higher-level API that lets you think in terms of row sets of rows in the data model whose contents your end-user needs to search for, create, delete, modify and save. They hide all of the lower-level generic SOA-method calling complexity. What's more, when you build UI displays that take advantage of the ADF Model layer for declarative data binding, you generally don't need to write client-side code at all to work with the active data model. Your displays are bound declaratively to view objects in the data model, and to custom business service methods when you need to perform any other kind of logic business service function.

4.5.2 Examples of the Active Data Model In Action

Consider the following three simple, concrete examples of the active data model in action:

New data appears in relevant displays without re-querying

A customer logs into the SRDemo application and sees their list of open service requests. They visit some wizard pages and create a new service request, when they return back to their home page, the new service request appears in their list of open requests without re-querying the database.

Changes caused by business domain layer logic automatically reflected
A manager edits a service request and assigns a technician to the case by picking their name from a poplist list of values page. Business logic encapsulated in the `ServiceRequest` entity object in the business domain layer behind the data model contains a simple rule that updates the assigned date to the current date and time whenever the service request's assigned to attribute is changed. The user interface updates to automatically reflect the assigned date that was changed by the logic in the business domain layer.

Invocation of a business service method re-queries data and sets current rows
In a tree display, the user clicks on a specific node in a tree. This declaratively invokes a business service method on your application module that re-queries master detail information and sets the current rows to an appropriate row in the row set. The display updates to reflect the new master/detail data and current row displayed.

Without an active data model, the developer using a less clever business service implementation approach is forced to write more code in the client to handle the straightforward, everyday CRUD-style operations. In addition, to keep pages up to date, they are forced to manage "refresh flags" that clue the controller layer in to requesting a "repull" of data from the business service to reflect data that might have been modified. When using an ADF application module to implement your business service, you can focus on the business logic at hand, instead of the plumbing to make your business work as your end users expect.

4.5.3 Active Data Model Allows You to Eliminate Most Client-Side Code

Because the application module's active data model feature ensures your client user interface is always up to date, you can typically avoid writing code in the client that is related to setting up or manipulating the data model. Oracle recommends encapsulating any code of this kind inside custom methods of your application module component. Whenever the programmatic code that manipulates view objects is a logical aspect of implementing your complete business service functionality you should encapsulate the details by writing a custom method in your application module's Java class. This includes, but is not limited to, code that:

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

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

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

Another typical type of client-side code you no longer have to write using ADF Business Components is code that coordinates detail data collections when a row in the master changes. By linking the view objects as you'll learn in the next chapter, you can have the coordination performed automatically for you.

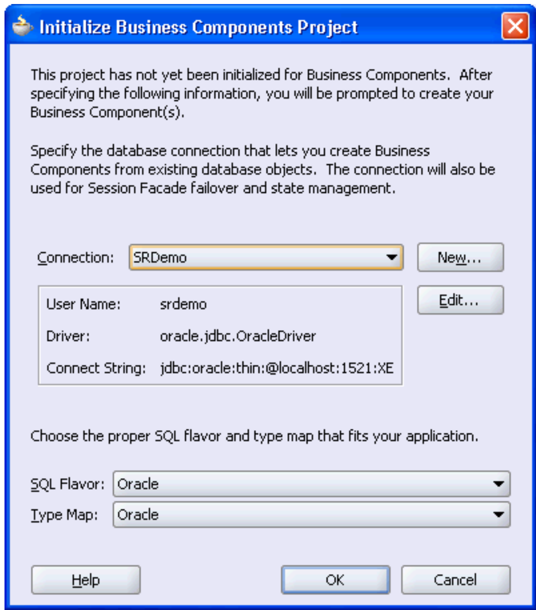
4.6 Overview of ADF Business Components Design Time Facilities

JDeveloper offer broad design time support for ADF Business Components. This section highlights the facilities you'll be using throughout the guide to work with your business components.

4.6.1 Choosing a Connection, SQL Flavor, and Type Map

The first time you create a component, you'll see the Initialize Business Components Project dialog shown in Figure 4–7. You use this dialog to select a design time database connection to work with while working on your business components in this project. The **Connection** dropdown list shows a list of all the named connection definitions you've created, or clicking **New...** allows you to create a new one if you don't see the one you need.

Figure 4–7 Initialize Business Components Project Dialog



The **SQL Flavor** setting controls the syntax of the SQL statements your view objects will use and the syntax of the DML statements your entity objects will use. If JDeveloper detects you are using an Oracle database driver, it defaults this setting to the `Oracle` SQL flavor. The supported SQL flavors include:

- `Oracle` — the default, for working with Oracle
- `OLite` — for the Oracle Lite database
- `SQLServer` — for working with a Microsoft SQLServer database
- `DB2` — for working with an IBM DB2 database
- `SQL92` — for working with any other supported SQL92- compliant database

The **Type Map** setting controls whether you want this project to use the optimized set of Oracle data types, or use only the basic Java data types. If JDeveloper detects you are using an Oracle database driver, it defaults this setting to the `Oracle` Type map. The supported type maps are:

- `Oracle` — use optimized types in the `oracle.jbo.domain` package
- `Java` — use basic Java types only

Note: If you plan to have your application run against both Oracle and non-Oracle databases, you should select the `SQL92` SQL Flavor when you begin building your application, not later. While this makes the application portable to both Oracle and non-Oracle databases, it sacrifices using some of the Oracle-specific optimizations that are inherent in using the Oracle SQL Flavor.

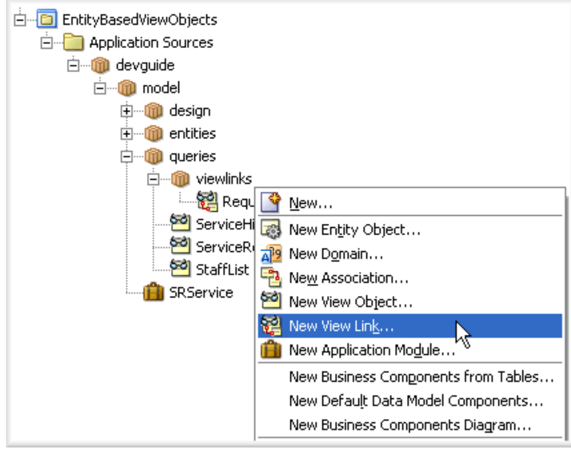
4.6.2 Creating New Components Using Wizards

In the **New Gallery** in the **ADF Business Components** category, JDeveloper offers a wizard to create each kind of business component. Each wizard allows you to specify the component name for the new component and to select the package into which you'd like to organize the component. If the package does not yet exist, the new component becomes the first component in that new package. The wizard presents a series of panels that capture the necessary information to create the component type. When you click **Finish**, JDeveloper creates the new component by saving its XML component definition file. If you have set your Java generation options to prefer their generation by default, JDeveloper also creates the initial custom Java class files.

4.6.3 Quick-Creating New Components Using the Context Menu

Once a package exists in the Application Navigator, you can quickly create additional business components of any type in the package by selecting it in the Application Navigator and using one of the options on the right-mouse context menu as shown in Figure 4–8.

Figure 4–8 Context Menu Options on a Package to Create Any Kind of Business Component



4.6.4 Editing Components Using the Component Editor

Once a component exists, you can edit it using the respective component editor that you access by either double-clicking on the component in the Application Navigator or selecting it and choosing the **Edit** option from the right-mouse context menu. The component editor presents a superset of the panels available in the wizard, and allows you to change any aspect of the component. When you click **OK**, JDeveloper updates the components XML component definition file and if necessary any of its related custom Java files.

4.6.5 Visualizing, Creating, and Editing Components Using UML Diagrams

As highlighted in the walkthrough in Chapter 2, "Overview of Development Process with Oracle ADF and JSF", JDeveloper offers extensive UML diagramming support for ADF Business Components. You can drop existing components you've already created onto a business components diagram to visualize them, use the diagram to create and modify components, or a mixture of the two. The diagrams are kept in sync with changes you make in the editors.

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

4.6.6 Testing Application Modules Using the Business Components Browser

Once you have created an application module component, you can test it interactively using the built-in Business Components Browser. To launch the Business Components Browser, select the application module in the Application Navigator or business components diagram and choose **Test...** from the right-mouse context menu.

This tool presents the view object instances in the application module's data model and allows you to interact with them using a dynamically generated user interface. This tool is invaluable for testing or debugging your business service both before and after you create the view layer of pages or Swing panels.

4.6.7 Refactoring Components

At any time, you can select a component in the Application Navigator and choose **Refactor > Rename** from the right-mouse context menu to rename the component. You can also select one or more components in the navigator — by holding down the [`Ctrl`] key while you select with the mouse click — and choose **Refactor > Move** to move the selected components to a new package. References to the old component names or packages in the current project are adjusted automatically.

5

Querying Data Using View Objects

This chapter describes how to query from the database using SQL queries encapsulated by ADF view objects.

This chapter includes the following sections:

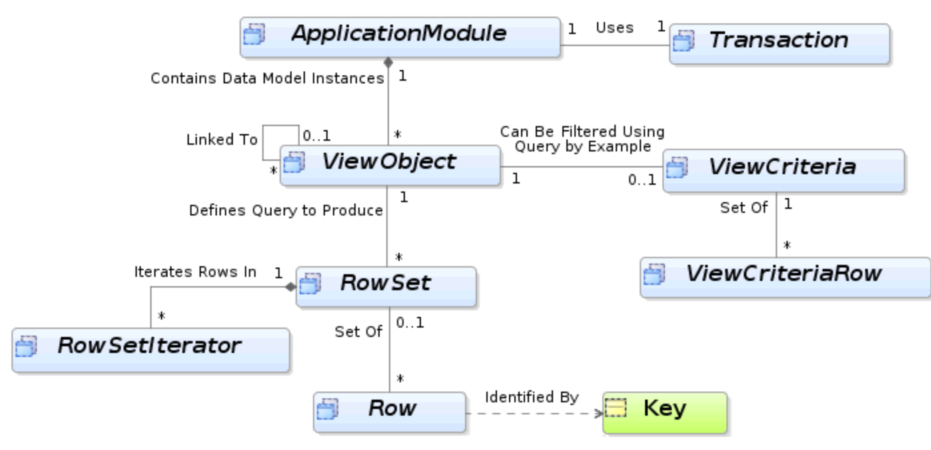
- Section 5.1, "Introduction to View Objects"
- Section 5.2, "Creating a Simple, Read-Only View Object"
- Section 5.3, "Using a View Object in an Application Module's Data Model"
- Section 5.4, "Defining Attribute Control Hints"
- Section 5.5, "Testing View Objects Using the Business Components Browser"
- Section 5.6, "Working Programmatically with View Object Query Results"
- Section 5.7, "How to Create a Command-Line Java Test Client"
- Section 5.8, "Filtering Results Using Query-By-Example View Criteria"
- Section 5.9, "Using Named Bind Variables"
- Section 5.10, "Working with Master/Detail Data"
- Section 5.11, "Generating Custom Java Classes for a View Object"

5.1 Introduction to View Objects

A view object is Oracle ADF component that encapsulates a SQL query and simplifies working with its results. By the end of this chapter, you'll understand all the concepts illustrated in Figure 5–1:

- You define a view object by providing a SQL query
- You use view object instances in the context of an application module that provides the database transaction for their queries
- You can link a view object to one or more others to create master/detail hierarchies
- At runtime, the view object executes your query and produces a row set of rows
- Each row is identified by a corresponding row key
- You iterate through the rows in a row set using a row set iterator
- You can filter the row set a view object produces by applying a set of query-by-example criteria rows

Figure 5–1 A View Object Defines a Query and Produces a RowSet of Rows



Note: To experiment with a working version of the examples in this chapter, download the `DevGuideExamples` workspace from the *Example Downloads* page at `http://otn.oracle.com/documentation/jdev/b25947_01/` and see the `QueryingDataWithViewObjects` project.

5.2 Creating a Simple, Read-Only View Object

View objects can be used for reading data as well as updating data. This chapter focuses on working with read-only data using view objects. In Chapter 7, "Building an Updatable Data Model With Entity-Based View Objects", you'll learn how to create view objects that can handle updating data.

5.2.1 How to Create a Read-Only View Object

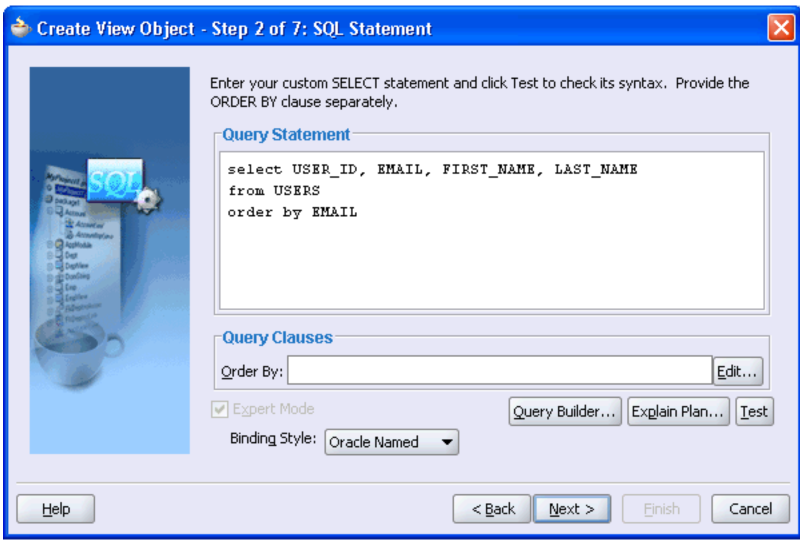
To create a view object, use the **Create View Object wizard**. The wizard is available from the **New Gallery** in the **Business Tier > ADF Business Components** category. If it's the first component you're creating in the project, the **Initialize Business Components Project** dialog appears to allow you to select a database connection. These examples assume that you are working with a connection named `SRDemo` for the `SRDEMO` schema.

As shown in Figure 5–2, provide a package name, a view object name, and indicate that you want this view object to manage data with read-only access. The figure illustrates creating a view object named `Users` in the `devguide.examples` package.

Figure 5–2 Defining the Package and Component Name for a New View Object

In step 2 of the wizard (the **SQL Statement** page), paste in any valid SQL statement into the **Query Statement** box or click **Query Builder** to use the interactive query builder. Figure 5–3 shows a query to retrieve a few columns of user information from the `USERS` table ordered by `EMAIL`.

Figure 5–3 Defining the SQL Query for a Read-Only View Object



Note: If you see an **Entity Objects** page instead of the **SQL Statement** page shown here, go back to step 1 and ensure that you've selected **Read-only Access**.

Since the query does not reference any bind variables, you can skip step 3 **Bind Variables** page for now. In Section 5.9, "Using Named Bind Variables", you'll add a bind variable and see how to work with it in the query.

In addition to the SQL query information, a view object captures information about the names and datatypes of each expression in its query's SELECT list. As you'll see in Section 5.6, "Working Programmatically with View Object Query Results", you can use these view object attribute names to access the data from any row in the view object's result set by name. You *could* directly use the SQL column names as attribute names in Java, but typically the SQL names are all uppercase and often comprised of multiple, underscore-separated words. The view object wizard converts these SQL-friendly names to Java-friendly ones.

In step 4 on the **Attribute Mappings** page, as shown in Figure 5–4 you can see how the SELECT list column names correspond to the more Java-friendly view object attribute names that the wizard has created by default. Each part of an underscore-separated column name like *`SOME_COLUMN_NAME`* is turned into a capitalized word in the attribute name like *`SomeColumnName`*. While the view object attribute names correspond to the underlying query columns in the SELECT list, the attribute names at the view object level need not match necessarily. You can later rename the view object attributes to any names that might be more appropriate without changing the underlying query.

Note: You'll see throughout the ADF Business Components wizards and editors, that the default convention is to use "CamelCapped" attribute names, beginning with a capital letter and using upper-case letters in the middle of the name to improve readability when the name comprises multiple words.

Figure 5–4 Wizard Creates Default Java-Friendly Attribute Names for Each Column in Select List

Click **Finish** at this point to create the view object.

5.2.2 What Happens When You Create a Read-Only View Object

When you create a view object, JDeveloper first describes the query to infer the following from the columns in the SELECT list:

- The Java-friendly view attribute names (e.g. `USER_ID` -> `UserId`)
- The SQL and Java data types of each attribute

JDeveloper then creates the XML component definition file that represents the view objects's declarative settings and saves it in the directory that corresponds to the name of its package. In the example above, the view object was named `Users` in the `devguide.examples` package, so that the XML file created will be `./devguide/examples/Users.xml` under the project's source path. This XML file contains the SQL query you entered, along with the names, datatypes, and other properties of each attribute. If you're curious to see its contents, you can see the XML file for the view object by selecting the view object in the **Application Navigator** and looking in the corresponding **Sources** folder in the Structure window. Double-clicking the `Users.xml` node will open the XML in an editor so that you can inspect it.

Note: If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom view object class `UsersImpl.java` and/or a custom view row class `UsersRowImpl.java` class.

5.2.3 What You May Need to Know About View Objects

Typically you create one view object for each SQL query your application will perform.

5.2.3.1 Editing an Existing View Object Definition

After you've created a view object, you can edit any of its settings by using the View Object Editor. Choose the **Edit** menu option on the context menu in the Application Navigator, or double-click the view object, to launch the dialog. By opening the different panels of the editor, you can adjust the SQL query, change the attribute names, add named bind variables, add UI controls hints, control Java generation options, and other settings that are described in later chapters.

5.2.3.2 Working with Queries That Include SQL Expressions

If your SQL query includes a calculated expression like this:

```
select USER_ID, EMAIL,
       SUBSTR(FIRST_NAME,1,1)||'. '||LAST_NAME
from USERS
order by EMAIL
```

use a SQL alias to assist the Create View Object wizard in naming the column with a Java-friendly name:

```
select USER_ID, EMAIL,
       SUBSTR(FIRST_NAME,1,1)||'. '||LAST_NAME AS USER_SHORT_NAME
from USERS
order by EMAIL
```

5.2.3.3 Controlling the Length, Precision, and Scale of View Object Attributes

As shown in Figure 5–5, by selecting a particular attribute name in the View Object Editor, you can see and change the values of the declarative settings that control its runtime behavior. One important property is the **Type** in the **Query Column** section. This property records the SQL type of the column, including the length information for `VARCHAR2` columns and the precision/scale information for NUMBER columns. The JDeveloper editors try to infer the type of the column automatically, but for some SQL expressions the inferred value might be `VARCHAR2(255)`. You can update the **Type** value for such attributes to reflect the correct length if you know it. For example, `VARCHAR2(30)` which shows as the **Type** for the `FirstName` attribute in Figure 5–5 means that it has a maximum length of 30 characters. For a `NUMBER` column, you would indicate a **Type** of `NUMBER(7,2)` for an attribute that you want to have a precision of 7 digits and a scale of 2 digits after the decimal.

Figure 5–5 Editing the Settings for a View Object Attribute

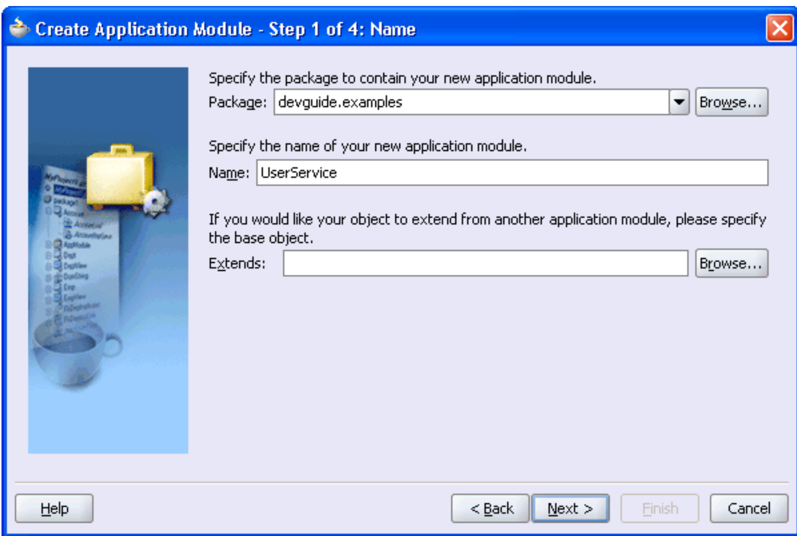
5.3 Using a View Object in an Application Module's Data Model

Any view object you create is a reusable component that can be used in the context of one or more application modules to perform the query it encapsulates in the context of that application module's transaction. The set of view objects used by an application module defines its data model, in other words, the set of data that a client can display and manipulate through a user interface. To start simple, create an application module and use the single view object you created above in the application module's data model.

5.3.1 How to Create an Application Module

To create an application module, use the **Create Application Module wizard**. The wizard is available from the **New Gallery** in the **Business Tier > ADF Business Components** category. As shown in Figure 5–6, provide a package name, and an application module name. The figure shows creating an application module `UserService` in the `devguide.examples` package.

Figure 5–6 Defining the Package and Component Name for a New Application Module



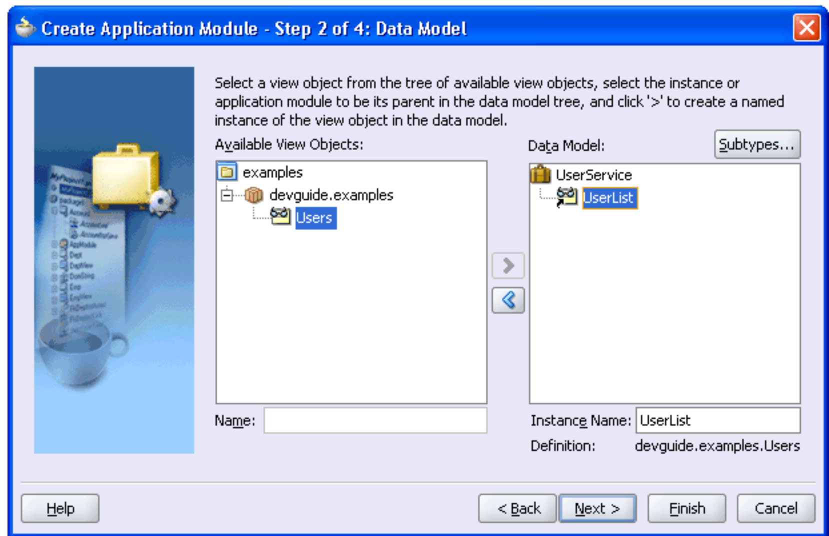
In step 2 on the **Data Model** page, Figure 5–7 illustrates that initially the data model is empty. That is, it contains no view object instances yet. The **Available View Objects** list shows all available view objects in your project, organized by package.

Figure 5–7 Defining the Data Model For a New Application Module

To add an instance of a view object to the data model, first select it in the **Available** list. The **Name** field below the list shows the name that will be used to identify the next instance of that view object that you add to the data model. By typing in a different name before pressing the add instance button **>**, you can change the name to be anything you like. Finally, to add an instance of the selected view object to the data model, identified by the instance name shown below, click the add instance button (>).

Assuming you decide on the instance name of `UserList`, Figure 5–8 shows what the **Data Model** page would look like after adding the instance. The **Instance Name** field below the selected view object instance in the **Data Model** list allows you to see and change the instance name if necessary. The **Definition** field displays the fully-qualified name of the view object component that will be used to create this view object instance at runtime. You see as expected that the definition that will be used is `devguide.examples.Users` view object.

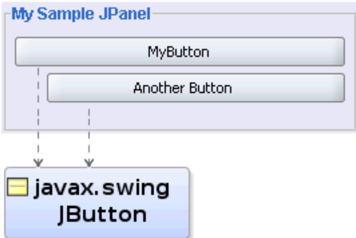
Figure 5–8 Data Model With One Instance Named UserList of the Users View Object



5.3.1.1 Understanding the Difference Between View Object Components and View Object Instances

It is important to understand the distinction between a view object component and a view object instance. The easiest way to understand the distinction is to first consider a visual example. Imagine that you need to build a Java user interface containing two buttons. Using JDeveloper's visual editor, you might create the page shown in Figure 5–9 by using the **Component Palette** to select the `JButton` component and click to add a JButton component to your panel. Repeating that same step a second time, you can drop another button onto the panel. You are designing a custom `JPanel` component that uses two *instances* of the `JButton` component. The panel does not own the `JButton` class, it's just *using* two instances of it.

Figure 5–9 Designing a Panel That Contains Two Instances of the JButton Component



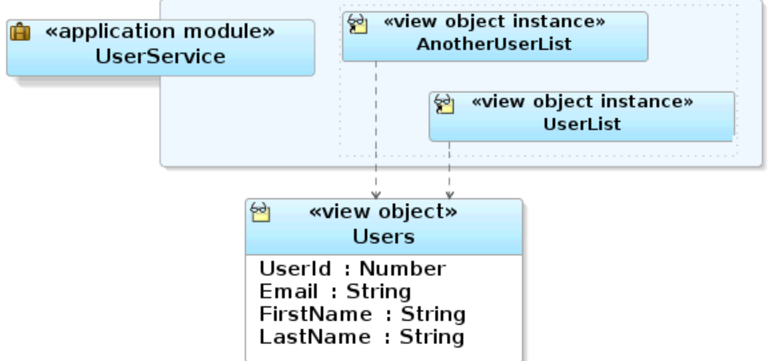
If you were to peek into the Java code of this new panel you're designing, you'd notice that there are two member fields of type `JButton` to hold a reference to the two instances of the button the panel is using. To distinguish the two instances in the code, one member field is named `myButton`, and the other member field is named `anotherButton`:

Example 5–1 Two Instances of a JButton Component

```
public class Panel1 extends JPanel implements JUPanel {
  private JButton myButton      = new JButton(); // First instance
  private JButton anotherButton = new JButton(); // Second instance
  // etc.
}
```

Even though the application module is a nonvisual component, you can still apply this same intuition about components, instances, and distinct member names to help understand the concept better. While designing an application module, you use instances of a view object component to define its data model. Figure 5–10 shows a JDeveloper business components diagram of a `UserService` application module. Just as the panel in Example 5–1 contained two instances of the `JButton` component with member names of `myButton` and `anotherButton` to distinguish them, your application module contains two instances of the `Users` view object component, with member names of `UserList` and `AnotherUserList` to distinguish them. At runtime, the two `JButton` instances are both based on the same definition — which explains why they both have the same set of properties and both exhibit `JButton`-like behavior. However the values of their properties like `Position` and `Text` are different. So too for the different instances of the `Users` view object in your `UserService` application module. At runtime, both instances share the same `Users` view object component definition — ensuring they have the same attribute structure and Users view object behavior — however, each might be used independently to retrieve data about different users. For example, some of the runtime properties like an additional filtering WHERE clause or the value of a bind variable might be different on the two distinct instances.

Figure 5–10 Designing an Application Module That Contains Two Instances of the Users View Object Component



Besides the obvious fact that one example is a visual panel while the other is a nonvisual data model component, the only logical difference is how the instances and member names are defined. In the visual panel in Example 5–1, you saw that the two member fields holding the distinct `JButton` instances were declared in code. In contrast, the `UserService` application module defines its member view object instances in its XML component definition file:

Example 5–2 Application Modules Define Member View Objects in XML

```
<AppModule Name="UserService">
   <ViewUsage Name="UserList" ViewObjectName="devguide.examples.Users"/>
   <ViewUsage Name="AnotherUserList" ViewObjectName="devguide.examples.Users"/>
</AppModule>
```

5.3.2 What Happens When You Create an Application Module

When you create an application module, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. In the example in Figure 5–6, the application module was named `UserService` in the `devguide.examples` package, so the XML file created will be `./devguide/examples/UserService.xml` under the project's source path. This XML file contains the information needed at runtime to recreate the view object instances in the application module's data model. If you're curious to see its contents, you can see the XML file for the application module by selecting the view object in the **Application Navigator** and looking in the corresponding **Sources** folder in the Structure window. Double-clicking the `UsersService.xml` node will open the XML in an editor so that you can inspect it.

Note: If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom application module class `UsersServiceImpl.java`.

5.3.3 What You May Need to Know About Application Modules

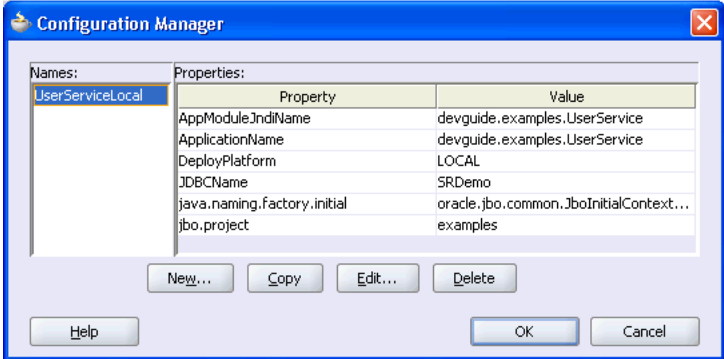
After you've created an application module, you can edit any of its settings by using the Application Module Editor. Select the **Edit** menu option on the context menu in the **Application Navigator** to launch the application module. By opening the different panels of the editor, you can adjust the view object instances in the data model, control Java generation options, and other settings you'll learn about in later chapters.

5.3.3.1 Editing an Application Module's Runtime Configuration Properties

Since it can be convenient to define and use multiple sets of runtime configuration properties each application module supports multiple, named runtime configurations. When you create an application module, JDeveloper creates a *default* set of runtime configuration properties for the application module. For an application module named `YourService`, its default set of configuration properties will be named `YourServiceLocal`. These settings are stored in an XML file named `bc4j.xcfg` in a subdirectory named `common`, relative to where the application module's XML component definition resides. For example, when you created the `UserService` application module above in the `devguide.examples` package, JDeveloper creates the file `bc4j.xcfg` in the `./devguide/examples/common` directory under the project's source path.

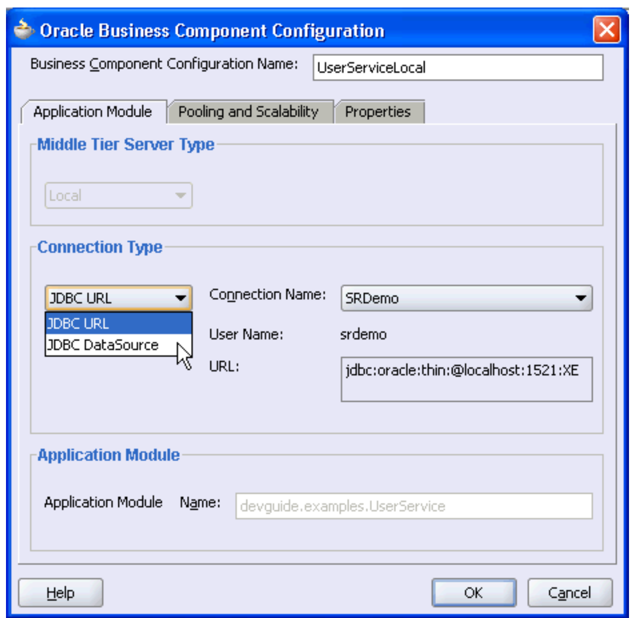
You can use the application module configuration manager to edit existing configurations or create new ones. To access the configuration manager, select the desired application module in the **Application Navigator** and choose **Configurations...**from the context menu. The **Configuration Manager** dialog appears, as shown in Figure 5–11. You can see the default `UserServiceLocal` configuration for the `UserService` application module. Any additional configurations you create, or any configuration properties you edit, are saved in the same `bc4j.xcfg`.

Figure 5–11 Application Module Configuration Manager



Click the **Edit** button in the **Configuration Manager** to edit a specific configuration. As shown in Figure 5–12, this editor allows you to configure the database connection information, a number of pooling and scalability settings, and a final tab of remaining properties that aren't covered by the first two tabs. All of the runtime properties and their meanings are covered in the JDeveloper online help.

Figure 5–12 Oracle Business Component Configuration Editor



Note: When building web applications, set the `jbo.locking.mode` property to `optimistic`. The default value is `pessimistic`, which is not the correct value for web applications. You can find this property listed alphabetically in the **Properties** tab of the **Configuration Editor**.

5.4 Defining Attribute Control Hints

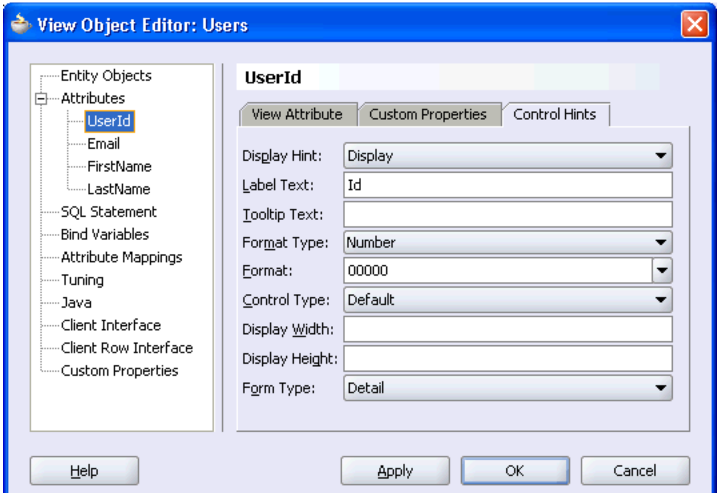
One of the many powerful, built-in features of the ADF Business Components is the ability to define control hints on attributes. Control hints are additional attribute settings that the view layer can use to automatically display the queried information to the user in a consistent, locale-sensitive way. JDeveloper manages storing the hints in a way that is easy to localize for multi-lingual applications.

5.4.1 How to Add Attribute Control Hints

To add attribute control hints for the attributes of the `UserList` view object, open the View Object Editor and expand the **Attributes** node in the left-side tree to reveal the list of the view object's attributes. As shown in Figure 5–13, by selecting a particular attribute name like `UserId` and selecting the **Control Hints** tab, you can enter a value for its **Label Text** hint like "Id". You can also set the **Format Type** to Number, and enter a **Format** mask of `00000`. You could select the other attributes in turn to define **Label Text** hints like "Email Address", "Given Name", and "Surname" for the `Email`, `FirstName`, and `LastName` attributes respectively.

Note: Java defines a standard set of format masks for numbers and dates that are different from those used by the Oracle database's SQL and PL/SQL languages. For reference, see the Javadoc for the `java.text.DecimalFormat` and `java.text.SimpleDateFormat` classes.

Figure 5–13 Setting UI Control Hints for View Object Attributes



5.4.2 What Happens When You Add Attribute Control Hints

When you define attribute control hints for a view object, JDeveloper creates a standard Java message bundle file in which to store them. The file is specific to the view object component to which it's related, and it is named accordingly. For the `UserList` view object in the `devguide.examples` package, the message bundle file created will be named `UserListRowImplMsgBundle.java` and it will be created in the `devguide.examples.common` subpackage. By selecting the `UserList` component in the **Application Navigator**, you'll see that this new file gets added to the **Sources** folder in the **Structure window** that shows the group of implementation files for each business component. Example 5–3 shows how the control hint information appears in the message bundle file. The first entry in each String array is a message key; the second entry is the locale-specific `String` value corresponding to that key.

Example 5–3 View Object Component Message Bundle Class Stores Locale-Sensitive Control Hints

```
package devguide.examples.common;
import oracle.jbo.common.JboResourceBundle;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---------------------------------------------------------------------
public class UsersRowImplMsgBundle extends JboResourceBundle {
  static final Object[][] sMessageStrings =
  {
    { "UserId_LABEL", "Id" },
    { "UserId_FMT_FORMATTER", "oracle.jbo.format.DefaultNumberFormatter" },
    { "UserId_FMT_FORMAT", "00000" },
    { "Email_LABEL", "Email Address" },
    { "FirstName_LABEL", "Given Name" },
    { "LastName_LABEL", "Surname" }
  };
```

5.4.3 What You May Need to Know About Message Bundles

Internationalizing the model layer of an application built using ADF Business Components entails producing translated versions of each component message bundle file. For example, the Italian version of the `UsersRowImplMsgBundle` message bundle would be a class named `UsersRowImplMsgBundle_it`, and a more specific *Swiss* Italian version would have the name `UsersRowImplMsgBundle_it_ch`. These classes typically extend the base message bundle class, and contain entries for the message keys that need to be localized, together with their localized translation. For example, assuming you didn't want to translate the number format mask for the Italian locale, the Italian version of the `UserList` view object message bundle would look like what you see in Example 5–4. Notice the overridden `getContents()` method. It returns an array of messages with the more *specific* translated strings merged together with those that are not overridden from the superclass bundle. At runtime, the appropriate message bundles are used automatically, based on the current user's locale settings.

Example 5–4 Localized View Object Component Message Bundle for Italian

```
package devguide.examples.common;
import oracle.jbo.common.JboResourceBundle;
public class UsersRowImplMsgBundle_it extends UsersRowImplMsgBundle {
  static final Object[][] sMessageStrings =
  {
    { "UserId_LABEL", "Codice Utente" },
    { "Email_LABEL", "Indirizzo Email" },
    { "FirstName_LABEL", "Nome" },
    { "LastName_LABEL", "Cognome" }
  };
  // merge this message bundles messages with those in superclass bundle
  public Object[][] getContents() {
    return super.getMergedArray(sMessageStrings, super.getContents());
  }
}
```

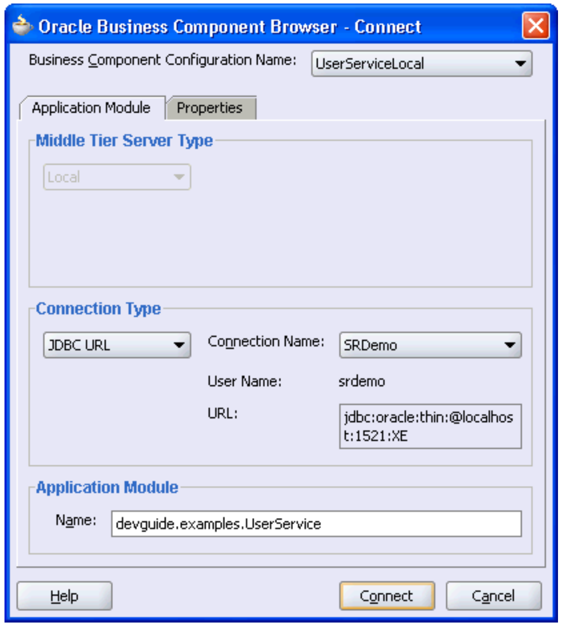
5.5 Testing View Objects Using the Business Components Browser

JDeveloper includes an interactive application module testing tool that enables you to to test all aspects of its data model without having to use your application user interface or write a test client program. It can often be the quickest way of exercising the data functionality of your business service during development.

5.5.1 How to Test a View Object Using the Business Components Browser

To test an application module, select it in the **Application Navigator** and choose **Test** from the context menu. The **Business Component Browser Connect** dialog appears as shown in Figure 5–14. In the upper right corner of the dialog, the **Configuration Name** list allows you to choose any of your application module's configurations for the current run of the tester tool. Click **Connect** to start the application module using the selected configuration.

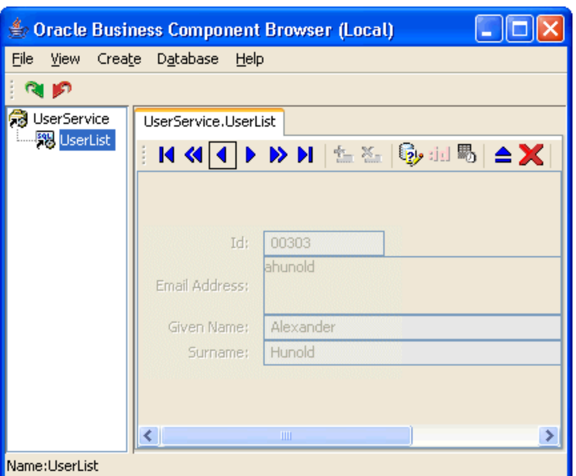
Figure 5–14 Business Component Browser



5.5.2 What Happens When You Use the Business Components Browser

When you launch the Business Components Browser, JDeveloper starts the tester tool in a separate process and the Business Components Browser appears. The tree at the left of the dialog displays all of the view object instances in your application module's data model. Figure 5–15 has only one instance called `UserList`. Double-clicking the `UserList` view object instance in the tree executes the view object — if it has not been executed so far in the testing session — and displays a panel to inspect the query results as shown in Figure 5–15. Additional context menu items on the view object node allow you to re-execute the query if needed, to remove the tester panel, and to perform other tasks.

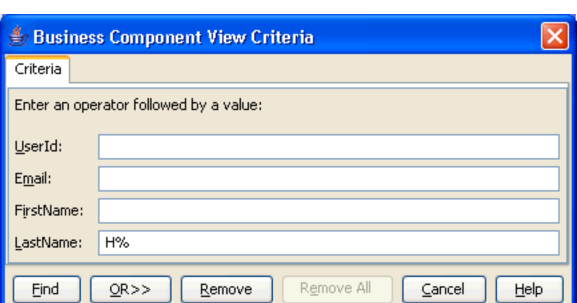
Figure 5–15 Testing the Data Model in the Business Component Browser



You can see that the fields in the test panel for the view object are disabled for the UserList view object instance, because the view object is read only. In Section 7.5, "Testing Entity-Based View Objects Interactively", you'll see that the tester tool becomes even more valuable by allowing you to experiment with inserting, updating, and deleting rows in view objects, too. But even for a read-only view object, the tool affords some useful features. Firstly, you can validate that the UI hints and format masks are defined correctly. The attributes display with their defined **Label Text** control hints as the prompt, and the UserId field is displayed as 00303 due to the 00000 format mask defined in Section 5.4.1, "How to Add Attribute Control Hints". Secondly, you can scroll through the data using the toolbar buttons.

Thirdly, you can enter query-by-example criteria to find a particular row whose data you want to inspect. By clicking the **Specify View Criteria** button in the toolbar, the **View Criteria** dialog displays as shown in Figure 5–16. You can enter a query criteria like "H%" in the LastName attribute and click **Find** to narrow the search to only those users with a last name that begins with the letter H (Hunold, Himuro, Hartstein, and Higgins).

Figure 5–16 Exercising Built-in Query-by-Example Functionality



5.5.3 What You May Need to Know About the Business Components Browser

When using the Business Components Browser you can customize configuration options for the current run. You can also enable ADF Business Component debug diagnostics to output messages to the console. Both of these features can help you test various portions of your application or find problems.

5.5.3.1 Customizing Configuration Options for the Current Run

As described in Figure 5–14, on the **Connect** dialog of the Business Component Browser you can select a predefined configuration to run the tool using that named set of runtime configuration properties. The **Connect** dialog also features a **Properties** tab that allows you to see the selected configurations settings and to override any of the configuration's settings for the current run of the browser. For example, you could test the Italian language translations of the UI control hints for a single Business Components Browser run by opening the **Properties** tab and setting the following two properties:

- jbo.default.country = IT
- jbo.default.language = it

If you wanted to make the changes permanent, you could use the **Configuration Manager** to copy the current `UserServiceLocal` configuration and create a new `UserServiceLocalItalian` which had these two additional properties set. This way, anytime you wanted to test in Italian you could simply choose to use the `UserServiceLocalItalian` configuration instead of the default `UserServiceLocal` one.

5.5.3.2 Enabling ADF Business Components Debug Diagnostics

When launching the Business Components Browser, if your project's current run configuration is set to include the Java System parameter `jbo.debugoutput=console`, you can enable ADF Business Components debug diagnostics with messages directed to the console. These will display in the JDeveloper Log window.

> **Note:** Despite the similar name, the JDeveloper project's run configurations are different from the ADF application module's configurations. The former are part of the project properties, the latter are defined along with your application module component in its `bc4j.xcfg` file and edited using the configuration editor.

To set the system property described above, open the **Run/Debug** page in the **Project Properties** dialog for your model project. Click **Edit** to edit the chosen run configuration, and add the string:

```
-Djbo.debugoutput=console
```

to the **Java Options** field in the panel. The next time you run the Business Component Browser and double-click on the `UserList` view object, you'll see detailed diagnostic output in the console.

Example 5–5 Diagnostic Output of Business Component Browser

```
  :
[234] Created root application module: 'devguide.examples.UserService'
[235] Stringmanager using default locale: 'en_US'
[236] Locale is: 'en_US'
[237] ApplicationPoolImpl.resourceStateChanged wasn't release related.
[238] Oracle SQLBuilder: Registered driver: oracle.jdbc.driver.OracleDriver
[239] Creating a new pool resource
[240] Trying connection/2: url='jdbc:oracle:thin:@localhost:1521:XE' ...
[241] Successfully logged in
[242] JDBCDriverVersion: 10.1.0.5.0
[243] DatabaseProductName: Oracle
[244] DatabaseProductVersion: Oracle Database 10g Release 10.2.0.1.0
[245] Column count: 4
[246] ViewObject: UserList Created new QUERY statement
[247] UserList>#q computed SQLStmtBufLen: 110, actual=70, storing=100
[248] select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
order by EMAIL
  :
```

Using the diagnostics, you can see everything the framework components are doing for your application.

Other legal values for this property are `silent` (the default, if not specified) and `file`. If you choose the file option, diagnostics are written to the system temp directory. One best practice is to create multiple JDeveloper run configurations, one with the ADF Business Components debug diagnostics set on, and another without it, so you can easily flip between seeing and not seeing debug diagnostics by choosing the appropriate project run configuration.

5.6 Working Programmatically with View Object Query Results

Now that you have a working application module containing an instance named `UserList`, you can build a simple test client program to illustrate the basics of working programmatically with the data in the `UserList` view object instance.

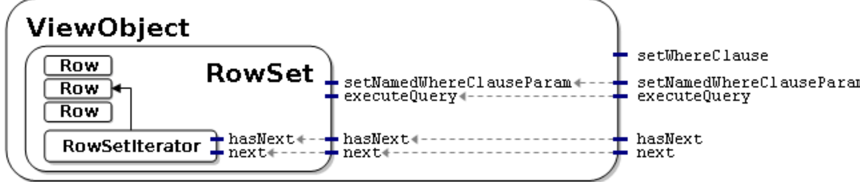
5.6.1 Common Methods for Working with the View Object's Default RowSet

The `ViewObject` interface in the `oracle.jbo` package provides the methods to make quick work of any data-retrieval task. Some of these methods used in the example include:

- `executeQuery()`, to execute the view object's query and populate its row set of results
- `setWhereClause()`, to add a dynamic predicate at runtime to narrow a search
- `setNamedWhereClauseParam()`, to set the value of a named bind variable
- `hasNext()`, to test whether the row set iterator has reached the last row of results
- `next()`, to advance the row set iterator to the next row in the row set
- `getEstimatedRowCount()`, to count the number of rows a view object's query would return

Chapter 27, "Advanced View Object Techniques" presents situations when you might want a single view object to produce *multiple* distinct row sets of results; however, most of the time you'll work only with a single row set of results at a time for a view object. That same later chapter, also describes scenarios when you might want to create *multiple* distinct row set iterators for a row set, however again most of the time you'll need only a single iterator. To simplify this overwhelmingly common use case, as shown in Figure 5–17, the view object contains a default `RowSet`, which, in turn, contains a default `RowSetIterator`. As you'll see in the examples below, the default `RowSetIterator` allows you to call all of the methods above directly on the `ViewObject` component itself, knowing that they will apply automatically to its default row set.

Figure 5–17 ViewObject Contains a Default RowSet and RowSetIterator



Note: Throughout this guide, whenever you encounter the phrase "working with the rows in a view object," what this means more precisely is working with the rows in the view object's default row set. Similarly, when you read "iterate over the rows in a view object," what this means more precisely is that you'll use the default row set iterator of the view object's default row set to loop over its rows.

With the concepts in place, you can create a test client program to put them into practice.

5.6.2 Counting the Number of Rows in a RowSet

The `getEstimatedRowCount()` method is used on a `RowSet` to determine how many rows it contains:

```
long numReqs = reqs.getEstimatedRowCount();
```

The implementation of the `getEstimatedRowCount()` initially issues a SELECT COUNT(*) query to calculate the number of rows that the query will return. The query is formulated by "wrapping" your view object's entire query in a statement like:

```
SELECT COUNT(*) FROM ( ... your view object's SQL query here ... )
```

This approach allows you to access the count of rows for a view object without necessarily retrieving all the rows themselves which is an important optimization for working with queries that return a large number of rows, or proactively testing how many rows a query *would* return before proceeding to work with the results of the query.

Once the estimated row count is calculated, subsequent calls to the method do not re-execute the `COUNT(*)` query. The value is cached until the next time the view object's query is executed, since the fresh query result set returned from the database could potentially contain more, fewer, or different rows compared with the last time the query was run. The estimated row count is automatically adjusted to account for pending changes in the current transaction, adding the number of relevant new rows and subtracting the number of removed rows from the count returned.

5.7 How to Create a Command-Line Java Test Client

To create a test client program, create a new Java class using the **Create Java Class wizard**. This is available in the **New Gallery** under the **General** category. Enter a class name like `TestClient`, a package name like `devguide.examples.client`, and ensure the **Extends** field says `java.lang.Object`. In the **Optional Attributes**, deselect the **Generate Default Constructor** and select the **Generate Main Method** checkbox. Then click **OK** to create the `TestClient.java` file. The file opens in the source editor to show you the skeleton code:

Example 5–6 Skeleton Code for TestClient.java

```
package devguide.examples.client;
public class TestClient {
  public static void main(String[] args) {

  }
}
```

Place the cursor on a blank line inside the body of the main() method and use the bc4jclient code template to create the few lines of necessary code. To use this predefined code template, type the characters bc4jclient followed by a [Ctrl]+[Enter] to expands the code template so that the class now should look like this:

Example 5–7 Expanded Skeleton Code for TestClient.java

```
package devguide.examples.client;
import oracle.jbo.client.Configuration;
import oracle.jbo.*;
import oracle.jbo.domain.Number;
import oracle.jbo.domain.*;
public class TestClient {
  public static void main(String[] args) {
    String        amDef = "test.TestModule";
    String        config = "TestModuleLocal";
    ApplicationModule am =
      Configuration.createRootApplicationModule(amDef,config);
    ViewObject vo = am.findViewObject("TestView");
    // Work with your appmodule and view object here
    Configuration.releaseRootApplicationModule(am,true);
  }
}
```

Adjust the values of the amDef andconfig variables to reflect the names of the application module definition and the configuration that you want to use, respectively. For the Example 5–7, you would change these two lines to read:

```
String amDef = "devguide.examples.UserService";
String config = "UserServiceLocal";
```

Finally, change view object instance name in the call to findViewObject() to be the one you want to work with. Specify the name exactly as it appears in the **Data Model** tree on the **Data Model** panel of the Application Module editor. Here, the view object instance is named UserList, so you need to change the line to:

```
ViewObject vo = am.findViewObject("UserList");
```

At this point, you have a working skeleton test client for the UserService application module whose source code looks like what you see in Example 5–8.

Note: Section 8.5, "Working Programmatically with an Application Module's Client Interface" expands this test client sample code to illustrate calling custom application module service methods, too.

Example 5–8 Working Skeleton Code for an Application Module Test Client Program

```
package devguide.examples.client;
import oracle.jbo.client.Configuration;
import oracle.jbo.*;
import oracle.jbo.domain.Number;
import oracle.jbo.domain.*;
public class TestClient {
  public static void main(String[] args) {
    String        amDef = "devguide.examples.UserService";
    String        config = "UserServiceLocal";
    ApplicationModule am =
      Configuration.createRootApplicationModule(amDef,config);
    ViewObject vo = am.findViewObject("UserList");
    // Work with your appmodule and view object here
    Configuration.releaseRootApplicationModule(am,true);
  }
}
```

To execute the view object's query, display the number of rows it will return, and loop over the result to fetch the data and print it out to the console, replace `// Work with your appmodule and view object here`, with the code in Example 5–9

Example 5–9 Looping Over a View Object and Printing the Results to the Console

```
System.out.println("Query will return "+
   vo.getEstimatedRowCount()+" rows...");
vo.executeQuery();
while (vo.hasNext()) {
  Row curUser = vo.next();
  System.out.println(vo.getCurrentRowIndex()+". "+
                     curUser.getAttribute("UserId")+" "+
                     curUser.getAttribute("Email"));
}
```

The first line calls `getEstimatedRowCount()` to show how many rows the query will retrieve. The next line calls the `executeQuery()` method to execute the view object's query. This produces a row set of zero or more rows that you can loop over using a `while` statement that iterates until the view object's `hasNext()` method returns `false`. Inside the loop, the code puts the current `Row` in a variable named `curUser`, then invokes the `getAttribute()` method twice on that current `Row` object to get the value of the `UserId` and `Email` attributes to print them to the console.

5.7.1 What Happens When You Run a Test Client Program

When you run the `TestClient` class by choosing **Run** from the context menu of the source editor, you'll see the results of the test in the log window. Notice that the `getCurrentRowIndex()` used in Example 5–10 shows that the row index in a row set is a zero-based count of the rows:

Example 5–10 Log Output from Running a Test Client

```
Query will return 27 rows...
0. 303 ahunold
1. 315 akhoo
:
25. 306 vpatabal
26. 326 wgietz
```

The call to `createRootApplicationModule()` on the `Configuration` object returns an instance of the `UserService` application module to work with. As you might have noticed in the debug diagnostic output, the ADF Business Components runtime classes load XML component definitions as necessary to instantiate the application module and the instance of the view object component that you've defined in its data model at design time. The `findViewObject()` method on the application module finds a view object instance by name from the application module's data model. After the loop described in Example 5–9, the call to `releaseRootApplicationModule()` on the `Configuration` object signals that you're done using the application module and allows the framework to clean up resources, like the database connection that was used by the application module.

5.7.2 What You May Need to Know About Running a Test Client

The `createRootApplicationModule()` and `releaseRootApplicationModule()` methods are very useful for command-line access to application module components, however you won't typically ever need to write these two lines of code in the context of an ADF-based web or Swing application. The ADF Model data binding layer cooperates automatically with the ADF Business Components layer to acquire and release application module components for you in those scenarios.

5.8 Filtering Results Using Query-By-Example View Criteria

When you need to filter the query results that a view object produces based on search criteria provided at runtime by the end user, you can apply a `ViewCriteria` to the view object. The view criteria is a row set of one or more view criteria rows, whose attributes mirror those in the view object. The key difference between a view row of query results and a view criteria row is that the data type of each attribute in the view criteria row is `String` to allow query-by-example operators to be entered like "`> 304`", for example.

5.8.1 How to Use View Criteria to Filter View Object Results

To use a view criteria, follow the steps illustrated in the `TestClientViewCriteria` class in Example 5–11 to call:

1. `createViewCriteria()` on the view object, to be filtered to create an empty view criteria row set
2. `createViewCriteriaRow()` on the view criteria, to create one or more empty view criteria rows
3. `setAttribute()` as appropriate on the view criteria rows, to set attribute values to filter on
4. `add()` on the view criteria, to add the view criteria rows to the view criteria row set

5. `applyViewCriteria()`, to apply the view criteria to the view object
6. `executeQuery()` on the view criteria, to execute the query with the applied filter criteria

The last step to execute the query is important since a newly applied view criteria is only applied to the view object's SQL query at its next execution.

Example 5–11 Creating and Applying a View Criteria

```
package devguide.examples.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewCriteria;
import oracle.jbo.ViewCriteriaRow;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
public class TestClientViewCriteria {
  public static void main(String[] args) {
    String amDef = "devguide.examples.UserService";
    String config = "UserServiceLocal";
    ApplicationModule am =
     Configuration.createRootApplicationModule(amDef, config);
    ViewObject vo = am.findViewObject("UserList");
    // 1. Create a view criteria rowset for this view object
    ViewCriteria vc = vo.createViewCriteria();
    // 2. Use the view criteria to create one or more view criteria rows
    ViewCriteriaRow vcr1 = vc.createViewCriteriaRow();
    ViewCriteriaRow vcr2 = vc.createViewCriteriaRow();
    // 3. Set attribute values to filter on in appropriate view criteria rows
    vcr1.setAttribute("UserId","> 304");
    vcr1.setAttribute("Email","d%");
    vcr1.setAttribute("UserRole","technician");
    vcr2.setAttribute("UserId","IN (324,326)");
    vcr2.setAttribute("LastName","Baer");
    // 4. Add the view criteria rows to the view critera rowset
    vc.add(vcr1);
    vc.add(vcr2);
    // 5. Apply the view criteria to the view object
    vo.applyViewCriteria(vc);
    // 6. Execute the query
    vo.executeQuery();
    while (vo.hasNext()) {
      Row curUser = vo.next();
      System.out.println(curUser.getAttribute("UserId") + " " +
                         curUser.getAttribute("Email"));
    }
    Configuration.releaseRootApplicationModule(am, true);
  }
}
```

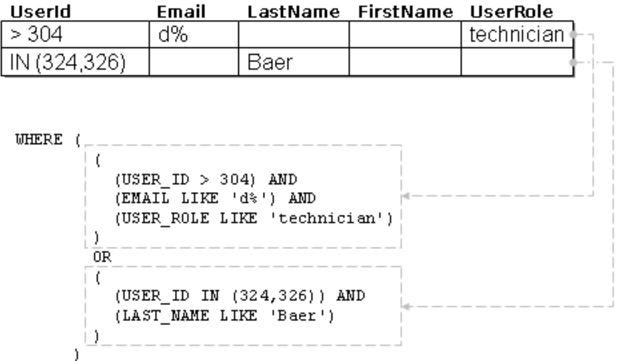
Running the `TestClientViewCriteria` example in Example 5–11 produces the output:

```
305 daustin
307 dlorentz
324 hbaer
```

5.8.2 What Happens When You Use View Criteria to Filter View Object Results

When you apply a view criteria containing one or more view criteria rows to a view object, the next time it is executed it augments its SQL query with an additional WHERE clause predicate corresponding the query-by-example criteria that you've populated in the view criteria rows. As shown in Figure 5–18, when you apply a view criteria containing multiple view criteria rows, the view object augments its design time WHERE clause by adding an additional runtime WHERE clause based on the non-`null` example criteria attributes in each view criteria row.

Figure 5–18 View Object Automatically Translates View Criteria Rows into Additional Runtime WHERE Filter



5.8.3 What You May Need to Know About Query-By-Example Criteria

There are several things you may need to know about query-by-example criteria, including how to test view criteria in the Business Components Browser, altering compound search conditions using multiple view criteria rows, searching for a row whose attribute value is NULL, searching case insensitively, clearing view criteria in effect, and how applying view criteria causes a query to be re-parsed.

5.8.3.1 Use Attribute Names in View Criteria, Column Names in WHERE Clause

In Section 5.6.1, "Common Methods for Working with the View Object's Default RowSet", you saw that the `setWhereClause()` method allows you to add a dynamic WHERE clause to a view object. As you'll see in later examples in this chapter, when you use `setWhereClause()` you pass a string that contains literal database column names like this:

```
vo.setWhereClause("LAST_NAME LIKE UPPER(:NameToFind)");
```

In contrast, when you use the view criteria mechanism, you saw in Example 5–11 above that you reference the view object attribute name instead like this:

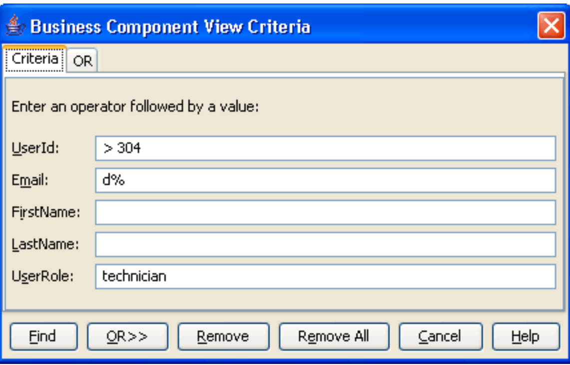
```
criteriaRow.setAttribute("LastName","B%");
```

As explained above, the view criteria rows are then translated by the view object into corresponding WHERE clause predicates that reference the corresponding column names.

5.8.3.2 Testing View Criteria in the Business Component Browser

As shown in Figure 5–19, for any view object instance that you browse, clicking the **Specify View Criteria** toolbar icon brings up the **Business Component View Criteria** dialog. The dialog allows you to create a view criteria comprising one or more view criteria rows. To apply criteria attributes from a single view criteria row, enter query-by-example criteria in the desired fields and click **Find**. To add additional view criteria rows, click **OR** and use the additional tabs that appear to switch between pages, each representing a distinct view criteria row. When you click **Find** the Business Components Browser uses the same APIs described above to create and apply the view criteria to filter the result.

Figure 5–19 Creating a View Criteria with One or More Rows in the Business Component Browser



5.8.3.3 Altering Compound Search Conditions Using Multiple View Criteria Rows

When you add multiple view criteria rows, you can call the `setConjunction()` method on a view criteria row to alter the conjunction used between the predicate corresponding to that row and the one for the previous view criteria row. The legal constants to pass as an argument are:

- `ViewCriteriaRow.VCROW_CONJ_AND`
- `ViewCriteriaRow.VCROW_CONJ_NOT`
- `ViewCriteriaRow.VCROW_CONJ_OR` (*default*)

The `NOT` value can be combined with `AND` or `OR` to create filter criteria like:

(*PredicateForViewCriteriaRow1*) AND (NOT(*PredicateForViewCriteriaRow2*))

or

(*PredicateForViewCriteriaRow1*) OR (NOT(*PredicateForViewCriteriaRow2*))

The syntax to achieve these requires using Java's bitwise `OR` operator like this:

```
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_AND | ViewCriteriaRow.VCROW_CONJ_
NOT);
```

5.8.3.4 Searching for a Row Whose Attribute Value is NULL Value

To search for a row containing a `NULL` value in a column, populate a corresponding view criteria row attribute with the value "`IS NULL`".

5.8.3.5 Searching Case-Insensitively

To search case-insensitively, call `setUpperColumns(true)` on the view criteria row to which you want the case-insensitivity to apply. This affects the WHERE clause predicate generated for `String`-valued attributes in the view object to use UPPER(COLUMN_NAME) instead of COLUMN_NAME in the predicate. Note that the value of the supplied view criteria row attributes for these `String`-valued attributes must be uppercase or the predicate won't match.

5.8.3.6 Clearing View Criteria in Effect

To clear any view criteria in effect, you can call `getViewCriteria()` on a view object and then delete all the view criteria rows from it using the `remove()` method, passing the zero-based index of the criteria row you want to remove. If you don't plan to add back other view criteria rows, you can also clear all the view criteria in effect by simply calling `applyViewCriteria(null)` on the view object.

5.8.3.7 Applying View Criteria Causes Query to be Re-parsed

A corollary of the view criteria feature described above is that each time you apply a new view criteria (or remove an existing one), the text of the view object's SQL query is effectively changed. Changing the SQL query causes the database to re-parse the statement again the next time it is executed. If you plan to use the view criteria filtering feature to apply different criteria *values* for fundamentally the same criteria attributes each time, you will get better performance by using a view object whose WHERE clause contains named bind variables as described in Section 5.9, "Using Named Bind Variables". In contrast to the view criteria filtering feature, using named bind variables you can change the *values* of the search criteria without changing the *text* of the view object's SQL statement each time those values change.

5.9 Using Named Bind Variables

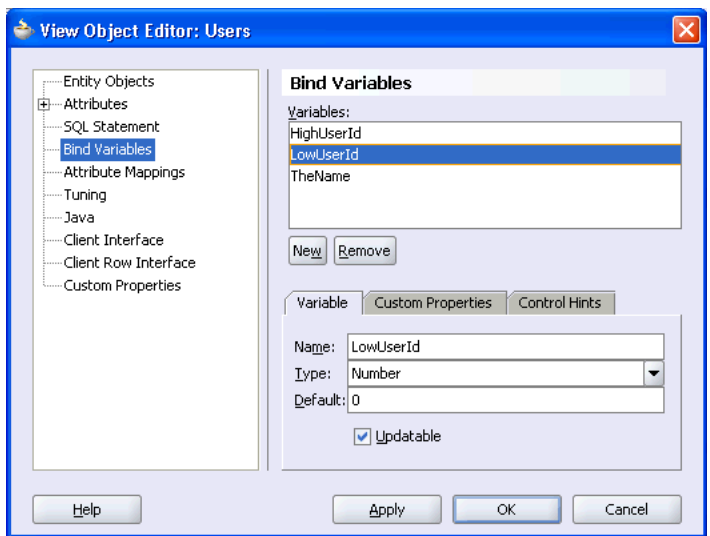
Whenever the WHERE clause of your query includes values that might change from execution to execution, you can use named bind variables. These are place holders in the SQL string whose value you can easily change at runtime without altering the text of the SQL string itself. Since the query doesn't change, the database can efficiently reuse the same parsed representation of the query across multiple executions which leads to higher runtime performance of your application.

5.9.1 Adding a Named Bind Variable

To add a named bind variable to a view object, use the **Bind Variables** tab of the Create View Object wizard or the View Object Editor. You can add as many named bind variables as you need. As shown in Figure 5–20, for each bind variable you specify its name, data type, and default value. You can name the variables as you like, but since they share the same namespace as view object attributes you need to choose names that don't conflict with existing view object attribute names. As with view objects attributes, by convention bind variable names are created with an initial capital letter.

On the **Control Hints** tab, you can also specify UI hints like **Label Text**, **Format Type**, **Format** mask and others, just as you did above with the view object attributes. These bind variable control hints are used automatically by the view layer when you build user interfaces like search pages that allow the user to enter values for the named bind variables. The **Updatable** checkbox controls whether the end user will be allowed to change the bind variable value through the user interface. If a bind variable is not updatable, then its value can only be changed programmatically by the developer.

Figure 5–20 Defining Named Bind Variables for a View Object



After defining the bind variables, the next step is to reference them in the SQL statement. While SQL syntax allows bind variables to appear both in the SELECT list and in the WHERE clause, you'll typically use them in the latter context, as part of your WHERE clause. You could edit the `UserList` view object created above, and open the **SQL Statement** page to introduce your named bind variables like this:

```
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and USER_ID between :LowUserId and :HighUserId
order by EMAIL
```

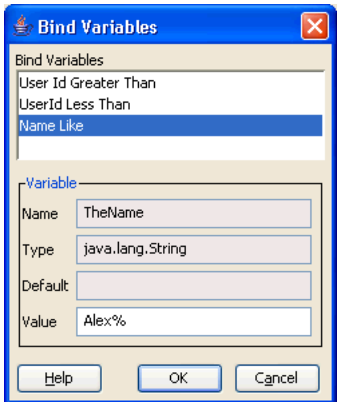
Notice that you reference the bind variables in the SQL statement by prefixing their name with a colon like `:TheName` or `:LowUserId`. You can reference the bind variables in any order and repeat them as many times as needed within the SQL statement.

5.9.2 What Happens When You Add Named Bind Variables

Once you've added one or more named bind variables to a view object, you gain the ability to easily see and set the values of these variables at runtime. Information about the name, type, and default value of each bind variable is saved in the view object's XML component definition file. If you have defined UI control hints for the bind variables, this information is saved in the view object's component message bundle file along with other control hints for the view object.

The Business Components Browser allows you to interactively inspect and change the values of the named bind variables for any view object, which can really simplify experimenting with your application module's data model when named bind parameters are involved. The first time you execute a view object in the tester, a **Bind Variables** dialog will appear, as shown in Figure 5–21. By selecting a particular bind variable in the list, you can see its name as well as both the default and current values. To change the value of any bind variable, just update its corresponding **Value** field before clicking **OK** to set the bind variable values and execute the query. Using the **Edit Bind Parameters** button in the toolbar — whose icon looks like "`:id`" — you can inspect and set the bind variables for the view object in the current panel.

Figure 5–21 Setting Bind Variables in the Tester



If you've defined the **Label Text**, **Format Type**, or **Format** control hints, the **Bind Variables** dialog helps you verify they are correctly setup by showing the label text hint in the **Bind Variables** list and formatting the **Value** attribute using the respective format mask. You can see in Figure 5–21 that the label text hints are showing for the three bind variables in the list.

5.9.3 What You May Need to Know About Named Bind Variables

There are several things you may need to know about named bind variables, including the runtime errors that are displayed when bind variables have mismatched names, the default value for bind variables, how to set existing bind variable values at runtime, and how to add a new named bind variable at runtime.

5.9.3.1 Errors Related to Bind Variables

You need to ensure that the list of named bind variables that you reference in your SQL statement matches the list of named bind variables that you've defined on the **Bind Variables** page of the View Object Editor. Failure to have these two agree correctly can result in one of the following two errors at runtime.

If you use a named bind variable in your SQL statement but have not defined it, you'll receive an error like this:

```
(oracle.jbo.SQLStmtException) JBO-27122: SQL error during statement preparation.
## Detail 0 ##
(java.sql.SQLException) Missing IN or OUT parameter at index:: 1
```

On the other hand, if you have defined a named bind variable, but then forgotten to reference it or mistyped its name in the SQL, then you will see an error like this:

```
oracle.jbo.SQLStmtException: JBO-27122: SQL error during statement preparation.
## Detail 0 ##
java.sql.SQLException: Attempt to set a parameter name that does not occur in the
SQL: LowUserId
```

The resolution in both cases is to double-check that the list of named bind variables in the SQL matches the list of named bind variables on the **Bind Variables** page.

5.9.3.2 Bind Variables Default to NULL If No Default Supplied

If you do not supply a default value for your named bind variable, it defaults to the `NULL` value at runtime. This means that if you have a WHERE clause like:

```
USER_ID = :TheUserId
```

and you do not provide a default value for the `TheUserId` bind variable, it will default to having a NULL value and cause the query to return no rows. Where it makes sense for your application, you can leverage SQL functions like NVL(), CASE, DECODE(), or others to handle the situation as you require. In fact, the `UserList` view object uses a WHERE clause fragment like:

```
upper(FIRST_NAME) like upper(:TheName)||'%'
```

so that the query will match any name if the value of `:TheName` is `null`.

5.9.3.3 Setting Existing Bind Variable Values at Runtime

To set named bind variables at runtime, use the `setNamedWhereClauseParam()` method on the `ViewObject` interface. You can use JDeveloper's **Refactor > Duplicate...** feature to create a new `TestClientBindVars` class based on the existing `TestClient.java` class from Section 5.7, "How to Create a Command-Line Java Test Client". In this new test client class, you can set the values of the `HighUserId` and `TheName` bind variables using the few additional lines of code shown in Example 5–12.

Example 5–12 Setting the Value of Named Bind Variables Programmatically

```
// changed lines in TestClient class
ViewObject vo = am.findViewObject("UserList");
vo.setNamedWhereClauseParam("TheName","alex%");
vo.setNamedWhereClauseParam("HighUserId", new Number(315));
vo.executeQuery();
// etc.
```

Running the `TestClientBindVars` class shows that your bind variables are filtering the data, and the resulting rows are only the two matching ones for Alexander Hunold and Alexander Khoo:

```
303 ahunold
315 akhoo
```

Whenever a view object's query is executed, the runtime debug diagnostics show you the actual bind variable values that get used like this:

```
[256] Bind params for ViewObject: UserList
[257] Binding param "LowUserId": 0
[258] Binding param "HighUserId": 315
[259] Binding param "TheName": alex%
```

This information that can be invaluable in isolating problems in your applications. Notice that since the code did not set the value of the `LowUserId` bind variable, it took on the design-time specified default value of `0` (zero). Also notice that the use of the UPPER() function in the WHERE clause and around the bind variable ensured that the match using the bind variable value for `TheName` was performed case-insensitively. The example code set the bind variable value to "`alex%`" with a lowercase "`a`", and the results show that it matched `Alexander`.

5.9.3.4 Adding a Named Bind Variable at Runtime

Using the view object's `setWhereClause()` method, you can add an additional filtering clause at runtime. This runtime-added WHERE clause predicate does *not* replace the design-time one, but rather further narrows the query result by getting applied in addition to any existing design-time WHERE clause. Whenever the dynamically added clause refers to a value that might change during the life of the application, you should use a named bind variable instead of concatenating the literal value into the WHERE clause predicate.

For example, assume you want to further filter the `UserList` view object at runtime based on the value of the `USER_ROLE` column in the table. Also assume that you plan to search sometimes for rows where `USER_ROLE = 'technician'` and other times where `USER_ROLE = 'User'`. While slightly fewer lines of code, it would be *bad* practice to do the following because it changes the where clause twice just to query two different *values* of the same USER_ROLE column:

```
// Don't use literal strings if you plan to change the value!
vo.setWhereClause("user_role = 'technician'");
// execute the query and process the results, and then later...
vo.setWhereClause("user_role = 'user'");
```

Instead, add a WHERE clause predicate that references named bind variables that you define at runtime like this:

```
vo.setWhereClause("user_role = :TheUserRole");
vo.defineNamedWhereClauseParam("TheUserRole", null, null);
vo.setNamedWhereClauseParam("TheUserRole","technician");
// execute the query and process the results, and then later...
vo.setNamedWhereClauseParam("TheUserRole","user");
```

This allows the text of the SQL statement to stay the same, regardless of the value of USER_ROLE you need to query on. When the query text stays the same across multiple executions, the database give you the results without having to reparse the query.

If you later need to remove the dynamically added WHERE clause and bind variable, you can use code like this:

```
vo.setWhereClause(null);
vo.removeNamedWhereClauseParam("TheUserRole");
```

An updated `TestClientBindVars` class illustrating these techniques would look like what you see in Example 5–13. In this case, the functionality that loops over the results several times has been refactored into a separate `executeAndShowResults()` method. The program first adds an additional WHERE clause of `user_id = :TheUserId` and then later replaces it with a second clause of `user_role = :TheUserRole`.

Example 5–13 TestClient Program Exercising Named Bind Variable Techniques

```
package devguide.examples.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.Number;
public class TestClient {
  public static void main(String[] args) {
    String        amDef = "devguide.examples.UserService";
    String        config = "UserServiceLocal";
    ApplicationModule am =
     Configuration.createRootApplicationModule(amDef,config);
    ViewObject vo = am.findViewObject("UserList");
    // Set the two design time named bind variables
    vo.setNamedWhereClauseParam("TheName","alex%");
    vo.setNamedWhereClauseParam("HighUserId", new Number(315));
    executeAndShowResults(vo);
    // Add an extra where clause with a new named bind variable
    vo.setWhereClause("user_id = :TheUserId");
    vo.defineNamedWhereClauseParam("TheUserId", null, null);
    vo.setNamedWhereClauseParam("TheUserId",new Number(303));
    executeAndShowResults(vo);
    vo.removeNamedWhereClauseParam("TheUserId");
    // Add an extra where clause with a new named bind variable
    vo.setWhereClause("user_role = :TheUserRole");
    vo.defineNamedWhereClauseParam("TheUserRole", null, null);
    vo.setNamedWhereClauseParam("TheUserRole","user");
    // Show results when :TheUserRole = 'user'
    executeAndShowResults(vo);
    vo.setNamedWhereClauseParam("TheUserRole","technician");
    // Show results when :TheUserRole = 'technician'
    executeAndShowResults(vo);
    Configuration.releaseRootApplicationModule(am,true);
  }
  private static void executeAndShowResults(ViewObject vo) {
    System.out.println("---");
    vo.executeQuery();
    while (vo.hasNext()) {
      Row curUser = vo.next();
      System.out.println(curUser.getAttribute("UserId")+" "+
                         curUser.getAttribute("Email"));
    }
  }
}
```

However, if you run this test program, you actually get a runtime error like this:

```
oracle.jbo.SQLStmtException: JBO-27122: SQL error during statement preparation.
Statement:
SELECT * FROM (select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and USER_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT  WHERE (user_role = :TheUserRole)
## Detail 0 ##
java.sql.SQLException: ORA-00904: "USER_ROLE": invalid identifier
```

The root cause, which appears after the `## Detail 0 ##` in the stack trace, is a SQL parsing error from the database reporting that USER_ROLE column does not exist. That's odd, since the USERS table definitely has a USER_ROLE column. The problem occurs due to the mechanism that ADF view objects use by default to apply additional runtime WHERE clauses on top of read-only queries. Section 5.9.3.5, "Understanding the Default Use of Inline Views for Read-Only Queries", explains a resolution for this issue.

5.9.3.5 Understanding the Default Use of Inline Views for Read-Only Queries

If you dynamically add an additional WHERE clause at runtime to a read-only view object, its query gets nested into an inline view before applying the additional WHERE clause. For example, suppose your query was defined as:

```
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and USER_ID between :LowUserId and :HighUserId
order by EMAIL
```

At runtime, when you set an additional WHERE clause like `user_role = :TheUserRole` as the test program did in Example 5–13, the framework nests the original query into an inline view like this:

```
SELECT * FROM(
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and USER_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT
```

and then adds the dynamic WHERE clause predicate at the end, so that the final query the database sees is:

```
SELECT * FROM(
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and USER_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT
WHERE user_role = :TheUserRole
```

This query "wrapping" is necessary in the general case since the original query could be arbitrarily complex, including SQL UNION, INTERSECT, MINUS, or other operators that combine multiple queries into a single result. In those cases, simply "gluing" the additional runtime onto the end of the query text could produce unexpected results since, for example, it might only apply to the *last* of several `UNION`'ed statements. By nesting the original query verbatim into an inline view, the view object guarantees that your additional WHERE clause is correctly used to filter the results of the original query, regardless of how complex it is. The downside that you're seeing here with the ORA-904 error is that the dynamically added WHERE clause can refer only to columns that have been selected in the original query.

Section 27.3.3.7, "Disabling the Use of Inline View Wrapping at Runtime" explains how to disable this query nesting when you don't require it, but for now the simplest solution is to edit the `UserList` view object and add the USER_ROLE column to the end of its query's SELECT list on the SQL Statement page. Just adding the new column name at the end of the existing SELECT list — of course, preceded by a comma — is enough to do the job: the **View Object Editor** will automatically keep your view object's attribute list in sync with the query statement.

The modified test program in Example 5–13 now produces the expected results:

```
---
303 ahunold
315 akhoo
---
303 ahunold
---
315 akhoo
---
303 ahunold
```

5.10 Working with Master/Detail Data

So far you've worked with a single view object that queries a single `USERS` table. In practice, many queries you'll need to work with will involve multiple tables that are related by foreign keys. There are two ways you can use view objects to handle this situation, you can either:

- Join them in the main query to show additional descriptive information in each row of the main query result
- Create separate view objects that query the related information and then link a "source" view object to one or more "target" view objects to form a master/detail hierarchy.

Figure 5–22 illustrates the different "shape" that these two options produce. The join is a single "flattened" result. The master/detail linked queries produce a multilevel result.

Figure 5–22 Difference Between Join Query Result and Linked Master/Detail Queries

5.10.1 How to Create a Read-Only View Object Joining Tables

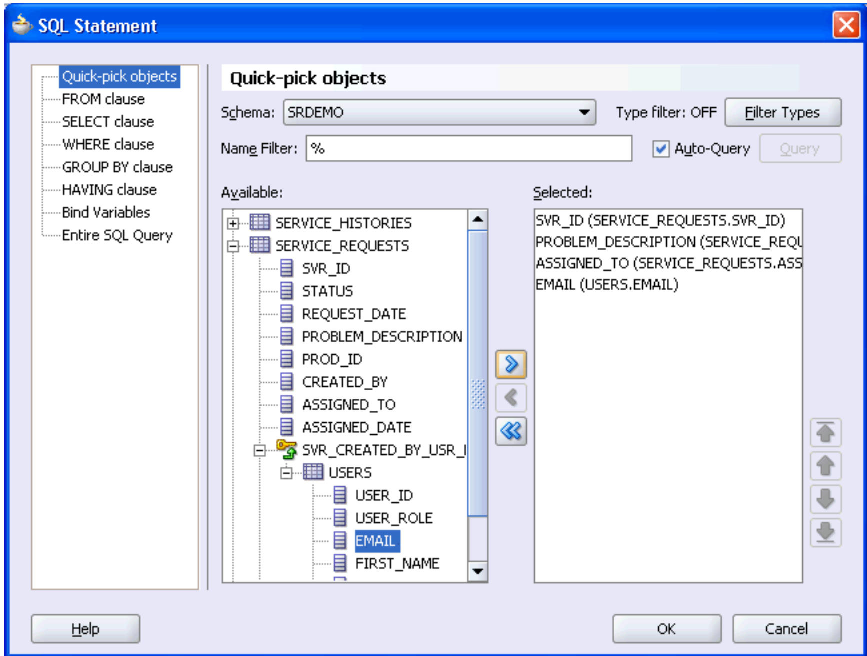
To create a read-only view object joining two tables, use the Create View Object wizard. As an example, you'll create a view object named `OpenOrPendingServiceRequests` that joins the `SERVICE_REQUEST` and `USER` tables. For each service request, you'll display the email of the user who created the request.

In step 1 ensure that you've selected **Read-only Access**, and in step 2 on the **SQL Statement** page, enter the SQL query statement that joins the desired tables. If you want interactive assistance to build up the right SQL statement, you can click on the **Query Builder** button.

5.10.1.1 Using the Query Builder to Simplify Creating Joins

As shown in Figure 5–23, on the **Quick-pick objects** page of the query builder dialog, you can see the tables in your schema, including the foreign keys that relate them to other tables. To include columns in the select list of the query, click on them in the **Available** list and shuttle them to the **Selected** list. Figure 5–23 shows the result of selecting the SVR_ID, PROBLEM_DESCRIPTION, and ASSIGNED_TO columns from the `SERVICE_REQUESTS` table, along with the EMAIL column from the USERS table. In the SERVICE_REQUESTS table, beneath the SVR_CREATED_BY_USR_FK foreign key, select the EMAIL column from the USERS table and the query builder automatically determines the required join clause for you.

Figure 5–23 Using the View Object Query Builder to Define a Join



On the WHERE Clause page of the query builder, shuttle the `STATUS` column into the **WHERE** clause box, and complete the job by adding the `IN ('Open','Pending')` yourself. Click **OK** in the query builder to create the following query:

Example 5–14 Creating a Query Using SQL Builder

```
SELECT
    SERVICE_REQUESTS.SVR_ID SVR_ID,
    SERVICE_REQUESTS.PROBLEM_DESCRIPTION PROBLEM_DESCRIPTION,
    SERVICE_REQUESTS.ASSIGNED_TO ASSIGNED_TO,
    USERS.EMAIL EMAIL
FROM
    SERVICE_REQUESTS INNER JOIN USERS
      ON SERVICE_REQUESTS.CREATED_BY = USERS.USER_ID
WHERE
    SERVICE_REQUESTS.STATUS IN ('Open', 'Pending')
```

Notice the EMAIL column in the query. It represents the email of the person who created the service request, but its column name is not as descriptive as it could be. In Section 5.2.3.2, "Working with Queries That Include SQL Expressions", you learned one way to affect the default Java-friendly name of the view object attributes by assigning a column alias. Here you can adopt an alternative technique. You'll use one of the later panels in the Create View Object wizard to rename the view object attribute directly as part of the creation process. Renaming the view object here saves you from having to edit the view object again, when you already know the different attribute names that you'd like to use.

Click **Next** four times to get to the **Attributes Settings** page. Select the `Email` attribute in the **Select Attribute** dropdown list at the top of the page and change the value in the `Name` field to `CreatedByEmail`. Then click **Finish** to create the `OpenOrPendingServiceRequests` view object. An `OpenOrPendingServiceRequests.xml` component definition file is created to save the view object's declarative settings.

5.10.1.2 Testing the Join View

To test the new view object, edit the `UserService` application module and on the **Data Model** page, add an instance of the `OpenOrPendingServiceRequests` to the data model. Instead of accepting the default `OpenOrPendingServiceRequests1` instance name, change the instance name to `AllOpenOrPendingServiceRequests`. After doing this, you can launch the Business Components Browser and verify that the join query is working as expected.

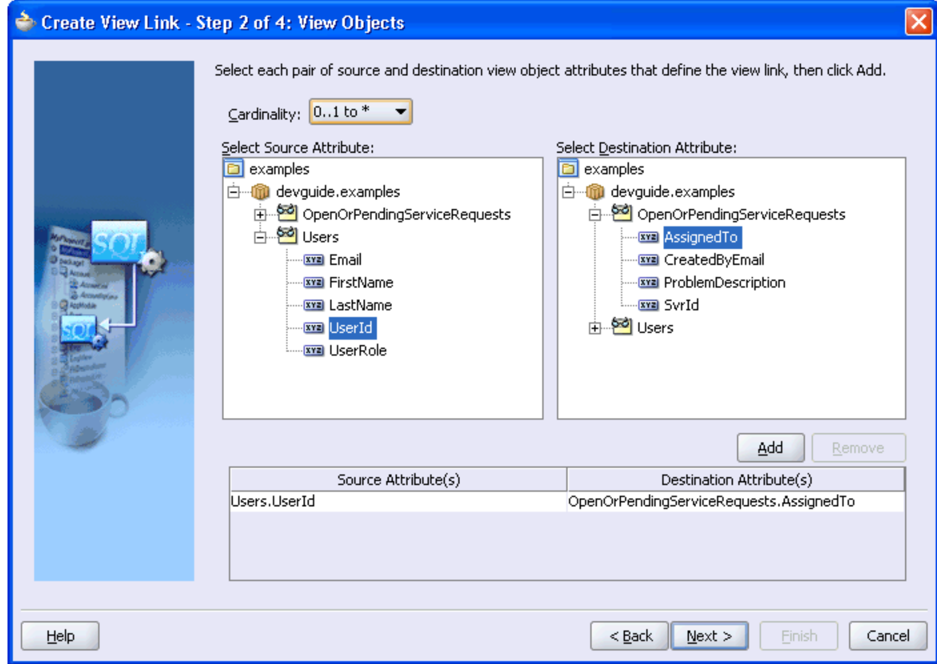
5.10.2 How to Create Master/Detail Hierarchies Using View Links

When your needs call for showing the user a set of master rows, and for each master row a set of coordinated detail rows, then you can create view links to define how you want the master and detail view objects to relate. Assume you want to link the `UserList` view object to the `OpenOrPendingServiceRequests` view object to create a master/detail hierarchy of users and the related set of open or pending service requests that have been *assigned* to them.

To create the view link, use the **Create View Link wizard**. The wizard is available from the **New Gallery** in the **Business Tier > ADF Business Components** category. In step 1, on the **Name** page provide a name for the view link and a package where its definition will reside. Given its purpose, a name like `RequestsAssignedTo` is a fine name, and for simplicity keep it in the same `devguide.examples` package as the view objects.

In step 2, on the **View Objects** page, select a "source" attribute to use from the view object that will act as the master. Figure 5–24 shows selecting the `UserId` attribute from the `Users` view object in this role. Next, select a corresponding destination attribute from the view object that will act as the detail. Since you want the detail query to show service requests that are assigned to the currently selected user, select the `AssignedTo` attribute in the `OpenOrPendingServiceRequests` to play this role. Finally, click **Add** to add the matching attribute pair to the table of source and destination attribute pairs below. If there were multiple attribute pairs required to define the link between master and detail, you could repeat these steps to add additional source/target attribute pairs. For this example, the one (`UserId`,`AssignedTo`) pair is all that's required.

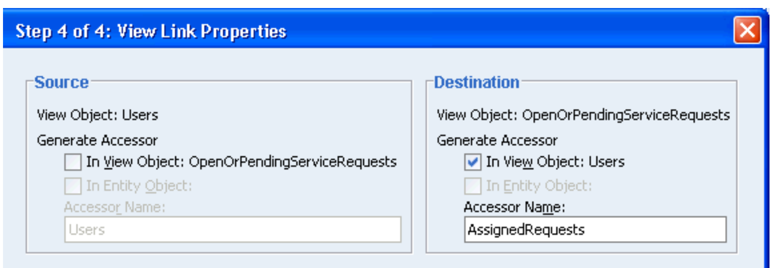
Figure 5–24 Defining Source/Target Attribute Pairs While Creating a View Link



In step 3, on the **View Link SQL** page, you can preview the view link SQL predicate that will be used at runtime to access the correlated detail rows from the destination view object for the current row in the source view object.

In step 4, on the **View Link Properties** page you control whether the view link represents a one-way relationship or a bidirectional one. Notice in Figure 5–25 that in the **Destination** group box for the `OpenOrPendingServiceRequests` view object, the **Generate Accessor In View Object: Users** box is checked. In contrast, in the **Source** group box for the `Users` view object, the **Generate Accessor In View Object: OpenOrPendingServiceRequests** box is not checked. By default, a view link is a one-way relationship that allows the current row of the source (master) to access a set of related rows in the destination (detail) view object. These checkbox settings indicate that you'll be able to access a detail collection of rows from the `OpenOrPendingServiceRequests` for the current row in the `Users` view object, but not vice versa. For this example, a default one-way view link will be fine, so leave the other checkbox unchecked.

Figure 5–25 View Link Properties Control Name and Direction of Accessors



The **Accessor Name** field in the destination group box indicates the name you can use to programmatically access the related collection of `OpenOrPendingServiceRequests` rows for the current row in `Users`. By default the accessor name will be `OpenOrPendingServiceRequests`, matching the name of the destination view object. To make it more clear that the related collection of service requests is a collection of requests that are assigned to the current user, as you can see in Figure 5–25 you can change the name of the accessor to `AssignedRequests`.

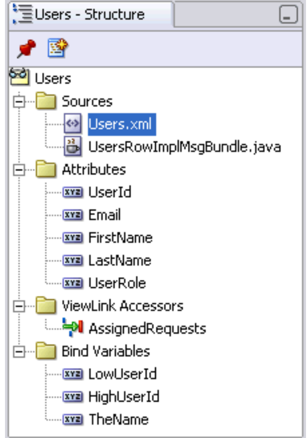
To create the view link, click **Finish**.

5.10.3 What Happens When You Create Master/Detail Hierarchies Using View Links

When you create a view link, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. In Section 5.10.2, the view link was named `RequestsAssignedTo` in the `devguide.examples` package, so the XML file created will be `./devguide/examples/RequestsAssignedTo.xml` under the project's source path. This XML file contains the declarative information about the source and target attribute pairs you've specified.

In addition to saving the view link component definition itself, JDeveloper also updates the XML definition of the *source* view object in the view link relationship to add information about the view link accessor you've defined. As a confirmation of this, you can select the `Users` view object in the **Application Navigator** and inspect its details in the **Structure window**. As illustrated in Figure 5–26, you now see the new `AssignedRequests` accessor in the **ViewLink Accessor** category.

Figure 5–26 Structure Window Showing Details of the Users View Object



5.10.4 What You May Need to Know About View Links

To work with view links effectively, there are a few more things you may need to know, including: that view link accessor attributes return a `RowSet`, how to access a detail collection using the view link accessor, and how to enable active master/detail coordination in the date model.

5.10.4.1 View Link Accessor Attributes Return a RowSet

At runtime the `getAttribute()` method on a `Row` allows you to access the value of any attribute of a row in the view object's result set by name. The view link accessor behaves like an additional attribute in the current row of the source view object, so you can use the same `getAttribute()` method to retrieve its value. The only practical difference between a regular view attribute and a view link accessor attribute is its data type. Whereas a regular view attribute typically has a scalar data type with a value like `303` or `ahunold`, the value of a view link accessor attribute is a row set of zero or more correlated detail rows. Assuming that `curUser` is a `Row` from some instance of the `Users` view object, you can write a line of code to retrieve the detail row set of open or pending assigned requests:

```
RowSet reqs = (RowSet)curUser.getAttribute("AssignedRequests");
```

Note: If you generate the custom Java class for your view row, the type of the view link accessor will be `RowIterator`. Since at runtime the return value will always be a `RowSet`, it is safe to cast the view link attribute value to a `RowSet`.

5.10.4.2 How to Access a Detail Collection Using the View Link Accessor

Once you've retrieved the `RowSet` of detail rows using a view link accessor, you can loop over the rows it contains using the same pattern used the view object's row set of results:step

```
while (reqs.hasNext()) {
  Row curReq = reqs.next();
  System.out.println("--> (" + curReq.getAttribute("SvrId") + ") " +
                     curReq.getAttribute("ProblemDescription"));
}
```

If you use JDeveloper's **Refactor > Duplicate...** functionality on the existing `TestClient.java` class, you can easily "clone" it to create a `TestClient2.java` class that you'll modify as shown in Example 5–15 to make use of these new techniques. Notice that the lines left in the `main()` method are setting a dynamic WHERE clause to restrict the `UserList` view object instance to show only users whose `USER_ROLE` has the value `technician`. The second change was enhancing the `executeAndShowResults()` method to access the view link accessor attribute and print out the request number (`SvrId`) and `ProblemDescription` attribute for each one.

Example 5–15 Programmatically Accessing Detail Rows Using the View Link Accessor

```
package devguide.examples.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
public class TestClient2 {
  public static void main(String[] args) {
    String amDef = "devguide.examples.UserService";
    String config = "UserServiceLocal";
    ApplicationModule am =
     Configuration.createRootApplicationModule(amDef, config);
    ViewObject vo = am.findViewObject("UserList");
    // Add an extra where clause with a new named bind variable
    vo.setWhereClause("user_role = :TheUserRole");
    vo.defineNamedWhereClauseParam("TheUserRole", null, null);
    vo.setNamedWhereClauseParam("TheUserRole", "technician");
    // Show results when :TheUserRole = 'technician'
    executeAndShowResults(vo);
    Configuration.releaseRootApplicationModule(am, true);
  }
  private static void executeAndShowResults(ViewObject vo) {
    System.out.println("---");
    vo.executeQuery();
    while (vo.hasNext()) {
      Row curUser = vo.next();
      // Access the row set of details using the view link accessor attribute
      RowSet reqs = (RowSet)curUser.getAttribute("AssignedRequests");
      long numReqs = reqs.getEstimatedRowCount();
      System.out.println(curUser.getAttribute("UserId") + " " +
                         curUser.getAttribute("Email")+" ["+
                         numReqs+" requests]");
      while (reqs.hasNext()) {
        Row curReq = reqs.next();
        System.out.println("--> (" + curReq.getAttribute("SvrId") + ") " +
                           curReq.getAttribute("ProblemDescription"));
      }
    }
  }
}
```

Running `TestClient2` shows the following results in the Log window. Each technician is listed, and for each technician that has some open or pending service requests, the information about those requests appears beneath their name.

```
---
303 ahunold [0 requests]
304 bernst [2 requests]
--> (102) Washing Machine does not turn on
--> (108) Freezer full of frost
305 daustin [1 requests]
--> (104) Spin cycle not draining
307 dlorentz [0 requests]
306 vpatabal [2 requests]
--> (107) Fridge is leaking
--> (112) My Dryer does not seem to be getting hot
```

If you run TestClient2 with debug diagnostics enabled, you will see the SQL queries that the view object performed. The view link WHERE clause predicate is used to automatically perform the filtering of the detail service request rows for the current row in the UserList view object.

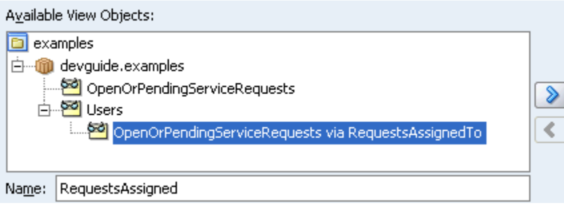
5.10.4.3 How to Enable Active Master/Detail Coordination in the Data Model

You've seen that the process of defining a view link introduces a view link attribute in the source view object, which enables programmatic navigation to a row set of correlated details. In this scenario, the view link plays a *passive* role, simply defining the information necessary to retrieve the coordinated detail row set when your code requests it. The view link accessor attribute is present and programmatically accessible in any result rows from any instance of the view link's source view object. In other words, programmatic access does not require modifying the UserService application module's data model.

However, since master/detail user interfaces are such a frequent occurrence in enterprise applications, the view link can be also used in a more *active* fashion to avoid having to coordinate master/detail screen programmatically. You opt to have this active master/detail coordination performed by *explicitly* adding an instance of a view-linked view object to your application module's data model.

To accomplish this, edit the UserService application module and open the **Data Model** page. As shown in Figure 5–27, you'll see that the **Available View Objects** list now shows the OpenOrPendingServiceRequests view object *twice*: once on its own, and once as a detail view object via the RequestsAssignedTo view link.

Figure 5–27 Adding a Detail View Object to the Data Model



To add a detail instance of a view object:

1. In the **Data Model** list on the right, select the instance of the Users view object in the **Data Model** list that you want to be the actively-coordinating master

   The data model has only one instance of the Users view object named UserList, so select the UserList instance.

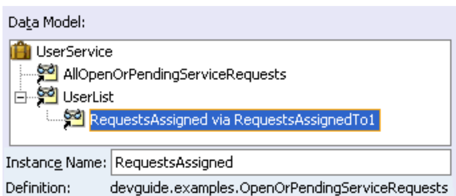
2. In the **Available View Objects** list, select the OpenOrPendingServiceRequests node that is indented beneath the Users view object.

3. Enter a name for the detail instance you're about to create in the **Name** field below the **Available View Objects** list. As shown in Figure 5–27, call the instance RequestsAssigned.

4. Click the Add Instance button > to add the detail instance to the currently selected master instance in the data model, with the name you've chosen.

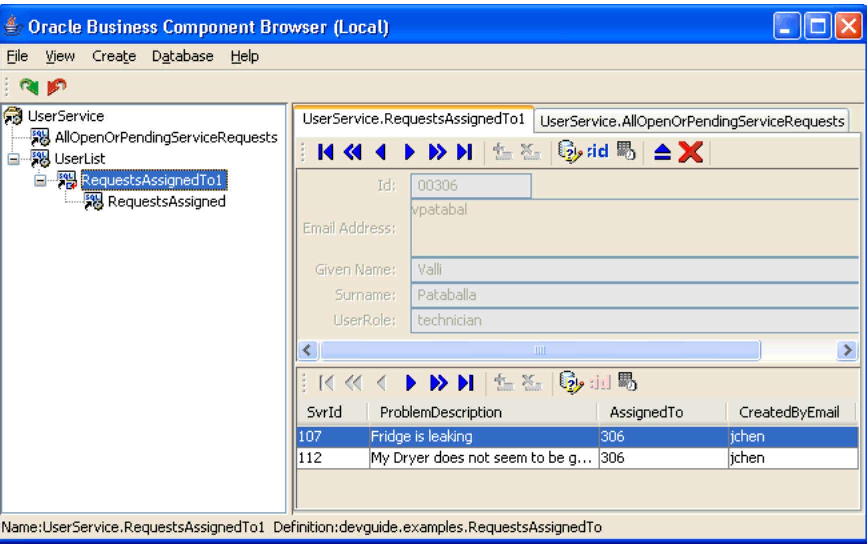
After following these steps, your **Data Model** list will look like what you see in Figure 5–28.

Figure 5–28 UserService Data Model with View Linked View Object



The easiest way to see the effect of this active master/detail coordination is to launch the **Business Components Browser** on the `UserService` by choosing **Test** from its context menu in the **Application Navigator**. Figure 5–29 shows the browser window you will see. The data model tree shows the view link instance that is actively coordinating the `UserList` view object instance with the `RequestsAssigned` view object instance. It has the default view link instance name of `RequestsAssignedTo1`. Double-clicking this view link instance node in the tree opens the master/detail panel that you see in Figure 5–29. You'll see that when you use the toolbar buttons to navigate in the master view object — changing the view object's current row as a result — the coordinated set of details is automatically refreshed and the user interface stays in sync.

Figure 5–29 Experimenting with Active Data Model Master/Detail Coordination



If you also double-click on the `AllOpenOrPendingServiceRequests` view object instance that you added earlier, a second tab will open to show its data. Notice that it is another instance of the same `devguide.examples.Users` view object; however, since it is not being actively coordinated by a view link, its query is not constrained by the current row in the `UserList`.

So far you've seen a view link that defines a basic master/detail relationship between two view objects. Keep in mind that by creating more view links you can achieve master/detail hierarchies of any complexity, including:

- Multilevel master/detail/detail
- Master with multiple (peer) details
- Detail with multiple masters

The steps to define these more complex hierarchies are the same as the ones covered here, you just need to create it one view link at time.

5.11 Generating Custom Java Classes for a View Object

As you've seen, all of the basic querying functionality of a view object can be achieved without using custom Java code. Clients can retrieve and iterate through the data of any SQL query without resorting to any custom code on the view object developer's part. In short, for many read-only view objects, once you've defined the SQL statement, you're done. However, it's important to understand how to enable custom Java generation for a view object when your needs might require it. Appendix D, "Most Commonly Used ADF Business Components Methods" provides a quick reference to the most common code that you will typically write, use, and override in your custom view object and view row classes. Later chapters discuss specific examples of how the SRDemo application uses custom code in these classes as well.

5.11.1 How To Generate Custom Classes

To enable the generation of custom Java classes for a view object, use the **Java** page of the View Object Editor. As shown in Figure 5–30, there are three optional Java classes that can be related to a view object. The first two in the list are the most commonly used:

- *View object class*, which represents the component that performs the query
- *View row class*, which represents each row in the query result

Figure 5–30 View Object Custom Java Generation Options

5.11.1.1 Generating Bind Variable Accessors

When you enable the generation of a custom view object class, if you also select the **Bind Variable Accessors** checkbox, then JDeveloper generates getter and setter methods in your view object class. Since the `Users` view object had three named bind variables (`TheName`, `LowUserId`, and `HighUserId`), the custom `UsersImpl.java` view object class would have corresponding methods like this:

```
public Number getLowUserId() {...}
public void setLowUserId(Number value) {...}
public Number getHighUserId(){...}
public void setHighUserId(Number value) {...}
public String getTheName() {...}
public void setTheName(String value){...}
```

These methods allow you to set a bind variable with compile-time type-checking to ensure you are setting a value of the appropriate type. That is, instead of writing a line like this to set the value of the `LowUserId`:

```
vo.setNamedWhereClauseParam("LowUserId",new Number(150));
```

You can write the code like:

```
vo.setLowUserId(new Number(150));
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `setLowUserName` instead of `setLowUserId`:

```
// spelling name wrong gives compile error
vo.setLowUserName(new Number(150));
```

Or if you were to incorrectly pass a value of the wrong data type, like "ABC" instead of `Number` value:

```
// passing String where number expected gives compile error
vo.setLowUserId("ABC");
```

Without the generated bind variable accessors, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both variable name and value wrong, but compiler cannot catch it
vo.setNamedWhereClauseParam("LowUserName","ABC");
```

It contains both an incorrectly spelled bind variable name, as well as a bind variable value of the wrong datatype. If you use the generic APIs on the `ViewObject` interface, errors of this sort will raise exceptions at runtime instead of being caught at compile time.

5.11.1.2 Generating View Row Attribute Accessors

When you enable the generation of a custom view row class, if you also select the **Accessors** checkbox, then JDeveloper generates getter and setter methods for each attribute in the view row. For the `Users` view object, the corresponding custom `UsersRowImpl.java` class would have methods like this generated in it:

```
public Number getUserId() {...}
public void setUserId(Number value) {...}
public String getEmail() {...}
public void setEmail(String value) {...}
public String getFirstName() {...}
public void setFirstName(String value) {...}
public String getLastName() {...}
public void setLastName(String value) {...}
public String getUserRole() {...}
public void setUserRole(String value) {...}
```

These methods allow you to work with the row data with compile-time checking of the correct datatype usage. That is, instead of writing a line like this to get the value of the `UserId` attribute:

```
Number userId = (Number)row.getAttribute("UserId");
```

you can write the code like:

```
Number userId = row.getUserId();
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `UserIdentifier` instead of `UserId`:

```
// spelling name wrong gives compile error
Number userId = row.getUserIdentifier();
```

Without the generated view row accessor methods, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both attribute name and type cast are wrong, but compiler cannot catch it
String userId = (String)row.getAttribute("UserIdentifier");
```

It contains both an incorrectly spelled attribute name, as well as an incorrectly-typed cast of the `getAttribute()` return value. Using the generic APIs on the `Row` interface, errors of this kind will raise exceptions at runtime instead of being caught at compile time.

5.11.1.3 Exposing View Row Accessors to Clients

When enabling the generation of a custom view row class, if you choose to generate the view row attribute accessor, you can also optionally select the **Expose Accessor to the Client** checkbox. This causes an additional custom row interface to be generated which application clients can use to access custom methods on the row without depending directly on the implementation class. As you learned in Chapter 4, "Overview of ADF Business Components", having client code work with business service tier interfaces instead of concrete classes is a best practice which ensures that client code does not need to change when your server-side implementation does.

In the case of the Users view object, exposing the accessors to the client will generate a custom row interface named UsersRow. This interface is created in the common subpackage of the package in which the view object resides. Having the row interface allows clients to write code that accesses the attributes of query results in a strongly typed manner. Example 5–16 shows a TestClient3 sample client program that casts the results of the next() method to the UsersRow interface so that it can call getUserId() and getEmail().

Example 5–16 Simple Example of Using Client Row Interface with Accessors

```
package devguide.examples.client;
import devguide.examples.common.UsersRow;
import oracle.jbo.ApplicationModule;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.Number;
public class TestClient3 {
  public static void main(String[] args) {
    String amDef = "devguide.examples.UserService";
    String config = "UserServiceLocal";
    ApplicationModule am =
      Configuration.createRootApplicationModule(amDef, config);
    ViewObject vo = am.findViewObject("UserList");
    vo.executeQuery();
    while (vo.hasNext()) {
     // Cast next() to a strongly-typed UsersRow interface
      UsersRow curUser = (UsersRow)vo.next();
      Number userId = curUser.getUserId();
      String email  = curUser.getEmail();
      System.out.println(userId+ " " + email);
    }
    Configuration.releaseRootApplicationModule(am, true);
  }
}
```

5.11.1.4 Configuring Default Java Generation Preferences

You've seen how to generate custom Java classes for your view objects when you need to customize their runtime behavior, or if you simply prefer to have strongly typed access to bind variables or view row attributes.

To configure the default settings for ADF Business Components custom Java generation, choose **Tools | Preferences** and open the **Business Components** page to set your preferences to be used for business components created in the future. Oracle recommends that developers getting started with ADF Business Components set their preference to generate no custom Java classes by default. As you run into specific needs, you can enable just the bit of custom Java you need for that one component. Over time, you'll discover which set of defaults works best for you.

5.11.2 What Happens When You Generate Custom Classes

When you choose to generate one or more custom Java classes, JDeveloper creates the Java file(s) you've indicated. For a view object named devguide.examples.Users, the default names for its custom Java files will be UsersImpl.java for the view object class and UsersRowImpl.java for the view row class. Both files get created in the same ./devguide/examples directory as the component's XML component definition file.

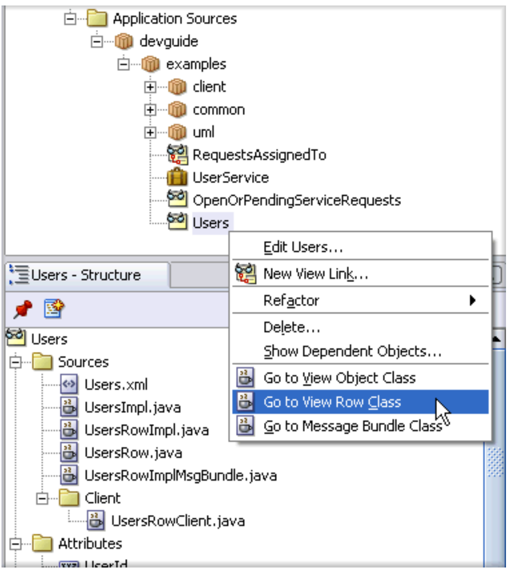
The Java generation options for the view object are continue to be reflected on the **Java** page on subsequent visits to the View Object Editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom java classes up to date with any changes you make in the editor. If later you decide you didn't require a custom Java file for any reason, unchecking the relevant options in the **Java** page causes the custom Java files to be removed.

5.11.2.1 Seeing and Navigating to Custom Java Files

As with all ADF components, when you select a view object in the Application Navigator, the Structure window displays all of the implementation files that comprise it. The only required file is the XML component definition file. You saw above that when translatable UI control hints are defined for a component, it will have a component message bundle file as well. As shown in Figure 5–31, when you've enabled generation of custom Java classes, they also appear under the **Sources** folder for the view object. When you need to see or work with the source code for a custom Java file, there are two ways to open the file in the source editor:

- Choose the relevant **Go to** option in the context menu as shown in Figure 5–31
- Double-click on a file in the **Sources** folder in the Structure window

Figure 5–31 Seeing and Navigating to Custom Java Classes for a View Object



5.11.3 What You May Need to Know About Custom Classes

See the following sections for additional information to help you use custom Java classes.

5.11.3.1 About the Framework Base Classes for a View Object

When you use an "XML-only" view object, at runtime its functionality is provided by the default ADF Business Components implementation classes. Each custom Java class that gets generated will automatically extend the appropriate ADF Business Components base class so that your code inherits the default behavior and can easily add or customize it. A view object class will extend `ViewObjectImpl`, while the view row class will extend `ViewRowImpl` (both in the `oracle.jbo.server` package).

5.11.3.2 You Can Safely Add Code to the Custom Component File

Based perhaps on previous negative experiences, some developers are hesitant to add their own code to generated Java source files. Each custom Java source code file that JDeveloper creates and maintains for you includes the following comment at the top of the file to clarify that it is safe to add your own custom code to this file:

```
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
```

JDeveloper does not blindly regenerate the file when you click the **OK** or **Apply** button in the component editor. Instead, it performs a smart update to the methods that it needs to maintain, leaving your own custom code intact.

5.11.3.3 Attribute Indexes and InvokeAccessor Generated Code

As you've seen, the view object is designed to function either in an XML-only mode or using a combination of an XML component definition and a custom Java class. Since attribute values are not stored in private member fields of a view row class, such a class is not present in the XML-only situation. Instead, in addition to a name, attributes are also assigned a numerical index in the view object's XML component definition, on a zero-based, sequential order of the <ViewAttribute> and association-related <ViewLinkAccessor> tags in that file. At runtime, the attribute values in an view row are stored in a structure that is managed by the base `ViewRowImpl` class, indexed by the attribute's numerical position in the view object's attribute list.

For the most part this private implementation detail is unimportant. However, when you enable a custom Java class for your view row, this implementation detail is related to some of the generated code that JDeveloper automatically maintains in your view row class, and you may want to understand what that code is used for. For example, in the custom Java class for the `Users` view row, Example 5–17 shows that each attribute or view link accessor attribute has a corresponding generated integer constant. JDeveloper ensures that the values of these constants correctly reflect the ordering of the attributes in the XML component definition.

Example 5–17 Attribute Constants Are Automatically Maintained in the Custom View Row Java Class

```
public class UsersRowImpl extends ViewRowImpl implements UsersRow {
  public static final int USERID = 0;
  public static final int EMAIL = 1;
  public static final int FIRSTNAME = 2;
  public static final int LASTNAME = 3;
  public static final int USERROLE = 4;
  public static final int ASSIGNEDREQUESTS = 5;
  // etc.
```

You'll also notice that the automatically maintained, strongly typed getter and setter methods in the view row class use these attribute constants like this:

```
// In devguide.examples.UsersRowImpl class
public String getEmail() {
  return (String) getAttributeInternal(EMAIL); // <-- Attribute constant
}
public void setEmail(String value) {
  setAttributeInternal(EMAIL, value);// <-- Attribute constant
}
```

The last two aspects of the automatically maintained code related to view row attribute constants are the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods. These methods optimize the performance of attribute access by numerical index, which is how generic code in the `ViewRowImpl` base class typically accesses attribute values. An example of the `getAttrInvokeAccessor()` method looks like the following from the `ServiceRequestImpl.java` class. The companion `setAttrInvokeAccessor()` method looks similar.

```
// In devguide.examples.UsersRowImpl class
protected Object getAttrInvokeAccessor(int index,AttributeDefImpl attrDef)
throws Exception {
  switch (index) {
  case USERID:           return getUserId();
  case EMAIL:            return getEmail();
  case FIRSTNAME:        return getFirstName();
  case LASTNAME:         return getLastName();
  case USERROLE:         return getUserRole();
  case ASSIGNEDREQUESTS: return getAssignedRequests();
  default:
    return super.getAttrInvokeAccessor(index, attrDef);
  }
}
```

The rules of thumb to remember about this generated attribute-index related code are the following.

The Do's

- Add custom code if needed inside the strongly typed attribute getter and setter methods
- Use the View Object Editor to change the order or type of view object attributes

  JDeveloper will change the Java signature of getter and setter methods, as well as the related XML component definition for you.

The Don'ts

- Don't modify the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods
- Don't change the values of the attribute index numbers by hand

> **Note:** If you need to manually edit the generated attribute constants, perhaps due to source control merge conflicts, you must ensure that the zero-based ordering reflects the sequential ordering of the <ViewAttribute> and <ViewLinkAccessor> tags in the corresponding view object XML component definition.

6 Creating a Business Domain Layer Using Entity Objects

This chapter describes how to use entity objects to create a reusable layer of business domain objects for use in your J2EE applications.

This chapter includes the following sections:

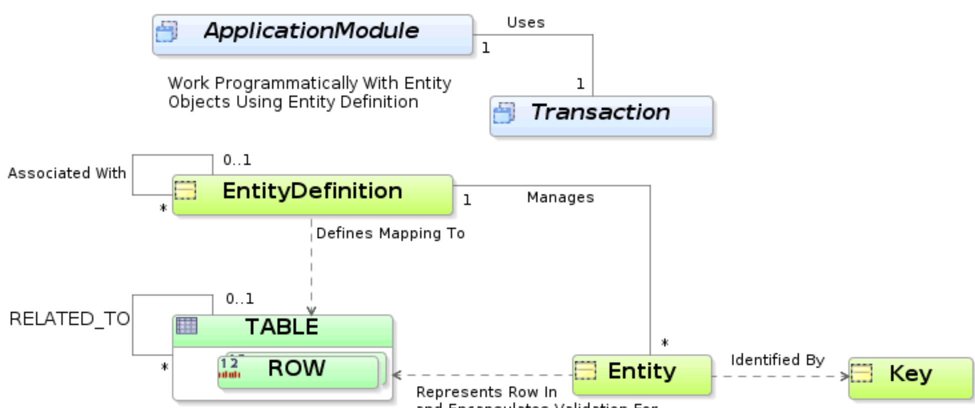
- Section 6.1, "Introduction to Entity Objects"
- Section 6.2, "Creating Entity Objects and Associations"
- Section 6.3, "Creating and Configuring Associations"
- Section 6.4, "Creating an Entity Diagram for Your Business Layer"
- Section 6.5, "Defining Attribute Control Hints"
- Section 6.6, "Configuring Declarative Runtime Behavior"
- Section 6.7, "Using Declarative Validation Rules"
- Section 6.8, "Working Programmatically with Entity Objects and Associations"
- Section 6.9, "Generating Custom Java Classes for an Entity Object"
- Section 6.10, "Adding Transient and Calculated Attributes to an Entity Object"

6.1 Introduction to Entity Objects

An entity object is the ADF Business Components component that represents a row in a database table and simplifies modifying its data. Importantly, it allows you to encapsulate domain business logic for those rows to ensure that your business policies and rules are consistently validated. By the end of this chapter, you'll understand the concepts shown in Figure 6–1:

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

Figure 6–1 Entity Object Encapsulates Business Logic for a Table



When your application module creates, modifies, or removes entity objects and commits the transaction, changes are saved automatically. When you need to work together with a `ServiceRequest` and the `User` who created it, or the `ServiceHistory` entries it logically contains, then associations between entities simplify the task. Entity objects support numerous declarative business logic features to enforce the validity of your data as well. As you'll see in more detail in later chapters, you will typically complement declarative validation with additional custom application logic and business rules to cleanly encapsulate a maximum amount of domain business logic into each entity object. Your associated set of entity objects forms a reusable business domain layer that you can exploit in multiple applications.

Note: To experiment with a working version of the examples in this chapter, download the `DevGuideExamples` workspace from the *Example Downloads* page at `http://otn.oracle.com/documentation/jdev/b25947_01/` and see the `BusinessLayerWithEntityObjects` project.

6.2 Creating Entity Objects and Associations

The simplest way to create entity objects and associations is to reverse-engineer them from existing tables. Since often you will already have a database schema to work with, the simplest way is also the most common approach that you'll use in practice. When needed, you can also create an entity object from scratch, and then generate a table for it later as well.

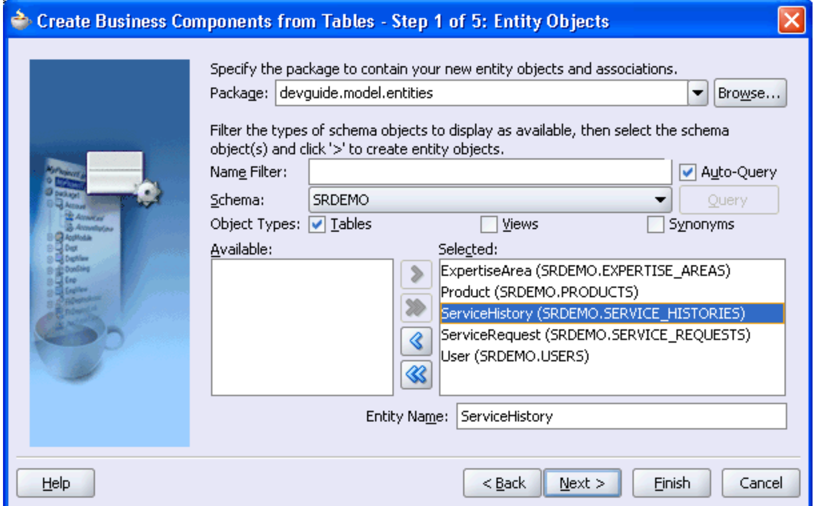
6.2.1 How to Create Entity Objects and Associations from Existing Tables

To create an entity object, use the **Business Components from Tables wizard**. The wizard is available from the **New Gallery** in the **Business Tier > ADF Business Components** category. If it's the first component you're creating in the project, the **Initialize Business Components Project** dialog appears to allow you to select a database connection before the wizard will appear. These examples assume that you're working with an connection named `SRDemo` for the `SRDEMO` schema.

In step 1 on the **Entity Objects** page, you select a list of tables from the **Available** list for which you want to create entity objects. The **Package** field at the top allows you to indicate the package in which all of the entity objects will be created. If the **Auto-Query** checkbox is checked, then the list of available tables appears immediately, otherwise you need to click the **Query** button to retrieve the list after optionally providing a name filter. Once you have selected a table from the **Available** list, the proposed entity object name for that table appears in the **Selected** list with the related table name in parenthesis. Clicking an entity object name in the **Selected** list, you can use the **Entity Name** field below to change the default entity object name. Since each entity object instance represents a **single** row in a particular table, it is best practice to name the entity objects with a singular noun like `User`, `Product`, and `ServiceHistory` instead of their plural counterparts. Figure 6–2 shows what the wizard page looks like after selecting all five tables in the `SRDEMO` schema, setting a package name of `devguide.model.entities`, and renaming each proposed entity object to have a singular name.

Note: Since an entity object represents a database row, it seems natural to call it an entity row. Alternatively, since at runtime the entity row is an instance of a Java object that encapsulates business logic for that database row, the more object-oriented term *entity instance* is also appropriate. Therefore, these two terms are interchangeable.

Figure 6–2 Creating Entity Objects for Existing Tables



Click **Finish** to create the entity objects. A progress dialog appears while the components are created, and then you can see the resulting components in the **Application Navigator** as shown in Figure 6–3. You can experiment with the **Flat Level** control and the **Sort by Type** button to see the effect they have on the display.

Figure 6–3 New Entity Objects in Application Navigator Using Flat Level 1 and Sorted By Type

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

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

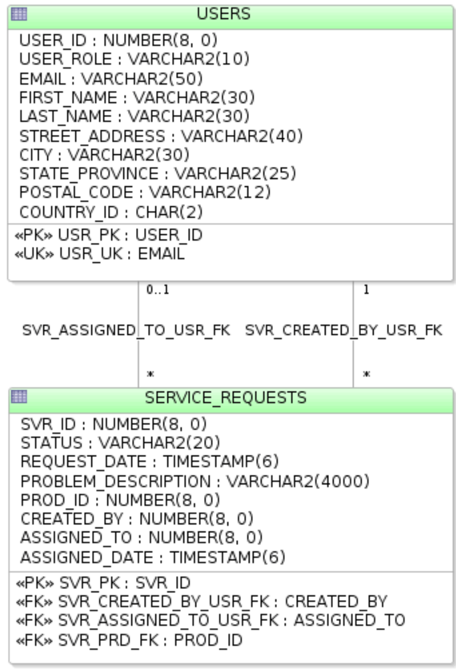
- The Java-friendly entity attribute names from the names of the table's columns (e.g. `USER_ID` -> `UserId`)
- The SQL and Java data types of each attribute based on those of the underlying column
- The length and precision of each attribute
- The primary and unique key attributes
- The mandatory flag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

JDeveloper then creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. One of the entities created above was named `User` in the `devguide.model.entities` package, so the XML file created will be `./devguide/model/entities/User.xml` under the project's source path. This XML file contains the name of the table, the names and data types of each entity attribute, and the column name for each attribute. If you're curious to see its contents, you can see the XML file for the entity object by selecting it in the **Application Navigator** and looking in the corresponding **Sources** folder in the **Structure window**. Double-clicking the `User.xml` node will open the XML in an editor so you can inspect it.

Note: If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom entity object class `UserImpl.java`.

In addition to the entity objects whose names you decided, the wizard also generates named association components that capture information about the relationships between entity objects. A quick glance at the database diagram in Figure 6–4 confirms that the default association names like `SvrAssignedToUsrFkAssoc` and `SvrCreatedByUsrFkAssoc` are derived by converting the foreign key constraint names to a Java-friendly name and adding the `Assoc` suffix. For each association created, JDeveloper creates an appropriate XML component definition file and saves it in the directory that corresponds to the name of its package. By default the associations reverse-engineered from foreign keys get created in the same package as the entities, so, for example, one of the association XML file will be named `./devguide/model/entities/SvrAssignedToUsrFkAssoc.xml`.

Figure 6–4 USERS and SERVICE_REQUESTS Tables Are Related By Foreign Keys



6.2.2.1 What Happens When a Table Has No Primary Key

If a table has no primary key constraint, then JDeveloper cannot infer the primary key for the entity object. Since every entity object must have at least one attribute marked as a primary key, the wizard will create an attribute named `RowID` and use the database `ROWID` value as the primary key for the entity. If appropriate, you can edit the entity object later to mark a different attribute as a primary key and remove the `RowID` attribute. If you use the **Create Entity Object wizard**, you will be prompted to use `RowID` as the primary key, if you have not set any other attribute as primary key.

6.2.3 Creating Entity Objects Using the Create Entity Wizard

To create a single entity object, you can use the **Create Entity Object wizard**. The wizard is available from the **New Gallery** in the **Business Tier > ADF Business Components** category. By selecting the name of an existing table in step 1, on the **Name** page, JDeveloper will infer all of the same information for the new entity that it would have done using the **Business Components from Tables wizard**. If you enter a name of a table that does not exist, you will need to define each attribute one by one on the Attributes page of the wizard. You can later create the table manually, or generate it, as described in Section 6.2.6, "Creating Database Tables from Entity Objects".

6.2.4 Creating an Entity Object for a Synonym or View

When creating an entity object using the **Business Components from Tables wizard** or the **Create Entity Object wizard**, an entity object can represent an underlying table, synonym, or view. You've seen in Section 6.2.2.1, "What Happens When a Table Has No Primary Key", that the framework can infer the primary key and related associations by inspecting database primary and foreign key constraints in the data dictionary. However, if the schema object you select is a database view then neither the primary key, nor associations can be inferred since database views do not have database constraints. In this case, if you use the **Business Components from Tables wizard**, the primary key defaults to `RowID`. If you use the **Create Entity Object wizard**, you'll need to specify the primary key manually by marking at least one of its attributes as a primary key.

If the schema object you choose is a synonym, then there are two possible outcomes. If the synonym is a synonym for a table, then the wizard and editor will behave as if you had specified a table. If instead the synonym refers to a database view, then they will behave as if you had specified the a view.

6.2.5 Editing an Existing Entity Object or Association

After you've created a new entity object, you can edit any of its settings by using the Entity Object Editor. Select the **Edit** menu option on the context menu in the Application Navigator, or double-click on the entity object, to launch the editor. By opening the different panels of the editor, you can adjust the settings that define the entity and govern its runtime behavior. Later sections of this chapter cover many of these settings.

6.2.6 Creating Database Tables from Entity Objects

To create database tables based on entity objects, select the package in the **Application Navigator** that contains the entity objects and choose **Create Database Objects...** from the context menu. A dialog appears to let you select the entities whose tables you'd like to create. This tool can be used to generate a table for an entity object you created from scratch, or to drop and re-create an existing table.

Caution: This feature does *not* generate a DDL script to run later; it performs its operations directly against the database and will drop existing tables. A dialog appears to confirm that you want to do this before proceeding. For entities based on existing tables, use with prudence.

6.2.6.1 Using Database Key Constraints for an Association

In the Association editor, the **Use Database Key Constraints** checkbox on the **Association Properties** page controls whether the related foreign key constraint will be generated when creating the tables for entity objects. Selecting this option does not have any runtime implications.

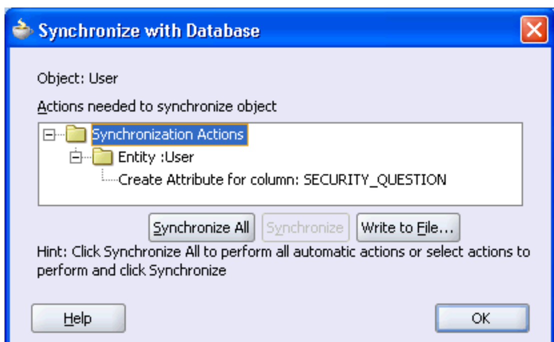
6.2.7 Synchronizing an Entity with Changes to Its Database Table

Inevitably you (or your DBA) might alter a table for which you've already created an entity object. Your existing entity will not be disturbed by the presence of additional attributes in its underlying table; however, if you want to access the new column in the table in your J2EE application, you'll need to synchronize the entity object with the database table. To perform this synchronization automatically, select the entity object in question and choose **Synchronize with Database...** from the context menu. For example, suppose you had done the following at the SQL*Plus command prompt to add a new `SECURITY_QUESTION` column to the `USERS` table:

```
ALTER TABLE USERS ADD (security_question VARCHAR2(60));
```

After you use the synchronize feature on the existing `User` entity, the **Synchronize with Database** dialog would appear as shown in Figure 6–5

Figure 6–5 Synchronizing an Entity Object with Its Underlying Table



The dialog proposes the changes that it can perform for you automatically, and by clicking the desired synchronize button you can carry out the synchronization.

6.2.8 What You May Need to Know About Creating Entities

The Business Components from Tables wizard makes it easy to quickly generate a lot of business components at the same time. In practice, this does not mean that you should use it to immediately create entity objects for every table in your database schema just because you can. If your application will be using all of those tables, that may be appropriate, but since you can use the wizard whenever needed, Oracle recommends creating the entity objects for the tables you know will be involved in the application. Section 8.9, "Deciding on the Granularity of Application Modules" outlines some thoughts on use case-driven design for your business services that can assist you in understanding which entity objects are required to support your application's business logic needs. You can always add more entity objects later as necessary.

6.3 Creating and Configuring Associations

If your database tables have no foreign key constraints defined, JDeveloper won't be able to automatically infer the associations between the entity objects that you create. Since several interesting runtime features that you'll learn about depend on the presence of entity associations, Oracle recommends that you create them manually.

6.3.1 How to Create an Association

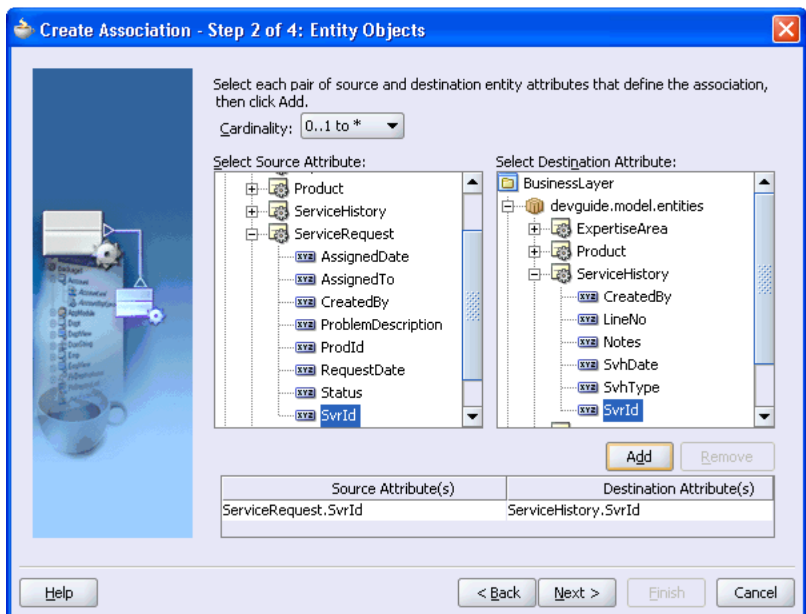
To create an association, use the **Create New Association wizard**.

Assuming the association between the `ServiceRequest` and the `ServiceHistory` entities did not already exist, you could create it manually following these steps:

To create an association:

1. Open the **Create New Association wizard** from the **New Gallery** in the **Business Tier > ADF Business Components** category.
2. In step 1 on the **Name** page, provide a name for the association component and a package to contain it.
3. In step 2, on the **Entity Objects** page, select a "source" attribute from one of the entity objects that will be involved in the association to act as the master. Figure 6–6 shows the selected `SvrId` attribute from the `ServiceRequest` entity object as the source entity attribute.

Figure 6–6 Manually Defining the Attribute Pairs That Relate Two Entity Objects



4. Next, select a corresponding destination attribute from the other entity object involved in the association. Since `ServiceHistory` rows contain a service request ID that relates them to a specific `ServiceRequest` row, select this `SvrId` foreign key attribute in the `ServiceHistory` entity object as the destination attribute.

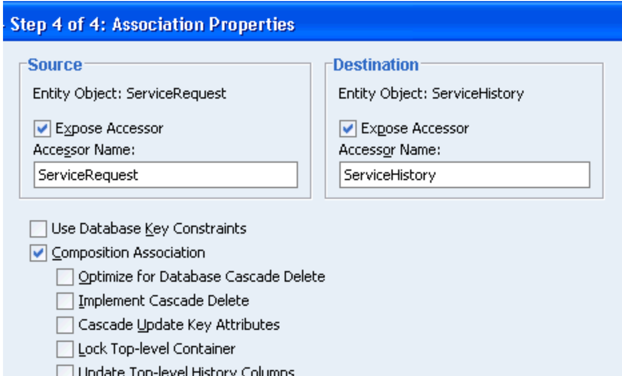
5. Next, click **Add** to add the matching attribute pair to the table of source and destination attribute pairs below. If there were multiple attribute pairs required to define the association, you could repeat these steps to add additional source/target attribute pairs. For this example, the one (`SvrId`,`SvrId`) pair is all that's required.

6. Finally, ensure that the **Cardinality** dropdown correctly reflects the cardinality of the association. Since the relationship between a `ServiceRequest` and its related `ServiceHistory` rows is one-to-many, you can leave the default setting.

7. In step 3, on the **Association SQL** page, you can preview the association SQL predicate that will be used at runtime to access the related `ServiceHistory` entity objects for a given instance of the `ServiceRequest` entity object.

8. In step 4, on the **Association Properties** page, you control whether the association represents a one-way relationship or a bidirectional one and set other properties that define the runtime behavior of the association. Notice in Figure 6–7 that the **Expose Accessor** checkbox is checked in both the **Source** and **Destination** group boxes. By default, an association is a bi-directional relationship allowing either entity object to access the related entity row(s) on the other side when needed. In this example, it means that if you are working with an instance of a `ServiceRequest` entity object, you can easily access the collection of its related `ServiceHistory` rows. You can also easily access the `ServiceRequest` to which it belongs, with any instance of a `ServiceHistory` entity object. Bidirectional navigation is more convenient for writing business validation logic, so in practice, you will typically leave these default checkbox settings.

Figure 6–7 Association Properties Control Runtime Behavior



6.3.1.1 Changing Entity Association Accessor Names

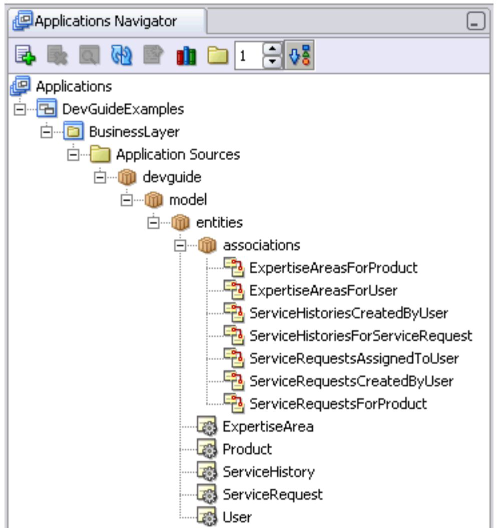
You should consider the default settings for the accessor names on the **Association Properties** page and decide whether changing the names to something more intuitive is appropriate. These define the names of the accessor attributes you will use at runtime to programmatically access the entities on the other side of the relationship. By default, the accessor names will be the names of the entity object on the other side. Since the accessor names on an entity must be unique among entity object attributes and other accessors, if one entity is related to another entity in *multiple* ways then the default accessor names are modified with a numeric suffix to make the name unique. For example, the `ServiceRequest` entity is associated once to the `User` entity to represent the user who created the request, and a second time to reflect the technician to whom the service request is assigned for resolution. The default accessor names on the `ServiceRequest` entity would be `User` and `User1`. By opening the respective **Association Properties** page for the associations in question, you can rename these accessors to more intuitive names like `CreatedByUser` and `TechnicianAssigned`.

6.3.1.2 Renaming and Moving Associations to a Different Package

Since the names of the default associations are not easy to understand, one of the first tasks you might want to perform after creating entity objects from tables is to rename them to something more meaningful. Furthermore, since associations are a component that you typically configure at the outset of your project and don't really visit frequently thereafter, you might want to move them to a different package so that your entity objects are easier to see. Both renaming components and moving them to a different package is straightforward using JDeveloper's refactoring functionality. To move a set of business components to a different package, select one or more of the components in the **Application Navigator** and choose **Refactor > Move...** from the context menu. To rename a component, select it in the navigator and choose **Refactor > Rename** from the context menu.

When you refactor ADF Business Components, JDeveloper automatically moves any XML and/or Java files related to the components, as well as updating any other components that might reference them. Figure 6–8 shows what the **Application Navigator** would look like after renaming all of the associations and moving them to the `devguide.model.entities.associations` package. While you can refactor the associations into any package names you choose, picking a subpackage like this keeps them logically related to the entities but allows collapsing the package of associations to avoid seeing them when you don't need to.

Figure 6–8 Application Navigator After Association Refactoring



6.3.2 What Happens When You Create an Association

When you create an association, JDeveloper creates an appropriate XML component definition file and saves it in the directory that corresponds to the name of its package. If you created an association named `ServiceHistoriesForServiceRequest` in the `devguide.model.entities.associations` package, then the association XML file will be created in the `./devguide/model/entities/associations` directory with the name `ServiceHistoriesForServiceRequest.xml`. At runtime, the entity object uses the association information to automate working with related sets of entities.

6.3.3 What You May Need to Know About Composition Associations

When you create composition associations, it is useful to know about the kinds of relationships you can represent, and the various options.

Associations between entity objects can represent two styles of relationships depending on whether the source entity:

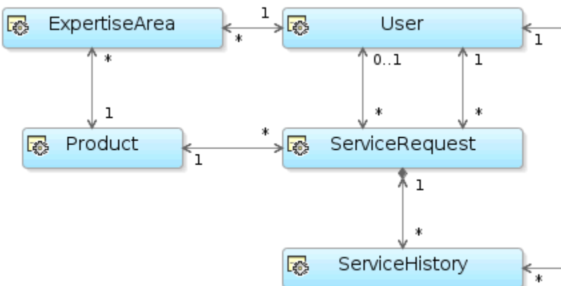
- *References* the destination entity
- *Contains* the destination entity as a logical, nested part

As shown in Figure 6–9, in your SRDemo application business layer you have a `ServiceRequest` that references a `Product`, the requesting `User`, and the assigned `User` (a technician). These relationships represent the first kind of association, reflecting that a `User` or a `Product` entity object can exist independently of a `ServiceRequest`. In addition, the removal of a `ServiceRequest` does not imply the cascade removal of the `Product` to which it was referring or the related `Users`.

In contrast, the relationship between `ServiceRequest` and its collection of related `ServiceHistory` details is stronger than a simple reference. The `ServiceHistory` entries comprise a logical part of the overall `ServiceRequest`. In other words, a `ServiceRequest` is composed of `ServiceHistory` entries. It does not make sense for a `ServiceHistory` entity row to exist independently from a `ServiceRequest`, and when a `ServiceRequest` is removed — assuming it is allowed — all of its composed parts should be removed as well.

This type of logical containership represents the second kind of association, called a *composition*. The UML diagram in Figure 6–9 illustrates the stronger composition relationship using the solid diamond shape on the side of the association which composes the other.

Figure 6–9 ServiceRequest is Composed of ServiceHistory Entries and References Both Product and User



The Business Components from Tables wizard creates composition associations by default for any foreign keys that have the `ON DELETE CASCADE` option. Using the Create Association wizard or the Association Editor, to indicate that an association is a composition association, check the **Composition Association** checkbox on the **Association Properties** page. An entity object offers additional runtime behavior in the presence of a composition. You'll learn the specifics and the settings that control it in Section 6.6.3.12, "Understanding and Configuring Composition Behavior".

6.4 Creating an Entity Diagram for Your Business Layer

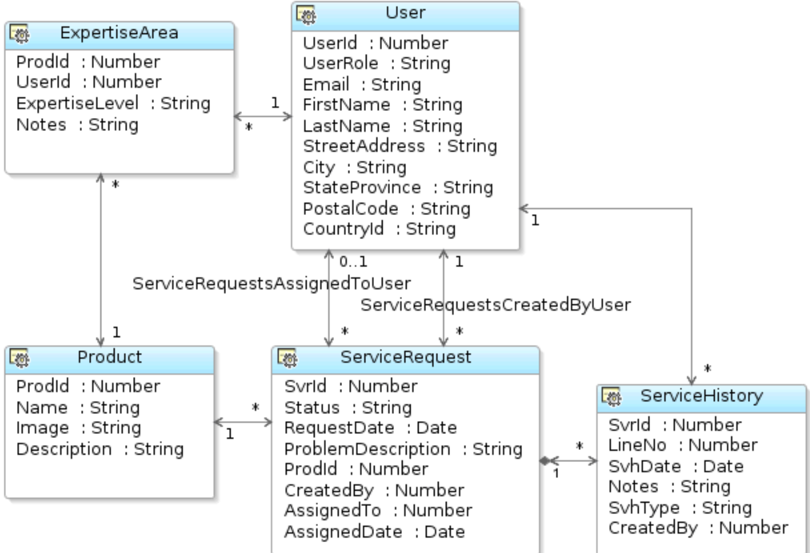
Since your layer of business domain objects represents a key reusable asset to your team, it is often convenient to visualize it using a UML model. JDeveloper supports easily creating a diagram for your business domain layer that you and your colleagues can use for reference.

6.4.1 How to Create an Entity Diagram

To create a diagram of your entity objects, use the **Create Business Components Diagram** dialog. You access it from the **New Gallery** in the **Business Tier > ADF Business Components** category. The dialog prompts you for a diagram name, and a package name in which the diagram will be created. Enter a diagram name like "`Business Domain Objects`" and a name for the package like `devguide.model.design`, and click **OK** to create the empty diagram.

To add your existing entity objects to the diagram, select them all in the **Application Navigator** and drop them onto the diagram surface. Use the property inspector to hide the package name, change the font, turn off the grid and page breaks, and display the name of the two associations that might have been otherwise ambiguous. The diagram should now look like what you see in: Figure 6–10:

Figure 6–10 UML Diagram of Business Domain Layer



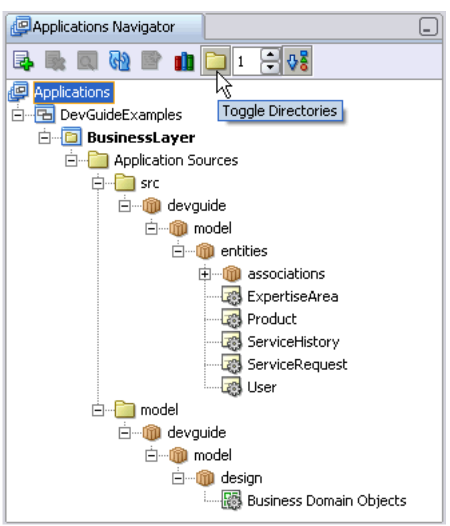
6.4.1.1 Publishing the Business Entity Diagram

To publish the diagram to PNG, JPG, SVG, or compressed SVG format, choose **Publish Diagram...** from the context menu on the diagram surface.

6.4.2 What Happens When You Create an Entity Diagram

When you create a business components diagram, JDeveloper creates an XML file representing the diagram in a subdirectory of the project's model path that matches the package name in which the diagram resides. For the `Business Domain Objects` diagram in Figure 6–10, it would create a matching `*.oxd_bc4j` file in the `./devguide/model/design` subdirectory of the model path. By default, the **Application Navigator** unifies the display of the project contents paths so that ADF components and Java files in the source path appear in the same package tree as the UML model artefacts in the project model path. However, as shown in Figure 6–11, using the **Toggle Directories** toolbar button on the navigator, you can see the distinct project content path root directories when you prefer.

Figure 6–11 Toggling the Display of Separate Content Path Folders



6.4.3 What You May Need to Know About Creating Entities On a Diagram

6.4.3.1 UML Diagram is Actively Synchronized with Business Components

The UML diagram of business components is not just a static picture that reflects the point in time when you dropped the entity objects onto the diagram. Rather, it is a UML-based rendering of the current component definitions, so it will always reflect the current state of affairs. What's more, the UML diagram is both a visualization aid and a visual navigation and editing tool. You can bring up the Entity Object Editor for any entity object in a diagram by selecting **Properties...** from the context menu (or double-clicking). You can also perform some entity object editing tasks directly on the diagram like renaming entities and entity attributes, as well as adding or removing attributes.

6.4.3.2 UML Diagram Adds Extra Metadata to XML Component Descriptors

When you include a business component like an entity object a UML diagram, JDeveloper adds extra metadata to a <Data> section of the component's XML component descriptor as shown in Example 6–1. This is additional information is used at design time only.

Example 6–1 Additional UML Metadata Added to an Entity Object XML Descriptor

```
<Entity Name="ServiceRequest" ... >
   <Data>
      <Property Name ="COMPLETE_LIBRARY" Value ="FALSE" />
      <Property Name ="ID"
                Value ="ff16fca0-0109-1000-80f2-8d9081ce706f::::EntityObject" />
      <Property Name ="IS_ABSTRACT" Value ="FALSE" />
      <Property Name ="IS_ACTIVE" Value ="FALSE" />
      <Property Name ="IS_LEAF" Value ="FALSE" />
      <Property Name ="IS_ROOT" Value ="FALSE" />
      <Property Name ="VISIBILITY" Value ="PUBLIC" />
   </Data>
   :
</Entity>
```

6.5 Defining Attribute Control Hints

With your basic business domain layer of entity objects in place, you can immediately add value by defining UI control hints to ensure that your domain data gets displayed consistently to your end users in locale-sensitive way. JDeveloper manages storing the hints in a way that is easy to localize for multilingual applications. This section explores how to define label text, tooltip, and format mask hints for entity object attributes. As you'll see in Chapter 7, "Building an Updatable Data Model With Entity-Based View Objects", the UI hints you define on your business domain layer are automatically inherited by any entity-based view objects.

6.5.1 How to Add Attribute Control Hints

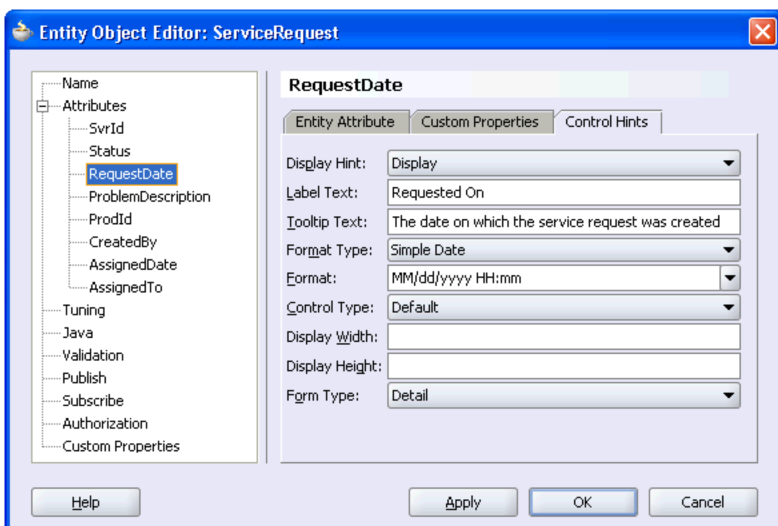
To add attribute control hints to an entity object, open the Entity Object Editor and expand the **Attributes** node in the left-side panel to reveal the list of the entity's attributes. Figure 6–12 shows what this would look like for the `ServiceRequest` entity object. Selecting a particular attribute name like `RequestDate` and selecting the **Control Hints** tab, you can set its:

- **Label Text** hint to "Requested On"
- **Tooltip Text** hint to "The date on which the service request was created"
- **Format Type** to **Simple Date**
- **Format** mask of `MM/dd/yyyy HH:mm`

You can select the other attributes in turn to define appropriate control hints for them as well.

Note: Java defines a standard set of format masks for numbers and dates that are different from those used by the Oracle database's SQL and PL/SQL languages. For reference, see the Javadoc for the `java.text.DecimalFormat` and `java.text.SimpleDateFormat` classes.

Figure 6–12 Setting UI Control Hints for Label for Format Mask for Entity Object Attributes



6.5.2 What Happens When You Add Attribute Control Hints

When you define attribute control hints for an entity object, JDeveloper creates a standard Java message bundle file in which to store them. The file is specific to the entity object component to which its related, and it is named accordingly. For the `ServiceRequest` entity in the `devguide.model.entities` package, the message bundle file created will be named `ServiceRequestImplMsgBundle.java` and it will be created in the `devguide.model.entities.common` subpackage. By selecting the `ServiceRequest` component in the **Application Navigator**, you'll see that this new file gets added to the **Sources** folder in the **Structure window** that shows the group of implementation files for each component. Example 6–2 shows how the control hint information appears in the message bundle file. The first entry in each `String` array is a message key, the second entry is the locale-specific `String` value corresponding to that key.

Example 6–2 Entity Object Component Message Bundle Class Stores Locale-Sensitive Control Hints

```
package devguide.model.entities.common;
import oracle.jbo.common.JboResourceBundle;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---------------------------------------------------------------------
public class ServiceRequestImplMsgBundle extends JboResourceBundle {
  static final Object[][] sMessageStrings = {
  { "AssignedDate_FMT_FORMAT", "MM/dd/yyyy HH:mm" },
  { "AssignedDate_FMT_FORMATTER", "oracle.jbo.format.DefaultDateFormatter" },
  { "AssignedDate_LABEL", "Assigned On" },
  { "AssignedTo_LABEL", "Assigned To" },
  { "CreatedBy_LABEL", "Requested By" },
  { "ProblemDescription_DISPLAYWIDTH", "60" },
  { "ProblemDescription_LABEL", "Problem" },
  { "RequestDate_FMT_FORMAT", "MM/dd/yyyy HH:mm" },
  { "RequestDate_FMT_FORMATTER", "oracle.jbo.format.DefaultDateFormatter" },
  { "RequestDate_LABEL", "Requested On" },
  { "RequestDate_TOOLTIP",
    "The date on which the service request was created" },
  { "Status_LABEL", "Status" },
  { "SvrId_LABEL", "Request" }
  };
// etc.
```

6.5.3 Internationalizing the Date Format

Internationalizing the model layer of an application built using ADF Business Components entails producing translated versions of each component message bundle file. For example, the Italian version of the `ServiceRequestImplMsgBundle` message bundle would be a class named `ServiceRequestImplMsgBundle_it` and a more specific *Swiss* Italian version would have the name `ServiceRequestImplMsgBundle_it_ch`. These classes typically extend the base message bundle class, and contain entries for the message keys that need to be localized, together with their localized translation.

For example, the Italian version of the `ServiceRequest` entity object message bundle would look like what you see in Example 6–3. Notice that in the Italian translation, the format masks for the `RequestDate` and `AssignedDate` have been changed to `dd/MM/yyyy HH:mm`. This ensures that an Italian user will see a date value like May 3rd, 2006, as `03/05/2006 15:55`, instead of `05/03/2006 15:55`, which the format mask in the default message bundle would produce. Notice the overridden `getContents()` method. It returns an array of messages with the more *specific* translated strings merged together with those that are not overridden from the superclass bundle. At runtime, the appropriate message bundles are used automatically, based on the current user's locale settings.

Example 6–3 Localized Entity Object Component Message Bundle for Italian

```
package devguide.model.entities.common;
import oracle.jbo.common.JboResourceBundle;
public class ServiceRequestImplMsgBundle_it
       extends ServiceRequestImplMsgBundle {
  static final Object[][] sMessageStrings = {
    { "AssignedDate_FMT_FORMAT", "dd/MM/yyyy HH:mm" },
    { "AssignedDate_LABEL", "Assegnato il" },
    { "AssignedTo_LABEL", "Assegnato a" },
    { "CreatedBy_LABEL", "Aperto da" },
    { "ProblemDescription_LABEL", "Problema" },
    { "RequestDate_FMT_FORMAT", "dd/MM/yyyy HH:mm" },
    { "RequestDate_LABEL", "Aperto il" },
    { "RequestDate_TOOLTIP", "La data in cui il ticket è stato aperto" },
    { "Status_LABEL", "Stato" },
    { "SvrId_LABEL", "Ticket" }
  };
  public Object[][] getContents() {
    return super.getMergedArray(sMessageStrings, super.getContents());
  }
}
```

6.6 Configuring Declarative Runtime Behavior

The entity object offers numerous declarative features to simplify implementing typical enterprise business applications. Depending on your task, sometimes the declarative facilities *alone* may satisfy your needs. However, when you need to go beyond the declarative behavior to implement more complex business logic or validation rules for your business domain layer, that is possible as well. This chapter focuses on giving you a solid understanding of the declarative features. In Chapter 9, "Implementing Programmatic Business Rules in Entity Objects", you'll study some of the most typical ways that you extend entity objects with custom code.

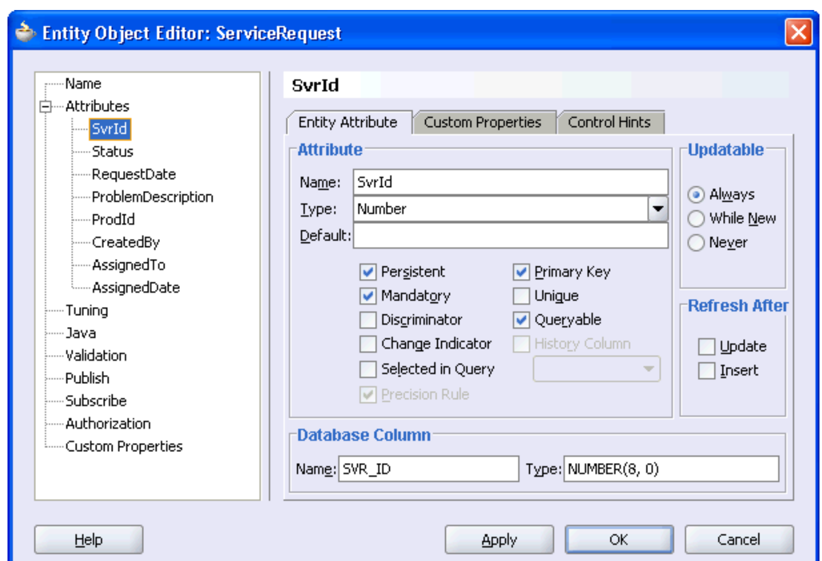
6.6.1 How To Configure Declarative Runtime Behavior

To configure the declarative runtime behavior of an entity object, use the Entity Object Editor. You access the editor by selecting an entity in the Application Navigator and choosing **Edit** from the context menu. Figure 6–13 shows what the editor looks like for the `ServiceRequest` entity object.

On the **Name** page, you can see the entity object's name and configure the database table to which it relates. On the **Attributes** page, you create or delete the attributes that represent the data relevant to an entity object. By expanding this node, you can access the properties of each of the entity object's attributes.

On the **Tuning** page, you set options to make database operations more efficient when you create, modify, or delete multiple entities of the same type in a single transaction. On the **Publish** page, you define events that your entity object can use to notify others of interesting changes in its state, optionally including some or all of the entity object's attributes in the delivered event. On the **Subscribe** page, you enroll your entity object to be notified when selected events of other entity objects fire. On the **Authorization** page, you define role-based updatability permissions for any or all attributes. And finally, on the **Custom Properties** page, you can define custom metadata you can access at runtime on the entity.

Figure 6–13 Use the Entity Object Editor to Configure Its Declarative Features



Note: If your entity has a long list of attribute names, there's a quick way to find the one you're looking for. With the **Attributes** node in the tree expanded, you can begin to type the letters of the attribute name and JDeveloper performs an incremental search to take you to its name in the tree.

6.6.2 What Happens When You Configure Declarative Runtime Behavior

All of the declarative settings that describe and control an entity object's runtime behavior are stored in its XML component definition file. When you modify settings of your entity using the editor, pressing **OK** updates the component's XML definition file and optional custom java files. If you need to immediately apply changes and continue working in the editor, use the **Apply** button. Applying changes while editing is typically useful only when you enable the generation of a custom Java file for the component for the first time and you want JDeveloper to generate those files before you open another page in the editor.

6.6.3 About the Declarative Entity Object Features

Since much of the entity object's declarative functionality is related to the settings of its attributes, this section covers the important options shown in Figure 6–13 in detail.

6.6.3.1 Legal Database and Java Data types for an Entity Object Attribute

The **Persistent** property controls whether the attribute value corresponds to a column in the underlying table, or whether it is just a transient value. If the attribute is persistent, the **Database Column** area lets you change the name of the underlying column that corresponds to the attribute and indicate its column type with precision and scale information (e.g. `VARCHAR2(40)` or `NUMBER(4,2)`). Based on this information, at runtime the entity object enforces the maximum length and precision/scale of the attribute value, throwing an exception if a value does not meet the requirements.

Both the Business Components from Tables wizard and the Create Entity Object wizard automatically infer the Java type of each entity object attribute from the SQL type of the database column type of the column to which it is related. The **Attribute Type** field allows you to change the Java type of the entity attribute to any type you might need. The **Database Column Type** field reflects the SQL type of the underlying database column to which the attribute is mapped. The value of the **Database Column Name** field controls the column to which the attribute is mapped.

Your entity object can handle tables with column types, as listed in Table 6–1. With the exception of the `java.lang.String` class, the default Java attribute types are all in the `oracle.jbo.domain` and `oracle.ord.im` packages and support efficiently working with Oracle database data of the corresponding type. The dropdown list for the Java **Type** field includes a number of other common types that are also supported.

Table 6–1 Default Entity Object Attribute Type Mappings

Oracle Column Type	Entity Column Type	Entity Java Type
NVARCHAR2(n), VARCHAR2(n),NCHAR VARYING(n),VARCHAR(n)	VARCHAR2	String
NUMBER	NUMBER	Number
DATE	DATE	Date
TIMESTAMP(n), TIMESTAMP(n) WITH TIME ZONE,TIMESTAMP(n) WITH LOCAL TIME ZONE	TIMESTAMP	Date
LONG	LONG	String
RAW(n)	RAW	Raw
LONG RAW	LONG RAW	Raw
ROWID	ROWID	RowID
NCHAR, CHAR	CHAR	String
NCLOB, CLOB	CLOB	ClobDomain
BLOB	BLOB	BlobDomain
BFILE	BFILE	BFileDomain
ORDSYS.ORDIMAGE	ORDSYS.ORDIMAGE	OrdImageDomain
ORDSYS.ORDVIDEO	ORDSYS.ORDVIDEO	OrdVideoDomain

Table 6–1 (Cont.) Default Entity Object Attribute Type Mappings

Oracle Column Type	Entity Column Type	Entity Java Type
ORDSYS.ORDAUDIO	ORDSYS.ORDAUDIO	OrdAudioDomain
ORDSYS.ORDDOC	ORDSYS.ORDDOC	OrdDocDomain

Note: In addition to the types mentioned here, you can use any Java object type as an entity object attribute's type, provided it implements the `java.io.Serializable` interface.

6.6.3.2 Indicating Datatype Length, Precision, and Scale

When working with types that support defining a maximum length like `VARCHAR2(n)`, the **Database Column Type** field includes the maximum attribute length as part of the value. So, for example, an attribute based on a `VARCHAR2(10)` column in the database will initially reflect the maximum length of 10 characters by showing `VARCHAR2(10)` as the **Database Column Type**. If for some reason you want to restrict the maximum length of the `String`-valued attribute to fewer characters than the underlying column will allow, just change the maximum length of the **Database Column Type** value. For example, the `EMAIL` column in the `USERS` table is `VARCHAR2(50)`, so by default the `Email` attribute in the `Users` entity object defaults to the same. If you know that the actual email addresses are always 8 characters or less, you can update the database column type for the `Email` attribute to be `VARCHAR2(8)` to enforce a maximum length of 8 characters at the entity object level.

The same holds for attributes related to database column types that support defining a precision and scale like `NUMBER(p[,s])`. So, for example, to restrict an attribute based on a `NUMBER(7,2)` column in the database to have a precision of 5 and a scale of 1 instead, just update the **Database Column Type** to be `NUMBER(5,1)`.

6.6.3.3 Controlling the Updatability of an Attribute

The Updatable setting controls when the value of a given attribute can be updated. If set to:

- **Always,** the attribute is always updatable
- **Never**, the attribute is read-only
- **While New**, the attribute can be set during the transaction that creates the entity row for the first time, but after being successfully committed to the database the attribute is read-only

6.6.3.4 Making an Attribute Mandatory

The **Mandatory** property controls whether the field is required.

6.6.3.5 Defining the Primary Key for the Entity

The **Primary Key** property indicates whether the attribute is part of the key that uniquely identifies the entity. Typically, you will use a single attribute for the primary key, but multiattribute primary keys are fully supported.

At runtime, when you access the related `Key` object for any entity row using the `getKey()` method, this `Key` object contains the value(s) of the primary key attribute(s) for the entity object. If your entity object has multiple primary key attributes, the `Key` object contains each of their values. It is important to understand that these values appear in the same *relative* sequential order as the corresponding primary key attributes in the entity object definition.

For example, the `ServiceHistory` entity object has multiple primary key attributes `SvrId` and `LineNo`. On the **Attributes** page of the Entity Object Editor, `SvrId` is first, and `LineNo` is second; an array of values encapsulated by the `Key` object for a entity row of type `ServiceHistory` will have these two attribute values in exactly this order. The reason why it is crucial to understand this point is that if you try to use `findByPrimaryKey()` to find an entity with a multiattribute primary key, and the Key object you construct has these multiple primary key attributes in the wrong order, the entity row will not be found as expected.

6.6.3.6 Defining a Static Default Value

The **Default** field specifies a static default value for the attribute. For example, you might set the default value of the `ServiceRequest` entity object's `Status` attribute to `Open`, or set the default value of the `User` entity object's `UserRole` attribute to `user`.

6.6.3.7 Synchronization with Trigger-Assigned Values

If you know that the underlying column value will be updated by a database trigger during insert or update operations, you can check the respective **Refresh After Insert** or **Refresh After Update** checkboxes to have the framework automatically retrieve the modified value to keep the entity object and database row in sync. The entity object uses the Oracle SQL `RETURNING INTO` feature, while performing the `INSERT` or `UPDATE` to return the modified column back to your application in a single database round-trip.

Note: If you create an entity object for a synonym that resolves to a remote table over a DBLINK, use of this feature will give an error at runtime like:

```
JBO-26041: Failed to post data to database during "Update"
## Detail 0 ##
ORA-22816: unsupported feature with RETURNING clause
```

Section 26.5, "Basing an Entity Object on a Join View or Remote DBLink" describes a technique to circumvent this database limitation.

6.6.3.8 Trigger-Assigned Primary Key Values from a Database Sequence

One common case where **Refresh After Insert** comes into play is a primary key attribute whose value is assigned by a `BEFORE INSERT FOR EACH ROW` trigger. Often the trigger assigns the primary key from a database sequence using PL/SQL logic similar to this:

```
CREATE OR REPLACE TRIGGER ASSIGN_SVR_ID
BEFORE INSERT ON SERVICE_REQUESTS FOR EACH ROW
BEGIN
 IF :NEW.SVR_ID IS NULL OR :NEW.SVR_ID < 0 THEN
   SELECT SERVICE_REQUESTS_SEQ.NEXTVAL
     INTO :NEW.SVR_ID
     FROM DUAL;
   END IF;
END;
```

Set the **Attribute Type** to the built-in datatype named `DBSequence`, as shown in Figure 6–14, and the primary key will be assigned automatically by the database sequence. Setting this datatype automatically enables the **Refresh After Insert** property.

When you create a new entity row whose primary key is a `DBSequence`, a unique negative number gets assigned as its temporary value. This value acts as the primary key for the duration of the transaction in which it is created. If you are creating a set of interrelated entities in the same transaction, you can assign this temporary value as a foreign key value on other new, related entity rows. At transaction commit time, the entity object issues its `INSERT` operation using the `RETURNING INTO` clause to retrieve the actual database trigger-assigned primary key value. Any related new entities that previously used the temporary negative value as a foreign key will get that value updated to reflect the actual new primary key of the master.

Note: As shown in Figure 6–14, you will typically also set the **Updatable** property of a DBSequence-valued primary key to **Never**. The entity object assigns the temporary ID, and then refreshes it with the actual ID value after the INSERT option. The end user never needs to update this value.

Figure 6–14 Setting Primary Key Attribute to DBSequence Type Automates Trigger-Assigned Key Handling

---

Note: The sequence name shown on the **Sequence** tab only comes into play at design time when you use the **Create Database Tables...** feature described in Section 6.2.6, "Creating Database Tables from Entity Objects". The sequence indicated here will be created along with the table on which the entity object is based.

6.6.3.9 Lost Update Protection

At runtime the framework provides automatic "lost update" detection for entity objects to ensure that a user cannot unknowingly modify data that another user has updated and committed in the meantime. Typically this check is performed by comparing the original values of each persistent entity attribute against the corresponding current column values in the database at the time the underlying row is locked. If an entity object detects that it would be updating a row that is now inconsistent with the current state of the database, it raises the `RowInconsistentException`.

You can make the lost update detection more efficient by identifying an attribute of your entity whose value you know will get updated whenever the entity gets modified. Typical candidates include a version number column or an updated date column in the row. The change indicator attribute value might be assigned by a database trigger you've written and refreshed in the entity object using the **Refresh After Insert** and **Refresh After Update** properties. Alternatively, you can indicate that the entity object should manage updating the change indicate attribute's value using the history attribute feature described in the next section. To detect whether the row has been modified since the user queried it in the most efficient way, select the **Change Indicator** to compare only the change indicator attribute values.

6.6.3.10 History Attributes

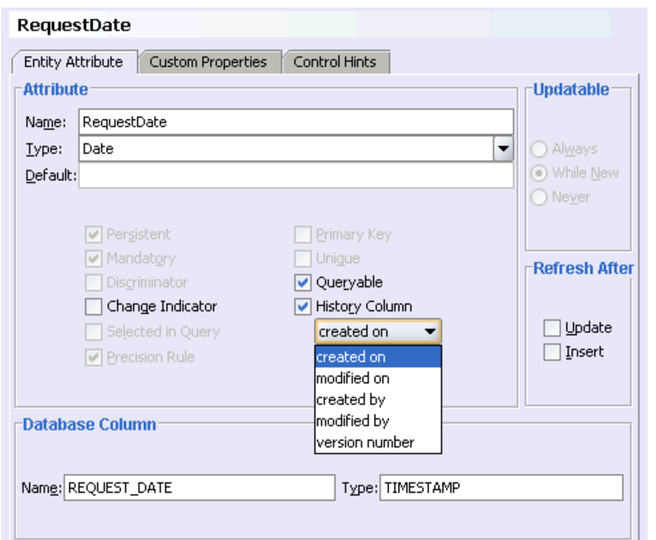
Frequently, you'll need to keep track of historical information in your entity object like:

- Who created this entity?
- When did they create it?
- Who last modified this entity?
- When did they modify it?
- How many times has this row been modified?

Entity objects store this information in a **History Column** attribute, as shown in Figure 6–15.

If an attribute's datatype is `Number`, `String`, or `Date`, and it is not part of the primary key, then you can enable this property to have your entity automatically maintain the attribute's value for historical auditing. How the attribute gets handled depends on the history attribute type that you indicate. If you choose the **Version Number** type of history column, ADF will automatically increment the value of the numeric attribute every time the object is updated. If you choose **Created By**, **Created On**, **Modified By**, or **Modified On**, the value will be updated with the current user's username or the current date, respectively, when the object is created or modified.

Figure 6–15 Defaulting a Date tor the Current Database Time Using a History Attribute



6.6.3.11 Setting the Discriminator Attribute for Entity Object Inheritance Hierarchies

Sometimes a single database table stores information about several different kinds of logically related objects. For example, a payroll application might work with hourly, salaried, and contract employees all stored in a single `EMPLOYEES` table with an `EMPLOYEE_TYPE` column. The value of the `EMPLOYEE_TYPE` column indicates with a value like `H`, `S`, or `C`, whether a given row represents an hourly, salaried, or contract employee respectively. While many employee attributes and behavior are the same for all employees, certain properties and business logic depends on the type of employee. In this situation it can be convenient to represent these different types of employees using an inheritance hierarchy. Attributes and methods common to all employees can be part of a base `Employee` entity object, while subtype entity objects like `HourlyEmployee`, `SalariedEmployee`, and `ContractEmployee` extend the base `Employee` object and add additional properties and behavior. The **Discriminator** attribute property is used to indicate which attribute's value distinguishes the type of row. Section 26.6, "Using Inheritance in Your Business Domain Layer" explains how to set up and use inheritance.

6.6.3.12 Understanding and Configuring Composition Behavior

When an entity object composes other entities, it exhibits additional runtime behavior to correctly play its role as a logical container of other nested entity object parts. The following features are always enabled for composing entity objects:

6.6.3.12.1 Orphan-row Protection for New Composed Entities When a composed entity object is created, it performs an existence check on the value of its foreign key attribute to ensure that it identifies an existing entity as its owning parent entity. Failure to provide a value for the foreign key at create time, or providing a value that does not identify an existing entity object, throws an `InvalidOwnerException` instead of allowing an "orphaned" child row to be created with no well-identified parent entity.

Note: The existence check performed finds new pending entities in the current transaction, as well as existing ones in the database if necessary.

6.6.3.12.2 Ordering of Changes Saved to the Database

This feature ensures that the sequence of DML operations performed in a transaction involving both composing and composed entity objects is performed in the correct order. For example, an `INSERT` statement for a new composing parent entity object will be performed before the DML operations related to any composed children.

6.6.3.12.3 Cascade Update of Composed Details from Refresh-On-Insert Primary Keys

When a new entity row having a **Refresh On Insert** primary key is saved, after its trigger-assigned primary value is retrieved, any composed entities will automatically have their foreign key attribute values updated to reflect the new primary key value.

There are a number of additional composition related features that you can control through settings on the **Association Properties** page of the Create Association wizard or the Association Editor. Figure 6–16 shows this page for the `ServiceHistoriesForServiceRequest` association between the `ServiceRequest` and `ServiceHistory` entity objects. These settings are the defaults that result from reverse-engineering the composition from an `ON DELETE CASCADE` foreign key constraint in the database.

Figure 6–16 Composition Settings for ServiceHistoriesForServiceRequest Association

The additional features, and the properties that affect their behavior, include the following:

6.6.3.12.4 Cascade Delete Support

You can either enable or prevent the deletion of a composing parent while composed children entities exist. When the **Implement Cascade Delete** is unchecked, the removal of the composing entity object is prevented if it contains any composed children. When checked, this option allows the composing entity object to be removed unconditionally and composed children entities are also removed. If the related **Optimize for Database Cascade Delete** option is unchecked, then the composed entity objects perform their normal `DELETE` statement at transaction commit time to make the changes permanent. If the option is checked, then the composed entities do not perform the `DELETE` statement on the assumption that the database `ON DELETE CASCADE` constraint will handle the deletion of the corresponding rows.

6.6.3.12.5 Cascade Update of Foreign Key Attributes When Primary Key Changes

By checking the **Cascade Update Key Attributes** option, you can enable the automatic update of the foreign key attribute values in composed entities when the primary key value of the composing entity is changed.

6.6.3.12.6 Locking of Composite Parent Entities

Using the **Lock Top-Level Container** option, you can control whether adding, removing, or modifying a composed detail entity row should attempt to lock the composing entity before allowing the changes to be saved.

6.6.3.12.7 Updating of Composing Parent History Attributes

Using the **Update Top-Level History Columns** option, you can control whether adding, removing, or modifying a composed detail entity object should update the **Modified By** and **Modified On** history attributes of the composing parent entity.

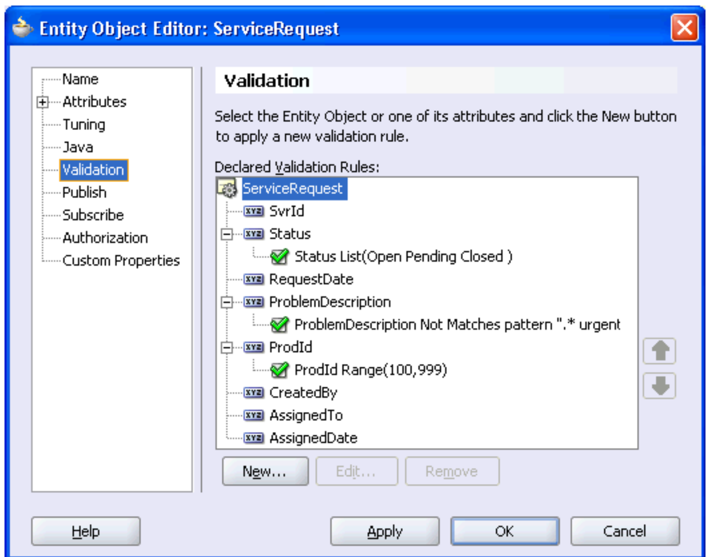
6.7 Using Declarative Validation Rules

One page of the Entity Object Editor worthy of special attention is the **Validation** page, where you can see and manage the declarative validation rules for the entity or any of its attributes. The framework enforces entity-level validation rules when a user tries to commit pending changes or simply navigates between rows. Attribute-level validation rules are enforced when the user changes the value of the related attribute. When you add a validation rule, you supply an appropriate error message and can later translate it easily into other languages if needed. Oracle ADF ships with a number of built-in declarative validation rules that you'll see in this section. Section 9.3, "Using Method Validators" explains how to use the **Method Validator** to invoke custom validation code and in Section 26.9, "Implementing Custom Validation Rules" you'll learn how to extend the basic set of declarative rules with custom rules of your own.

6.7.1 How to Add a Validation Rule

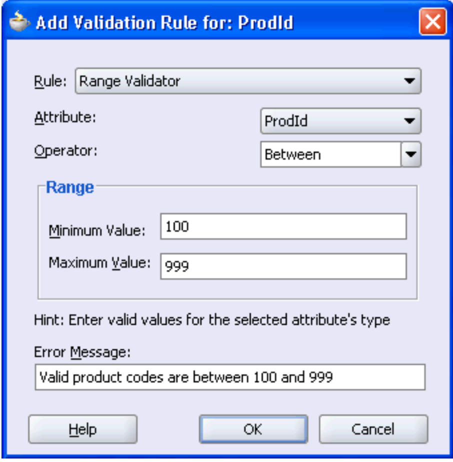
To add a validation rule to an entity object, use the **Validation** page of the Entity Object Editor, as shown in Figure 6–17. To add an attribute-level validation rule, select the attribute in the **Declared Validation Rules** tree, and click **New...**. Defining an entity-level validation rule is similar, except that you select the root entity object node in the tree before clicking **New...**.

Figure 6–17 Validation Page of the Entity Object Editor



When you add a new validation rule, the **Add Validation Rule** dialog appears. Use the **Rule** dropdown list to select the kind of validation rule you want, and configure its declarative settings using the other controls in the page. The controls will change depending on the kind of validation rule you select. Figure 6–18 illustrates what the **Add Validation Rule** dialog would look like when defining a range validation rule for the `ProdId` attribute of the `ServiceRequest` entity object. This validation rule has been selected to enforce that the value lie between `100` and `999` inclusive. When you add a validation rule, you also can enter an error message that will be shown to the user if the validation rule fails.

Figure 6–18 Adding a New Range Validation Rule for the ProdId Attribute



6.7.2 What Happens When You Add a Validation Rule

When you add a validation rule to an entity object, JDeveloper updates its XML component definition to include an entry describing what rule you've used and what rule properties you've entered. For example, if you add the range validation rule above to the `ProdId` attribute, this results in a `RangeValidationBean` entry in the XML file:

```
<Entity Name="ServiceRequest"
   <!-- : -->
   <Attribute Name="ProdId" IsNotNull="true" Precision="8" Scale="0"
      ColumnName="PROD_ID" Type="oracle.jbo.domain.Number"
      ColumnType="NUMBER" SQLType="NUMERIC" TableName="SERVICE_REQUESTS" >
      <RangeValidationBean
         xmlns="http://xmlns.oracle.com/adfm/validation"
         ResId="ProdId_Rule_0"
         OnAttribute="ProdId"
         OperandType="LITERAL"
         MinValue="100"
         MaxValue="999" >
      </RangeValidationBean>
   </Attribute>
   <!-- : -->
</Entity>
```

At runtime, the rule is automatically enforced by the entity object based on this declarative information. The error message is a translatable string and is managed in the same way as translatable UI control hints in an entity object message bundle class. The `ResId` property in the XML component definition entry for the validator corresponds to the `String` key in the message bundle. Example 6–4 shows the relevant bit of the `ServiceRequest` entity object's message bundle, where the `ProdId_Rule_0` key appears with the error message for the default locale. The validation errors messages get translated using the same mechanism described above for UI control hints.

Example 6–4 Entity Object Message Bundle Contains Validation Error Messages

```
package devguide.model.entities.common;
import oracle.jbo.common.JboResourceBundle;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---------------------------------------------------------------------
public class ServiceRequestImplMsgBundle extends JboResourceBundle {
  static final Object[][] sMessageStrings = {
  // other strings here
  { "ProdId_Rule_0", "Valid product codes are between 100 and 999" },
  // other strings here
  };
  // etc.
}
```

6.7.3 What You May Need to Know About Validation Rules

It is important to know that some validators can be used at the entity level, and some are used on the attribute level. Also, you should be aware the List Validator is designed for working with a relatively small set.

6.7.3.1 Understanding the Built-in Entity-Level Validators

You can use the following built-in validators at the entity object level:

Unique Key Validator
Validates that the primary key for an entity is unique.

Method Validator
Invokes a method in an entity object's custom Java class to evaluate a programmatic validation.

6.7.3.2 Understanding the Built-in Attribute-Level Validators

You can use the following built-in validators at the entity object attribute level:

Compare Validator
Validates an attribute value against:

- A literal value,
- A selected attribute of the first row of a view object query result, or
- The first column of the first row of a SQL query result

List Validator
Validates that an attribute exists in an *in-memory* set of values from a:

- Static list,
- A selected attribute in the rows of view object's default row set, or
- The first column value in the rows of a SQL query result.

Range Validator
Validates that an attribute lies between a minimum and maximum value, inclusive.

Length Validator
Validates whether the string length of an attribute's value is less than, equal to, or greater than a fixed number of characters.

Regular Expression Validator
Validates that an attribute's value matches a regular expression.

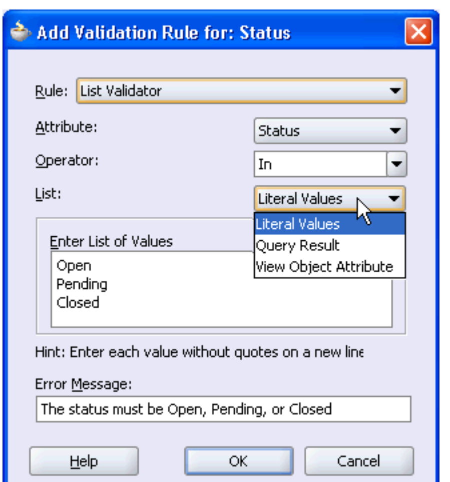
Method Validator
Invokes a method in an entity object's custom Java class to evaluate a programmatic validation.

6.7.3.3 Caveat About the List Validator

The List Validator is designed for validating an attribute against a relatively small set of values. As shown in Figure 6–19, if you select the **Query Result** or **View Object Attribute** style of list validation, keep in mind the validator will retrieve all of the rows from the query before performing an in-memory scan to validate whether the attribute value in question matches an attribute in the list. The query performed by the Validator's SQL or view object query does not reference the value being validated in the `WHERE` clause of the query.

In other words, this is not the feature to use if you want to validate that a user-entered product code exists in a table of a million products. Section 9.6, "Using View Objects for Validation", explains the technique you can use to efficiently perform SQL-based validations by using a view object to perform a targeted validation query against the database.

Figure 6–19 List Validator is Designed for Relatively Small Lists of Values



6.8 Working Programmatically with Entity Objects and Associations

While external client programs can access an application module and work with any view object in its data model, by design neither UI-based nor programmatic clients work directly with entity objects. In Chapter 7, "Building an Updatable Data Model With Entity-Based View Objects", you'll learn how to easily combine the flexible SQL-querying of view objects with the business logic enforcement and automatic database interaction of entity objects for an incredibly powerful application-building combination. The combination enables a fully updatable application module data model, designed to the needs of the current end-user tasks at hand, that shares the centralized business logic in your reusable domain business object layer.

However, it is important first to understand how view objects and entity objects can be used on their own before learning to harness their combined power. By learning about these objects in greater detail, you will have a better understanding of when you should use them alone and when to combine them in your own applications.

Since clients don't work *directly* with entity objects, any code you write that works programmatically with entity objects will typically be custom code in a custom application module class or in the custom class of another entity object. This section illustrates examples of working programmatically with entity objects and associations from within custom methods of an application module named `SRService` in the `devguide.model` package, manipulating the `SRDemo` entities you learned how to create earlier in the chapter.

6.8.1 Finding an Entity Object by Primary Key

To access an entity row, you use a related object called the entity definition. At runtime, each entity object has a corresponding entity definition object that describes the structure of the entity and manages the instances of the entity object it describes. After creating an `SRService` application module in the `devguide.model` package and enabling a custom Java class for it, imagine you wanted to write a method to return a specific service request's current status. It might look like the `retrieveServiceRequestStatus()` method like in the `SRServiceImpl.java` file shown in Example 6–5.

The example breaks down into these basic steps:

1. Find the entity definition.

   You obtain the entity definition object for the `devguide.model.entities.ServiceRequest` entity by passing its fully qualified name to the static `findDefObject()` method on the `EntityDefImpl` class. The `EntityDefImpl` class in the `oracle.jbo.server` package implements the entity definition for each entity object.

2. Construct a key.

   You build a `Key` object containing the primary key attribute that you want to look up. In this case, you're creating a key containing the single `requestId` value passed into the method as an argument.

3. Find the entity object using the key.

   You use the entity definition's `findByPrimaryKey()` method to find the entity object by key, passing in the current transaction object, which you can obtain from the application module using its `getDBTransaction()` method. The concrete class that represents an entity object row is the `oracle.jbo.server.EntityImpl` class.

4. Return some of its data to the caller.

   You use the `getAttribute()` method of `EntityImpl` to return the value of the `Status` attribute to the caller.

Example 6–5 Finding a ServiceRequest Entity Object by Key

```
// Custom method in SRServiceImpl.java
public String findServiceRequestStatus(long requestId) {
  String entityName = "devguide.model.entities.ServiceRequest";
  // 1. Find the entity definition for the ServiceRequest entity
  EntityDefImpl svcReqDef = EntityDefImpl.findDefObject(entityName);
  // 2. Create the key
  Key svcReqKey = new Key(new Object[]{requestId});
  // 3. Find the entity object instance using the key
  EntityImpl svcReq = svcReqDef.findByPrimaryKey(getDBTransaction(),svcReqKey);
  if (svcReq != null) {
    // 4. Return the Status attribute of the ServiceRequest
    return (String)svcReq.getAttribute("Status");
  }
  else {
    return null;
  }
}
```

> **Note:** The `oracle.jbo.Key` object constructor takes an *Object* array to support creating multiattribute keys, in addition to the more typical single-attribute value keys.

6.8.2 Accessing an Associated Entity Using the Accessor Attribute

In Section 6.3, "Creating and Configuring Associations", you learned that associations enable easy access from one entity object to another. Here's a simple method that helps illustrate what that means in practice. You can add a `findServiceRequestTechnician()` method that finds a service request, then accesses the associated `User` entity object representing the technician assigned to the request.

However, since this is the second method in the application module that will be finding a ServiceRequest entity object by ID, you might first want to refactor this functionality into the following retrieveServiceRequestById() helper method that you can then reuse anywhere in the application module that requires finding a service request by ID:

```
// Helper method in SRServiceImpl.java
private EntityImpl retrieveServiceRequestById(long requestId) {
  String entityName = "devguide.model.entities.ServiceRequest";
  EntityDefImpl svcReqDef = EntityDefImpl.findDefObject(entityName);
  Key svcReqKey = new Key(new Object[]{requestId});
  return svcReqDef.findByPrimaryKey(getDBTransaction(),svcReqKey);
}
```

Example 6–6 shows the code for findServiceRequestTechnician(). The example follows three basic steps:

1. Find the ServiceRequest by ID.

   Using the retrieveServiceRequestById() helper method, it retrieves the ServiceRequest entity object by ID.

2. Access the associated entity using the accessor attribute.

   In Section 6.3.1.1, "Changing Entity Association Accessor Names" above, you renamed the association accessor for the ServiceRequestsAssignedToUser association so that a ServiceRequest entity could access one of its two related User entity objects with the accessor name of TechnicianAssigned. Using the same getAttribute() method used to retrieve any entity attribute value, you can pass in the name of an association accessor and get back the entity object on the other side of the relationship.

3. Return some of its data to the caller.

   Using the getAttribute() method on the returned User entity, it returns the assigned technician's name by concatenation his first and last names.

Notice that you did not need to write any SQL to access the related User entity. The relationship information captured in the ADF association between the ServiceRequest and User entity objects is enough to allow the common task of data navigation to be automated.

Example 6–6 Accessing an Associated Entity Using the Accessor Attribute

```
// Custom method in SRServiceImpl.java
public String findServiceRequestTechnician(long requestId) {
  // 1. Find the service request entity
  EntityImpl svcReq = retrieveServiceRequestById(requestId);
  if (svcReq != null) {
    // 2. Access the User entity object using the association accessor attribute
    EntityImpl tech  = (EntityImpl)svcReq.getAttribute("TechnicianAssigned");
    if (tech != null) {
      // 3. Return some of the User entity object's attributes to the caller
      return tech.getAttribute("FirstName")+" "+tech.getAttribute("LastName");
    }
    else {
      return "Unassigned";
    }
  }
  else {
    return null;
  }
}
```

6.8.3 Updating or Removing an Existing Entity Row

Once you've got an entity row in hand, it's simple to update it or remove it. You could add a method like the `updateRequestStatus()` shown in Example 6–7 to handle the job. The example has three simple steps:

1. Find the ServiceRequest by ID

   Use the `retrieveServiceRequestById()` helper method to retrieve the `ServiceRequest` entity object by ID.

2. Set one or more attributes to new values.

   Use the `EntityImpl` class' `setAttribute()` method to update the value of the `Status` attribute to the new value passed in.

3. Commit the transaction.

   Use the application module's `getDBTransaction()` method to accesses the current transaction object and call its `commit()` method to commit the transaction.

Example 6–7 Updating an Existing Entity Row

```
// Custom method in SRServiceImpl.java
public void updateRequestStatus(long requestId, String newStatus) {
  // 1. Find the service request entity
  EntityImpl svcReq = retrieveServiceRequestById(requestId);
  if (svcReq != null) {
    // 2. Set its Status attribute to a new value
    svcReq.setAttribute("Status",newStatus);
    try {
      // 3. Commit the transaction
      getDBTransaction().commit();
    }
    catch (JboException ex) {
      getDBTransaction().rollback();
      throw ex;
    }
  }
}
```

The example for *removing* an entity row would be the same as this, except that after finding the existing entity, you would use the following line instead to remove the entity before committing the transaction:

```
// Remove the entity instead!
svcReq.remove();
```

6.8.4 Creating a New Entity Row

In addition to using the entity definition for finding existing entity rows, you can also use it to create new ones. Changing focus from service requests to products for a moment, you could write a `createProduct()` method like the one shown in Example 6–8 to accept the name and description of a new product, and return the new product ID assigned to it. Assume that the `ProdId` attribute of the `Product` entity object has been updated to have the `DBSequence` type discussed in Section 6.6.3.8, "Trigger-Assigned Primary Key Values from a Database Sequence", so that its value is automatically refreshed to reflect the value the `ASSIGN_PRODUCT_ID` trigger on the `PRODUCTS` table will assign to it from the `PRODUCTS_SEQ` sequence in the SRDemo application schema.

The example follows these steps:

1. Find the entity definition.

   Use `EntityDefImpl.findDefObject()` to find the entity definition for the `Product` entity.

2. Create a new instance.

   Use the `createInstance2()` method on the entity definition to create a new instance of the entity object.

Note: The method name really has a `2` at the end. The regular `createInstance()` method has `protected` access and is designed to be customized by developers as described Section D.2.5, "EntityDefImpl Class" of Appendix D, "Most Commonly Used ADF Business Components Methods". The second argument of type `AttributeList` is used to supply attribute values that *must* be supplied at create time; it is *not* used to initialize the values of all attributes found in the list. For example, when creating a new instance of a composed child entity row using this API, you must supply the value of a composing parent entity's foreign key attribute in the `AttributeList` object passed as the second argument. Failure to do so results in an `InvalidOwnerException`.

---

3. Set attribute values.

   Use the `setAttribute()` method on the entity object to assign values for the `Name` and `Description` attributes in the new entity row.

4. Commit the transaction

   Call `commit()` on the current transaction object to commit the transaction.

5. Return the trigger-assigned product ID to the caller

   Use `getAttribute()` to retrieve the `ProdId` attribute as a `DBSequence`, then call `getSequenceNumber().longValue()` to return the sequence number as a `long` value to the caller.

Example 6–8 Creating a New Entity Row

```
// Custom method in SRServiceImpl.java
public long createProduct(String name, String description) {
  String entityName = "devguide.model.entities.Product";
  // 1. Find the entity definition for the Product entity
  EntityDefImpl productDef = EntityDefImpl.findDefObject(entityName);
  // 2. Create a new instance of a Product entity
  EntityImpl newProduct = productDef.createInstance2(getDBTransaction(),null);
  // 3. Set attribute values
  newProduct.setAttribute("Name",name);
  newProduct.setAttribute("Description",description);
  try {
      // 4. Commit the transaction
      getDBTransaction().commit();
  }
  catch (JboException ex) {
    getDBTransaction().rollback();
    throw ex;
  }
  // 5. Access the database trigger assigned ProdId value and return it
  DBSequence newIdAssigned = (DBSequence)newProduct.getAttribute("ProdId");
  return newIdAssigned.getSequenceNumber().longValue();
}
```

6.8.5 Testing Using a Static Main Method

At this point, you are ready to test your custom application module methods. One common technique to build testing code into an object is to include that code in the `static main()` method. Example 6–9 shows a sample `main()` method you could add to your `SRServiceImpl.java` custom application module class to test the sample methods you wrote above. You'll make use of the same `Configuration` object you used in Section 5.7, "How to Create a Command-Line Java Test Client", to instantiate and work with the application module for testing.

Note: The fact that this `Configuration` object resides in the `oracle.jbo.client` package suggests its use for accessing an application module as an application *client*, and a `main()` method is a kind of programmatic, command-line client, so this OK. Furthermore, even though it is not best practice to cast the return value of `createRootApplicationModule()` directly to an application module's implementation class, it's legal to do in this one situation since despite its being a client to the application module, the `main()` method's code resides right inside the application module implementation class itself.

A quick glance through the code shows that it's exercising the four methods created above to:

1. Retrieve the status of service request 101
2. Retrieve the name of the technician assigned to service request 101
3. Set the status of service request 101 to illegal value "Reopened"
4. Create a new product supplying a null product name
5. Create a new product and display its newly assigned product ID

Example 6–9 Sample Main Method to Test SRService Application Module from the Inside

```
// Main method in SRServiceImpl.java
   public static void main(String[] args) {
     String        amDef = "devguide.model.SRService";
     String        config = "SRServiceLocal";
     ApplicationModule am =
       Configuration.createRootApplicationModule(amDef,config);
     /*
      * NOTE: This cast to use the SRServiceImpl class is OK since this
      * ----  code is inside a business tier *Impl.java file and not in a
      *       client class that is accessing the business tier from "outside".
      */
     SRServiceImpl service = (SRServiceImpl)am;
     // 1. Retrieve the status of service request 101
     String status = service.findServiceRequestStatus(101);
     System.out.println("Status of SR# 101 = " + status);
     // 2. Retrieve the name of the technician assigned to service request 101
     String techName = service.findServiceRequestTechnician(101);
     System.out.println("Technician for SR# 101 = " + techName);
     try {
       // 3. Set the status of service request 101 to illegal value "Reopened"
       service.updateRequestStatus(101,"Reopened");
     }
     catch (JboException ex) {
       System.out.println("ERROR: "+ex.getMessage());
     }
     long id = 0;
     try {
       // 4. Create a new product supplying a null product name
       id = service.createProduct(null,"Makes Blended Fruit Drinks");
     }
     catch (JboException ex) {
       System.out.println("ERROR: "+ex.getMessage());
     }
     // 5. Create a new product and display its newly assigned product id
     id = service.createProduct("Smoothie Maker","Makes Blended Fruit Drinks");
     System.out.println("New product created successfully with id = "+id);
     Configuration.releaseRootApplicationModule(am,true);
   }
```

Running the `SRServiceImpl.java` class calls the `main()` method in Example 6–9, and shows the following output:

```
Status of SR# 101 = Closed
Technician for SR# 101 = Bruce Ernst
ERROR: The status must be Open, Pending, or Closed
ERROR: JBO-27014: Attribute Name in Product is required
New product created successfully with id = 209
```

Notice that the attempt to set the service request status to "Reopened" failed due to the List Validator failing on the `ServiceRequest` entity object's `Status` attribute, shown in Figure 6–17. That validator was configured to allow only values from the static list `Open`, `Pending`, or `Closed`. Also notice that the first attempt to call `createProduct()` with a null for the product name raises an exception due to the built-in mandatory validation on the `Name` attribute of the `Product` entity object.

Note: You may be asking yourself, "How would a client application invoke the custom service methods I've created in my `SRService` application module, instead of being called by a `main()` method in the same class?" You'll learn the simple steps to enable this in Section 8.4, "Publishing Custom Service Methods to Clients". You'll see that it's a straightforward configuration option involving the **Client Interface** page of the Application Module Editor

6.9 Generating Custom Java Classes for an Entity Object

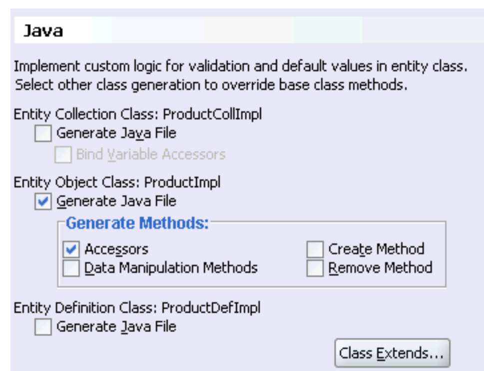
As you've seen so far in this chapter, all of the database interaction and a large amount of declarative runtime functionality of an entity object can be achieved without using custom Java code. When you need to go beyond the declarative features to implement custom business logic for your entities, you'll need to enable custom Java generation for the entities that require custom code. Appendix D, "Most Commonly Used ADF Business Components Methods", provides a quick reference to the most common code that you will typically write, use, and override in your custom entity object and entity definition classes. Later chapters discuss specific examples of how the SRDemo application uses custom code in its entity classes as well.

6.9.1 How To Generate Custom Classes

To enable the generation of custom Java classes for an entity object, use the **Java** page of the Entity Object Editor. As shown in Figure 6–20, there are three optional Java classes that can be related to an entity object. While the **Entity Collection Class** is rarely customized in practice, the **Entity Object Class** is the most frequently customized, with the **Entity Definition Class** getting customized less frequently:

- Entity collection class — rarely customized.
- Entity object class — the most frequently customized, it represents each row in the underlying database table.
- Entity definition class — less frequently customized, it represents the related class that manages entity rows and defines their structure.

Figure 6–20 Entity Object Custom Java Generation Options



6.9.1.1 Choosing to Generate Entity Attribute Accessors

When you enable the generation of a custom entity object class, if you also select the **Accessors** checkbox, then JDeveloper generates getter and setter methods for each attribute in the entity object. For the `ServiceRequest` entity object, the corresponding custom `ServiceRequestImpl.java` class would have methods like this generated in it:

```
public Number getSvrId() {...}
public void setSvrId(Number value) {...}
public String getStatus() {...}
public void setStatus(String value) {...}
public Date getRequestDate() {...}
public void setRequestDate(Date value) {...}
public String getProblemDescription() {...}
public void setProblemDescription(String value) {...}
public Number getProdId() {...}
public void setProdId(Number value) {...}
public Number getCreatedBy() {...}
public void setCreatedBy(Number value) {...}
public Number getAssignedTo() {...}
public void setAssignedTo(Number value) {...}
public Date   getAssignedDate() {...}
public void setAssignedDate(Date value) {...}
public ProductImpl getProduct() {...}
public void setProduct(ProductImpl value) {...}
public RowIterator getServiceHistories() {...}
public UserImpl getTechnicianAssigned() {...}
public void setTechnicianAssigned(UserImpl value) {...}
public UserImpl getCreatedByUser() {...}
public void setCreatedByUser(UserImpl value) {...}
```

These methods allow you to work with the row data with compile-time checking of the correct datatype usage. That is, instead of writing a line like this to get the value of the `ProdId` attribute:

```
Number prodId = (Number)svcReq.getAttribute("ProdId");
```

you can write the code like:

```
Number prodId = svcReq.getProdId();
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `ProductCode` instead of `ProdId`:

```
// spelling name wrong gives compile error
Number prodId = svcReq.getProductCode();
```

Without the generated entity object accessor methods, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both attribute name and type cast are wrong, but compiler cannot catch it
String prodId = (String)svcReq.getAttribute("ProductCode");
```

It contains both an incorrectly spelled attribute name, as well as an incorrectly typed cast of the `getAttribute()` return value. When you use the generic APIs on the `Row` interface, which the base `EntityImpl` class implements, errors of this kind will raise exceptions at runtime instead of being caught at compile time.

6.9.2 What Happens When You Generate Custom Classes

When you select one or more custom Java classes to generate, JDeveloper creates the Java file(s) you've indicated. For an entity object named `devguide.model.entities.ServiceRequest`, the default names for its custom Java files will be `ServiceRequestImpl.java` for the entity object class and `ServiceRequestDefImpl.java` for the entity definition class. Both files get created in the same `./devguide/model/entities` directory as the component's XML component definition file.

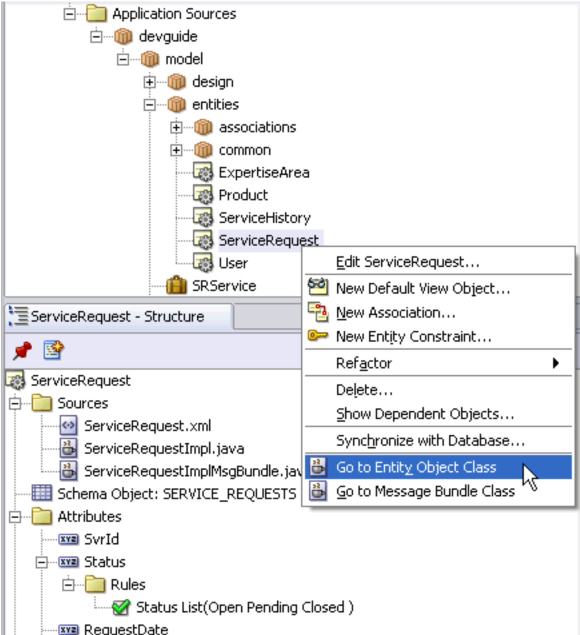
The Java generation options for the entity object continue to be reflected on the **Java** page on subsequent visits to the View Object Editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you didn't require a custom Java file for any reason, unchecking the relevant options in the **Java** page will cause the custom Java files to be removed.

6.9.3 Seeing and Navigating to Custom Java Files

As with all ADF components, when you select an entity object in the Application Navigator, the Structure window displays all of its related implementation files. The only *required* file is the XML component definition file. You saw above that when translatable UI control hints are defined for a component, it will have a component message bundle file as well. As shown in Figure 6–21, when you've enabled generation of custom Java classes, they also appear under the **Sources** folder for the entity object. When you need to see or work with the source code for a custom Java file, there are two ways to open the file in the source code editor:

- You can choose the relevant **Go to** option in the context menu, as shown in Figure 6–21
- You can double-click on a file in the **Sources** folder in the Structure window

Figure 6–21 Seeing and Navigating to Custom Java Classes for an Entity Object



6.9.4 What You May Need to Know About Custom Java Classes

See the following sections for additional information about custom Java classes.

6.9.4.1 About the Framework Base Classes for an Entity Object

When you use an "XML-only" entity object, at runtime its functionality is provided by the default ADF Business Components implementation classes. Each custom Java class that gets generated will automatically extend the appropriate ADF Business Components base class so your code inherits the default behavior and can easily add or customize it. An entity object class will extend `EntityImpl`, while the entity definition class will extend `EntityDefImpl` (both in the `oracle.jbo.server` package).

6.9.4.2 You Can Safely Add Code to the Custom Component File

Based perhaps on previous negative experiences, some developers are hesitant to add their own code to generated Java source files. Each custom Java source code file that JDeveloper creates and maintains for you includes the following comment at the top of the file to clarify that it is safe to add your own custom code to this file.

```
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
```

JDeveloper does not blindly regenerate the file when you click the **OK** or **Apply** button in the component editor. Instead, it performs a smart update to the methods that it needs to maintain, leaving your own custom code intact.

6.9.4.3 Configuring Default Java Generation Preferences

You've seen how to generate custom Java classes for your view objects when you need to customize their runtime behavior or simply prefer to have strongly typed access to bind variables or view row attributes.

To configure the default settings for ADF Business Components custom Java generation, you can select the **Tools | Preferences...** menu and open the **Business Components** page to set your preferences to be used for business components created in the future. Oracle recommends that developers getting started with ADF Business Components set their preference to generate no custom Java classes by default. As you run into a specific need for custom Java code, as you've learned in this section, you can enable just the bit of custom Java you need for that one component. Over time, you'll discover which set of defaults works best for you.

6.9.4.4 Attribute Indexes and InvokeAccessor Generated Code

As you've seen, the entity object is designed to function either in an XML-only mode or using a combination of an XML component definition and a custom Java class. Due to this feature, attribute values are not stored in private member fields of an entity's class since such a class is not present in the XML-only situation. Instead, in addition to a name, attributes are also assigned a numerical index in the entity's XML component definition based on the zero-based, sequential order of the <Attribute> and association-related <AccessorAttribute> tags in that file. At runtime attribute values in an entity row are stored in a sparse array structure managed by the base `EntityImpl` class, indexed by the attribute's numerical position in the entity's attribute list.

For the most part this private implementation detail is unimportant, since as a developer using entity objects you are shielded from having to understand this. However, when you enable a custom Java class for your entity object, this implementation detail is related to some of the generated code that JDeveloper automatically maintains in your entity object class. It is sensible to understand what that code is used for. For example, in the custom Java class for the `ServiceRequest` entity object, Example 6–10 shows that each attribute or accessor attribute has a corresponding generated integer constant. JDeveloper ensures that the values of these constants correctly reflect the ordering of the attributes in the XML component definition.

Example 6–10 Attribute Constants are Automatically Maintained in the Custom Entity Java Class

```
public class ServiceRequestImpl extends EntityImpl {
  public static final int SVRID = 0;
  public static final int STATUS = 1;
  public static final int REQUESTDATE = 2;
  public static final int PROBLEMDESCRIPTION = 3;
  public static final int PRODID = 4;
  public static final int CREATEDBY = 5;
  public static final int ASSIGNEDTO = 6;
  public static final int ASSIGNEDDATE = 7;
  public static final int TECHNICIANASSIGNED = 8;
  public static final int CREATEDBYUSER = 9;
  public static final int PRODUCT = 10;
  public static final int SERVICEHISTORIES = 11;
  // etc.
```

You'll also notice that the automatically maintained, strongly typed getter and setter methods in the entity object class use these attribute constants like this:

```
// In devguide.model.entities.ServiceRequestImpl class
public Number getAssignedTo() {
  return (Number)getAttributeInternal(ASSIGNEDTO); // <-- Attribute constant
}
public void setAssignedTo(Number value) {
  setAttributeInternal(ASSIGNEDTO, value); // <-- Attribute constant
}
```

That last aspect of the automatically maintained code related to entity attribute constants are the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods. These methods optimize the performance of attribute access by numerical index, which is how generic code in the `EntityImpl` base class typically accesses attribute values when performing generic processing. An example of the `getAttrInvokeAccessor()` method looks like the following from the `ServiceRequestImpl.java` class. The companion `setAttrInvokeAccessor()` method looks similar.

```
// In devguide.model.entities.ServiceRequestImpl class
/** getAttrInvokeAccessor: generated method. Do not modify. */
protected Object getAttrInvokeAccessor(int index,AttributeDefImpl attrDef)
throws Exception {
  switch (index) {
  case SVRID:              return getSvrId();
  case STATUS:             return getStatus();
  case REQUESTDATE:        return getRequestDate();
  case PROBLEMDESCRIPTION: return getProblemDescription();
  case PRODID:             return getProdId();
  case CREATEDBY:          return getCreatedBy();
  case ASSIGNEDTO:         return getAssignedTo();
  case ASSIGNEDDATE:       return getAssignedDate();
  case SERVICEHISTORIES:   return getServiceHistories();
  case TECHNICIANASSIGNED: return getTechnicianAssigned();
  case CREATEDBYUSER:      return getCreatedByUser();
  case PRODUCT:            return getProduct();
  default:
    return super.getAttrInvokeAccessor(index, attrDef);
  }
}
```

The rules of thumb to remember about this generated attribute-index related code are the following.

The Do's

- Add custom code if needed inside the strongly typed attribute getter and setter methods.
- Use the Entity Object Editor to change the order or type of entity object attributes.

  JDeveloper will change the Java signature of getter and setter methods, as well as the related XML component definition for you.

The Don'ts

- Don't modify the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods.
- Don't change the values of the attribute index numbers by hand.

> **Note:** If you need to manually edit the generated attribute constants because of source control merge conflicts or other reasons, you must ensure that the zero-based ordering reflects the sequential ordering of the `<Attribute>` and `<AccessorAttribute>` tags in the corresponding entity object XML component definition.

6.9.5 Programmatic Example for Comparison Using Custom Entity Classes

In order to better evaluate the difference of using custom generated entity classes versus working with the generic `EntityImpl` class, Example 6–11 shows a version of the `SRServiceImpl.java` methods that you implemented above in a second `SRService2Impl.java` application module class. A few of the interesting differences to notice are:

- Attribute access is performed using strongly typed attribute accessors.
- Association accessor attributes return the strongly typed entity class on the other side of the association.
- Using the `getDefinitionObject()` method in your custom entity class avoids working with fully qualified entity definition names as strings.
- The `createPrimaryKey()` method in your custom entity class simplifies creating the Key object for an entity.

Example 6–11 Programmatic Entity Examples Using Strongly Typed Custom Entity Object Classes

```
package devguide.model;
import devguide.model.entities.ProductImpl;
import devguide.model.entities.ServiceRequestImpl;
import devguide.model.entities.UserImpl;

import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.DBSequence;
import oracle.jbo.domain.Number;
import oracle.jbo.server.ApplicationModuleImpl;
import oracle.jbo.server.EntityDefImpl;
import oracle.jbo.server.EntityImpl;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
/**
 * This custom application module class illustrates the same
 * example methods as SRServiceImpl.java, except that here
 * we're using the strongly typed custom Entity Java classes
 * ServiceRequestImpl, UserImpl, and ProductImpl instead of working
 * with all the entity objects using the base EntityImpl class.
 */
public class SRService2Impl extends ApplicationModuleImpl {
  /**This is the default constructor (do not remove)
   */
  public SRService2Impl() {
  }
  /*
   * Helper method to return a ServiceRequest by Id
   */
  private ServiceRequestImpl retrieveServiceRequestById(long requestId) {
    EntityDefImpl svcReqDef = ServiceRequestImpl.getDefinitionObject();
    Key svcReqKey =
      ServiceRequestImpl.createPrimaryKey(new DBSequence(requestId));
    return (ServiceRequestImpl)svcReqDef.findByPrimaryKey(getDBTransaction(),
```

```
                                                svcReqKey);
}

/*
 * Find a ServiceRequest by Id
 */
public String findServiceRequestStatus(long requestId) {
  ServiceRequestImpl svcReq = retrieveServiceRequestById(requestId);
  if (svcReq != null) {
    return svcReq.getStatus();
  }
  return null;
}

/*
 * Create a new Product and Return its new id
 */
public long createProduct(String name, String description) {
  EntityDefImpl productDef = ProductImpl.getDefinitionObject();
  ProductImpl newProduct = (ProductImpl)productDef.createInstance2(
                                         getDBTransaction(),null);
  newProduct.setName(name);
  newProduct.setDescription(description);
  try {
    getDBTransaction().commit();
  }
  catch (JboException ex) {
    getDBTransaction().rollback();
    throw ex;
  }
  DBSequence newIdAssigned = newProduct.getProdId();
  return newIdAssigned.getSequenceNumber().longValue();
}
/*
 * Update the status of an existing service request
 */
public void updateRequestStatus(long requestId, String newStatus) {
  ServiceRequestImpl svcReq = retrieveServiceRequestById(requestId);
  if (svcReq != null) {
    svcReq.setStatus(newStatus);
    try {
      getDBTransaction().commit();
    }
    catch (JboException ex) {
      getDBTransaction().rollback();
      throw ex;
    }
  }
}

/*
 * Access an associated Used entity from the ServiceRequest entity
 */
public String findServiceRequestTechnician(long requestId) {
  ServiceRequestImpl svcReq = retrieveServiceRequestById(requestId);
  if (svcReq != null) {
    UserImpl tech  = (UserImpl)svcReq.getTechnicianAssigned();
    if (tech != null) {
      return tech.getFirstName()+" "+tech.getLastName();
    }
```

```
      else {
        return "Unassigned";
      }
    }
    else {
      return null;
    }
  }
  // Original main() method generated by the application module editor
  //
  //  /**Sample main for debugging Business Components code using the tester.
  //   */
  //  public static void main(String[] args) {
  //    launchTester("devguide.model", /* package name */
  //      "SRServiceLocal" /* Configuration Name */);
  //  }
   /*
    * Testing method
    */
    public static void main(String[] args) {
      String        amDef = "devguide.model.SRService";
      String        config = "SRServiceLocal";
      ApplicationModule am =
        Configuration.createRootApplicationModule(amDef,config);
      /*
       * NOTE: This cast to use the SRServiceImpl class is OK since this
       * ----  code is inside a business tier *Impl.java file and not in a
       *       client class that is accessing the business tier from "outside".
       */
      SRServiceImpl service = (SRServiceImpl)am;
      String status = service.findServiceRequestStatus(101);
      System.out.println("Status of SR# 101 = " + status);
      String techName = service.findServiceRequestTechnician(101);
      System.out.println("Technician for SR# 101 = " + techName);
      try {
        service.updateRequestStatus(101,"Reopened");
      }
      catch (JboException ex) {
        System.out.println("ERROR: "+ex.getMessage());
      }
      long id = 0;
      try {
        id = service.createProduct(null,"Makes Blended Fruit Drinks");
      }
      catch (JboException ex) {
        System.out.println("ERROR: "+ex.getMessage());
      }
      id = service.createProduct("Smoothie Maker","Makes Blended Fruit Drinks");
      System.out.println("New product created successfully with id = "+id);
      Configuration.releaseRootApplicationModule(am,true);
    }
}
```

6.10 Adding Transient and Calculated Attributes to an Entity Object

In addition to having attributes that map to columns in an underlying table, your entity objects can include transient attributes that are value holders or that display values calculated in Java. This section explores a simple example of adding a `FullName` transient attribute to the `Users` entity object that calculates its value by concatenating the values of the `FirstName` and `LastName` attributes.

6.10.1 How to Add a Transient Attribute

To add a transient attribute to an entity object:

1. Open the **Attributes** page in the Entity Object Editor and click the **New...** button. As shown in Figure 6–22:
2. Enter a name for the attribute like `FullName`,
3. Set the Java **Attribute Type** like `String`, and
4. Deselect the **Persistent** checkbox
5. If the value will be calculated, set **Updateable** to **Never**

Then click **OK** to create the attribute.

Figure 6–22 Adding a New Transient Attribute

6.10.2 What Happens When You Add Transient Attribute

When you add a transient attribute and finish the Entity Object Editor, JDeveloper updates the XML component definition for the entity object to reflect the new attribute. Whereas a persistent entity association looks like this in the XML:

```
<Attribute
   Name="FirstName"
   IsNotNull="true"
   Precision="30"
   ColumnName="FIRST_NAME"
   Type="java.lang.String"
   ColumnType="VARCHAR2"
   SQLType="VARCHAR"
   TableName="USERS" >
</Attribute>
```

a transient attribute's <Attribute> tag looks like this, with no `TableName` and a `ColumnName` of `$none$`:

```
<Attribute
   Name="FullName"
   IsUpdateable="false"
   IsQueriable="false"
   IsPersistent="false"
   ColumnName="$none$"
   Type="java.lang.String"
   ColumnType="$none$"
   SQLType="VARCHAR" >
</Attribute>
```

6.10.3 Adding Java Code in the Entity Class to Perform Calculation

A transient attribute is a placeholder for a data value. If you change the **Updatable** property of the transient attribute to **While New** or **Always**, then the end user can enter a value for the attribute. If you want the transient attribute to display a calculated value, then you'll typically leave the **Updatable** property set to **Never** and write custom Java code that calculates the value.

After adding a transient attribute to the entity object, to make it a *calculated* attribute you need to:

- Enable a custom entity object class on the **Java** page of the Entity Object Editor, choosing to generate accessor methods
- Write Java code inside the accessor method for the transient attribute to return the calculated value

For example, after generating the `UserImpl.java` view row class, the Java code to return its calculated value would reside in the `getFullName()` method like this:

```
// Getter method for FullName calculated attribute in UserImpl.java
public String getFullName() {
  // Commented out original line since we'll always calculate the value
  // return (String)getAttributeInternal(FULLNAME);
      return getFirstName()+" "+getLastName();
}
```

To ensure that the `FullName` calculated attribute is reevaluated whenever the `LastName` or `FirstName` attributes might be changed by the end user, you can add one line to their respective setter methods to mark `FullName` as "dirty" whenever either's value is set.

```
// Setting method for FirstName attribute in UserImpl.java
public void setFirstName(String value) {
  setAttributeInternal(FIRSTNAME, value);
    // Notify any clients that the FullName attribute has changed
    populateAttribute(FULLNAME,null,true, /* send notification */
                                false, /* markAsChanged     */
                                false);/* saveCopy          */
}
```

and

```
  public void setLastName(String value) {
    setAttributeInternal(LASTNAME, value);
    // Notify any clients that the FullName attribute has changed
    populateAttribute(FULLNAME,null,true, /* send notification */
                                false, /* markAsChanged     */
                                false);/* saveCopy          */
  }
```

7 Building an Updatable Data Model With Entity-Based View Objects

This chapter describes how to build updatable view objects that cooperate automatically with entity objects to enable a fully updatable data model.

This chapter includes the following sections:

- Section 7.1, "Introduction to Entity-Based View Objects"
- Section 7.2, "Creating an Entity-Based View Object"
- Section 7.3, "Including Reference Entities in Join View Objects"
- Section 7.4, "Creating an Association-Based View Link"
- Section 7.5, "Testing Entity-Based View Objects Interactively"
- Section 7.6, "Adding Calculated and Transient Attributes to an Entity-Based View Object"
- Section 7.7, "Understanding How View Objects and Entity Objects Cooperate at Runtime"
- Section 7.8, "Working Programmatically with Entity-Based View Objects"
- Section 7.9, "Summary of Difference Between Entity-Based View Objects and Read-Only View Objects"

7.1 Introduction to Entity-Based View Objects

An entity-based view object supports updatable rows. The view object queries just the data needed for the client-facing task at hand, then cooperates with one or more entity objects in your business domain layer to automatically validate and save changes made to its view rows. Like the read-only view object, an entity-based view object encapsulates a SQL query, can be linked into master/detail hierarchies, and can be used in the data model of your application modules.

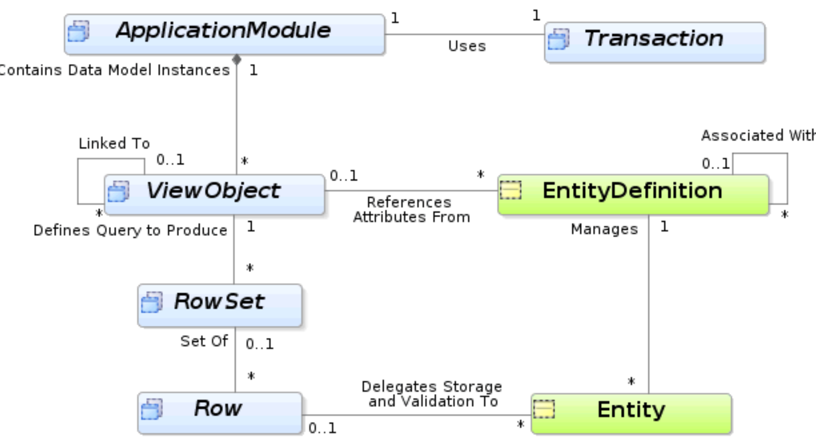
By the end of this chapter, you will understand the concepts shown in Figure 7–1:

- You define an updatable view object by referencing attributes from one or more entity objects.
- You can use multiple, associated entity objects to simplify working with reference information.
- You can define view links based on underlying entity associations.

- You use your entity-based view objects in the context of an application module that provides the transaction.
- At runtime, the view row delegates the storage and validation of its attributes to underlying entity objects.

This chapter explains how instances of entity-based view objects in an application module's data model enable clients to search for, update, insert, and delete business domain layer information in a way that combines the full data shaping power of SQL with the clean, object-oriented encapsulation of reusable, domain business objects. And all without requiring a line of code.

Figure 7–1 View Objects and Entity Objects Collaborate to Enable an Updatable Data Model



Note: The examples in this chapter use the same basic SRDemo application business domain layer of `ServiceRequest`, `ServiceHistory` `Product`, `User`, and `ExpertiseArea` entity objects from Chapter 6, "Creating a Business Domain Layer Using Entity Objects". To experiment with a working version of the examples, download the `DevGuideExamples` workspace from the *Example Downloads* page at `http://otn.oracle.com/documentation/jdev/b25947_01` and see the `EntityBasedViewObjects` project.

7.2 Creating an Entity-Based View Object

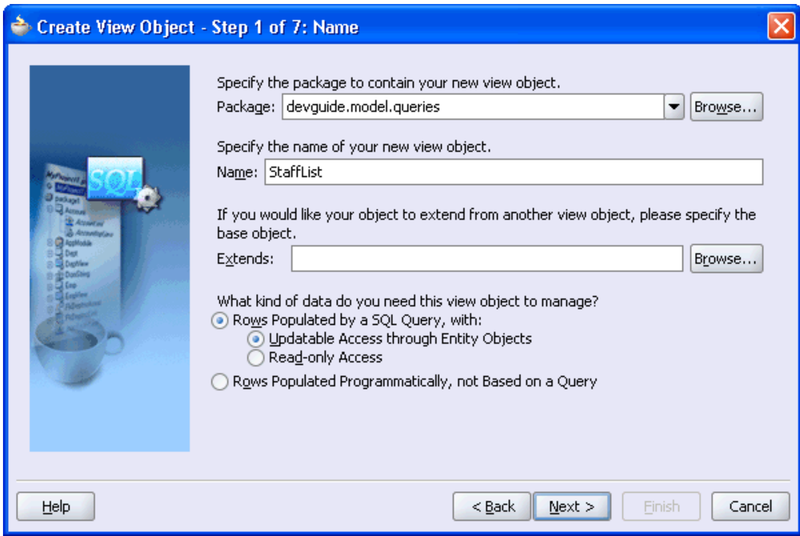
Creating an entity-based view object is even easier than creating a read-only view object, since you don't have to type in the SQL statement yourself. An entity-based view object also offers significantly more runtime functionality than its read-only counterpart.

7.2.1 How to Create an Entity-Based View Object

To create an entity-based view object, use the Create View Object wizard. The wizard is available from the **New Gallery** in the **Business Tier > ADF Business Components** category. Assume that you want to create a `StaffList` view object in the `devguide.model.queries` package to retrieve an updatable list of staff members.

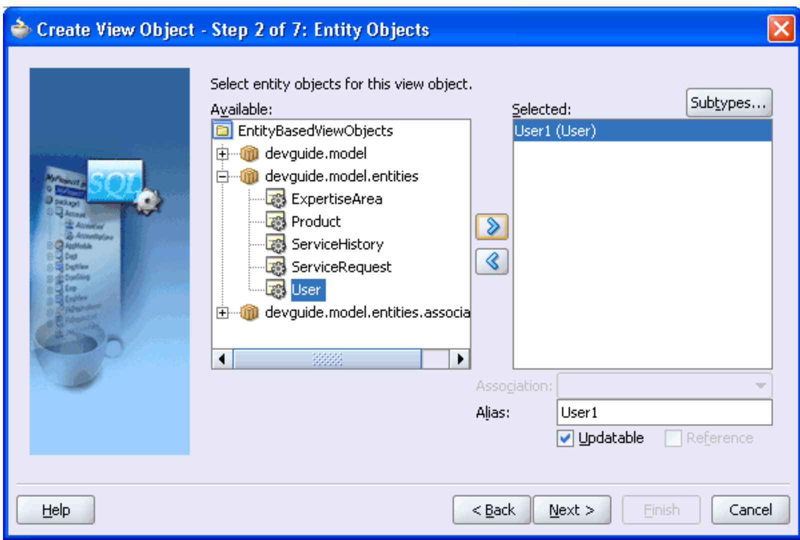
As shown in Figure 7–2, in step 1 on the **Name** page, provide the view object's name and package. Keep the default setting to manage data with **Updatable Access through Entity Objects**.

Figure 7–2 Providing a Name and Package for an Updatable View Object



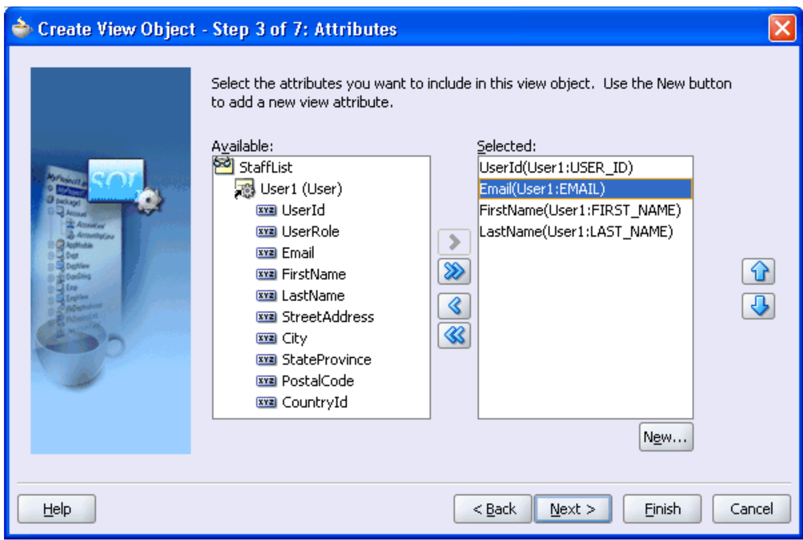
In step 2 on the **Entity Objects** page, select the entity object whose data you want to use in the view object. Figure 7–3 shows the result after selecting the `User` entity object and shuttling it into the **Selected** list. An entry in this list is known as an *entity usage* — since it records the entity objects that the view object will be using. It could also be thought of as an *entity reference*, since the view object references attributes from that entity.

Figure 7–3 Selecting the Entity Objects Whose Data You Want to Include in the View Object



In step 3 on the **Attributes** page, select the attributes you want to include from the entity usage in the **Available** list and shuttle them to the **Selected** list. In Figure 7–4, the `UserId`, `Email`, `FirstName`, and `LastName` attributes have been selected.

Figure 7–4 Selecting the Entity Attributes Whose Data You Want to Include in the View Object



In step 4 on the **Attribute Settings** page, you can use the **Select Attribute** dropdown to switch between the view object attributes in order to change their names or any of their initial settings. For this example, you can accept the defaults.

In step 5 on the **SQL Statement** page, as shown in Figure 7–5, JDeveloper automatically generates the SELECT statement based on the entity attributes you've selected. You can add a WHERE and ORDER BY clause to the query to filter and order the data as required. Since this `StaffList` view object should display only the rows in the USERS table that have a value of technician or manager for the USER_ROLE column, you can include an appropriate WHERE clause predicate in the **Where** field. To order the data by last name and first name, included an appropriate ORDER BY clause in the **Order By** field. Notice that the **Where** and **Order By** field values appear without the WHERE or ORDER BY keyword. The view object adds those keywords at runtime when it executes the query.

Figure 7–5 Adding Custom Where and Order By Clauses to the Generated SQL Statement

Click **Finish** at this point to create the view object.

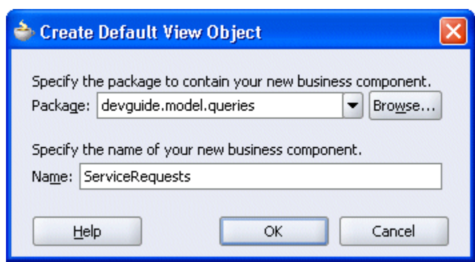
7.2.1.1 Creating a View Object Having All Attributes of an Entity Object

When you want to allow the client to work with *all* of the attributes of an underlying entity object, you can use the Create View Object wizard as described in Section 7.2.1, "How to Create an Entity-Based View Object". After selecting the entity object, simply select *all* of its attributes on the **Attributes** page. However, for this frequent operation, there is an even quicker way to perform the same task in the Application Navigator.

To create a new entity-based view object:

1. Select the desired entity object in the Application Navigator.
2. Choose **New Default View Object...** from the context menu.
3. Provide a package and component name for the new view object in the Create Default View Object dialog, as shown in Figure 7–6.

Figure 7–6 Shortcut to Creating a Default View Object for an Entity Object



The new entity-based view object created will be identical to one you could have created with the Create View Object wizard. By default, it will have a single entity usage referencing the entity object you selected in the Application Navigator, and will include all of its attributes. It will initially have neither a WHERE nor ORDER BY clause, and you may want to use the View Object Editor to:

- Remove unneeded attributes
- Refine its selection with a WHERE clause
- Order its results with an ORDER BY clause
- Customize any of the view object properties

7.2.2 What Happens When You Create an Entity-Based View Object

When you create an entity-based view object, JDeveloper creates the XML component definition file that represents the view object's declarative settings and saves it in the directory that corresponds to the name of its package. In Figure 7–2, the view object was named `StaffList` in the `devguide.model.queries` package, so the XML file created will be `./devguide/model/queries/StaffList.xml` under the project's source path. This XML file contains the information about the SQL query, the name of the entity usage, and the properties of each attribute. If you're curious to see its contents, you can see the XML file for the view object by selecting the view object in the Application Navigator and looking in the corresponding **Sources** folder in the Structure Window. Double-clicking on the `StaffList.xml` node will open the XML in an editor so you can inspect it.

Note: If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom view object class `StaffListImpl.java` and/or a custom view row class `StaffListRowImpl.java` class.

7.2.3 Editing an Existing Entity-Based View Object Definition

After you've created an entity-based view object, you can edit any of its settings by using the View Object Editor. Select the **Edit** menu option on the context menu in the Application Navigator, or double-click on the view object, to launch the view object editor. By opening the different panels of the editor, you can adjust the WHERE and ORDER BY clause of the query, change the attribute names, add named bind variables, add UI controls hints, control Java generation options, and configure other settings.

7.2.4 What You May Need to Know About View Objects

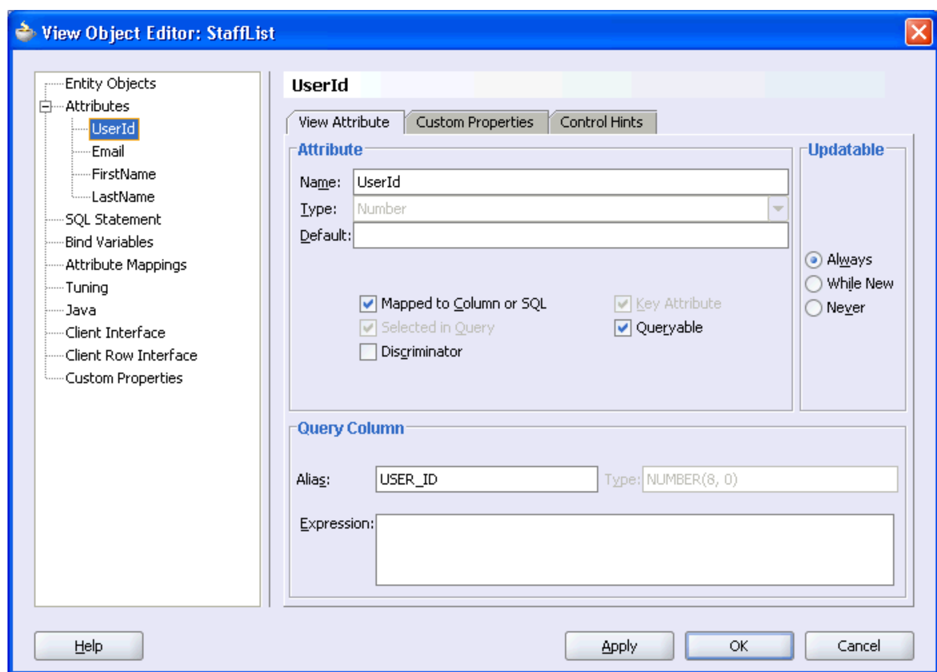
Each view object attribute inherits the properties of the corresponding entity object attribute.

7.2.4.1 View Object Attributes Inherit Properties from Underlying Entity Object Attributes

One interesting aspect of entity-based view objects is that each attribute that relates to an underlying entity object attribute inherits that attributes properties. Figure 7–7 shows the View Object Editor with the `UserId` attribute selected. You can see that properties like the Java **Attribute Type** and the **Query Column Type** are disabled and their values are inherited from the related UserId attribute of the `User` entity object to which this view object is related. Some properties like the attribute's datatype are inherited and cannot be changed at the view object level.

Other properties like Queryable and Updatable are inherited but can be overridden as long as their overridden settings are more restrictive than the inherited settings. For example, the UserId attribute in the User entity object has an **Updatable** setting of **Always**. As shown Figure 7–7, the View Object Editor would allow you to set the corresponding view object attribute to a more restrictive setting like **While New** or **Never**. However, if the UserId attribute in the User entity object had instead an **Updatable** setting of **Never**, then the editor would not allow the StaffList's related view object attribute to have a less restrictive setting like **Always**.

Figure 7–7 View Object Attributes Inherit Properties from Underlying Entity Object Attributes



7.3 Including Reference Entities in Join View Objects

It is extremely common in business applications to supplement information from a *primary* business domain object with secondary reference information to help the end user understand what foreign key attributes represent. Take the example of the ServiceRequest entity object. It contains foreign key attributes of type Number like:

- CreatedBy, representing the user who created the request
- AssignedTo, representing the user to whom the request is assigned
- ProdId, representing the product to which the request pertains

From experience, you know that showing an end user exclusively these "raw" numerical values won't be very helpful. Ideally, reference information from the related User and Product entity objects should be displayed to improve the application's usability. One typical solution involves performing a join query that retrieves the combination of the primary and reference information. Alternatively, developers populate "dummy" fields in each queried row with reference information based on extra queries against the lookup tables. When the end user can *change* the foreign key values as she edits the data, this presents an additional challenge.

For example, when reassigning a service request from one technician to another, the end user expects the reference information to stay in sync. Luckily, entity-based view objects support easily including reference information that's always up to date. The key requirement to leverage this feature is the presence of associations between the entity object that act as the view object's primary entity usage and the entity objects that contribute reference information.

This section describes how to modify the default `ServiceRequests` view object created above to include reference information from the `User` and `Product` entity objects.

7.3.1 How to Include Reference Entities in a View Object

To include reference entities in a view object, open the View Object Editor on an entity-based view object that already has a single entity usage, and open the **Entity Objects** page. The first entity usage in the **Selected** list on this page is known as the *primary* entity usage for the view object. The list is not limited to a single entity usage, however. To use additional entity objects in the view object, select them in the **Available** list and shuttle them to the **Selected** list.

7.3.1.1 Adding Additional Reference Entity Usages to the View Object

Figure 7–8 shows the result of adding three additional reference entity usages to the existing `ServiceRequests` view object: one for the `Product` and two separate usages of the `User` entity. When you click on an entity usage in the **Selected** list, the state of the **Reference** checkbox indicates that the second and subsequent entity usages added to a view object are marked as reference information by default. Similarly, the secondary entity usages default to being not updatable, as the unchecked state of their **Updatable** checkbox confirms.

Note: When adding secondary entity usages, they default to having their **Updatable** flag false and their **Reference** flag true. This is by far the most common usage pattern. In Section 27.9, "Creating a View Object with Multiple Updatable Entities", you'll explore the less common, yet still useful, situation of having a join view object with *multiple*, updatable entity usages.

The **Association** dropdown list shows you the name of the association that relates the selected entity usage to the primary one. The **Alias** field allows you to give a more meaningful name to the entity usage when the default name is not clear. For example, after shuttling two entity usages for the `User` entity object into the **Selected** list, initially the alias for the usages was `User1` and `User2`. You can see in the figure that renaming these to be `Technician` and `Customer` instead greatly clarifies what reference information they are contributing to the view object. Importantly, the figure also illustrates that when you add multiple entity usages for the same entity, you need to use the **Association** dropdown list to select which association represents that usage's relationship to the primary entity usage. For the `Technician` entity usage select the `ServiceRequestsAssignedToUser` association, and for the `Customer` entity usage select the `ServiceRequestsCreatedByUser` association.

Figure 7–8 Indicating Correct Associations for Multiple Reference Entity Usages

7.3.1.2 Selecting Additional Attributes from Reference Entity Usages

After adding these secondary entity usages, switch to the **Attributes** page of the View Object Editor and select the specific, additional *attributes* from these new usages that you want to include in the view object. Figure 7–9 illustrates the result of shuttling the following extra attributes into the **Selected** list:

- The `Name` attribute from the `Product` entity usage
- The `Email` attribute from the `Technician` entity usage
- The `Email` attribute from the `Customer` entity usage

Notice that even if you didn't intend to include them, JDeveloper automatically verifies that the primary key attribute from each entity usage is part of the **Selected** list. If it's not already present in the list, JDeveloper adds it for you.

Figure 7–9 Selecting Additional Reference Entity Attributes to Include in the View Object

Selecting the **SQL Statement** page, you can see that JDeveloper has included the new columns in the SELECT statement and has updated the **Where** field to include the appropriate join clauses of:

```
((ServiceRequest.PROD_ID = Product.PROD_ID) AND
 (ServiceRequest.ASSIGNED_TO = Technician.USER_ID)) AND
 (ServiceRequest.CREATED_BY = Customer.USER_ID)
```

7.3.1.3 Renaming Attributes from Reference Entity Usages

Expanding the **Attributes** node in the tree at the left of the View Object Editor, you can see the additional attributes are added at the end of the list. Since the default attribute names are not as clear as they could be, by selecting each one in turn, you can rename them as follows:

- `Name` -> `ProductName`
- `ProdId1` -> `PKProdId` (Primary Key from `Product` entity usage)
- `Email` -> `TechnicianEmail`
- `UserId` -> `PKTechnicianUserId` (Primary Key from `Technician` entity usage)
- `Email1` -> `CustomerEmail`
- `UserId1` -> `PKCustomerUserId` (Primary Key from `Customer` entity usage)

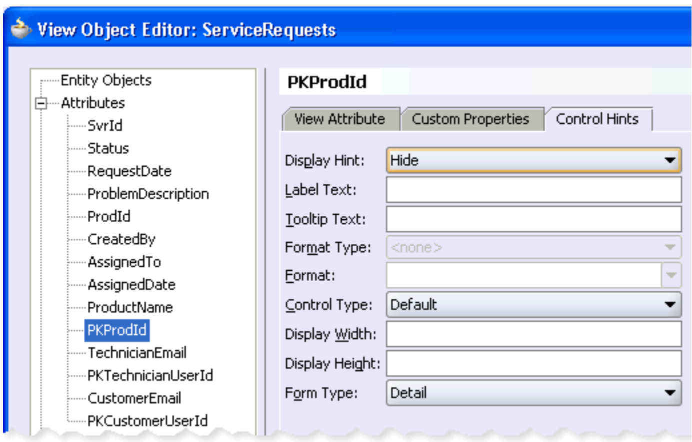
7.3.1.4 Removing Unnecessary Key Attributes from Reference Entity Usages

The view object attribute corresponding to the primary key attribute of the primary entity usage acts at the primary key for identifying the view row. When you add secondary entity usages, JDeveloper also marks the view object attributes corresponding to their primary key attributes as part of the view row key as well. When your view object consists of a single updatable primary entity usage and a number of reference entity usages, the primary key attribute from the primary entity usage already is enough to uniquely identify the view row. These additional key attributes are unneeded and you should toggle their **Key Attribute** setting to `false`. For the view object created above, toggle this setting to `false` for the following attributes so that **Key Attribute** is no longer checked: `PKProdId`, `PKTechnicianUserId`, and `PKCustomerUserId`.

7.3.1.5 Hiding the Primary Key Attributes from Reference Entity Usages

Since you generally won't want to display the primary key attributes that were automatically added to the view object, you can set their **Display Hint** property on the UI **Control Hints** page to **Hide** as shown in Figure 7–10.

Figure 7–10 Setting an Attribute Control Hint to Hide Primary Key Attributes from Reference Entity Usages



Click **OK** to save your changes to the view object.

7.3.2 What Happens When You Reference Entities in a View Object

When you include secondary entity usages by reference in a view object, JDeveloper updates the view object's XML component definition to include information about the additional entity usages. For example, if you look at the `ServiceRequests.xml` file for the view object that was enhanced above to include three additional reference entity usages, you will see this information recorded in the multiple `<EntityUsage>` elements in that file. For example, you'll see an entry for the primary entity usage like this:

```
<EntityUsage
   Name="ServiceRequest"
   Entity="devguide.model.entities.ServiceRequest"/>
```

The secondary reference entity usages will have a slightly different entry like this, including information about the association that relates it to the primary entity usage:

```
<EntityUsage
   Name="Product"
   Entity="devguide.model.entities.Product"
   Association=
    "devguide.model.entities.associations.ServiceRequestsForProduct"
   AssociationEnd=
    "devguide.model.entities.associations.ServiceRequestsForProduct.Product"
   SourceUsage="devguide.model.queries.ServiceRequests.ServiceRequest"
   ReadOnly="true"
   Reference="true"/>
```

Each attribute entry in the XML indicates which entity usage it references. This entry for the `ProblemDescription` attribute shows that it's related to the `ServiceRequest` entity usage:

```
<ViewAttribute
   Name="ProblemDescription"
   IsNotNull="true"
   EntityAttrName="ProblemDescription"
   EntityUsage="ServiceRequest"
   AliasName="PROBLEM_DESCRIPTION" >
</ViewAttribute>
```

While the `CustomerEmail` attribute is related to the `Customer` entity usage.

```
<ViewAttribute
   Name="CustomerEmail"
   IsUpdatable="false"
   IsNotNull="true"
   EntityAttrName="Email"
   EntityUsage="Customer"
   AliasName="EMAIL1" >
</ViewAttribute>
```

The View Object Editor uses this association information at design time to automatically build the view object's join WHERE clause. It uses the information at runtime to enable keeping the reference information up to date when the end user changes foreign key attribute values.

7.3.3 What You May Need to Know About Join View Objects

If your view objects reference multiple entity objects, these are displayed as separate entity usages on a business components diagram. Note that you can also modify the default inner join clause to be an outer join when appropriate.

7.3.3.1 Showing View Objects in a Business Components Diagram

Section 6.4, "Creating an Entity Diagram for Your Business Layer" explained how to create a **Business Components Diagram** to visualize your business domain layer. In addition to supporting entity objects, JDeveloper's UML diagramming support allows you to drop view objects onto diagram as well to visualize their structure and entity usages. If you create a new **Business Components Diagram** named `SRService Data Model` in the `devguide.model.design` package, and drag the `ServiceRequests` view object from the Application Navigator onto the diagram, you'll see what's shown in Figure 7–11. When viewed as an expanded node, the diagram shows a compartment containing the view objects entity usages.

Figure 7–11 View Object and Its Entity Usages in a Business Components Diagram

7.3.3.2 Modify Default Join Clause to Be Outer Join When Appropriate

When JDeveloper creates the WHERE clause for the join between the table for the primary entity usage and the tables for secondary entity usages that are related to it, by default it always creates inner joins. Study the WHERE clause of the `ServiceRequests` view object more closely:

```
((ServiceRequest.PROD_ID = Product.PROD_ID) AND
 (ServiceRequest.ASSIGNED_TO = Technician.USER_ID)) AND
 (ServiceRequest.CREATED_BY = Customer.USER_ID)
```

When service requests are not yet assigned to a technician, their `AssignedTo` attribute will be `null`. The default inner join condition generated above will not retrieve these unassigned service requests. Assuming that you want unassigned service requests to be viewable and updatable through the `ServiceRequests` view object, you'll need to revisit the **SQL Statement** page of the View Object Editor to change the query into an outer join to the USER table for the possibly null ASSIGNED_TO column value. The updated WHERE clause shown below includes the additional `(+)` operator on the side of the equals sign for the related table whose data is allowed to be missing in the join:

```
((ServiceRequest.PROD_ID = Product.PROD_ID) AND
 (ServiceRequest.ASSIGNED_TO = Technician.USER_ID (+) )) AND
 (ServiceRequest.CREATED_BY = Customer.USER_ID)
```

7.4 Creating an Association-Based View Link

Just as with read-only view objects, you can link entity-based view objects to other view objects to form master/detail hierarchies of any complexity. The only difference in the creation steps involves the case when both the master and detail view objects are entity-based view objects and their respective entity usages are related by an association. In this situation, since the association captures the set of source and destination attribute pairs that relate them, you can create the view link just by indicating which association it should be based on.

7.4.1 How to Create an Association-Based View Link

To create an association-based view link, you use the Create View Link wizard.

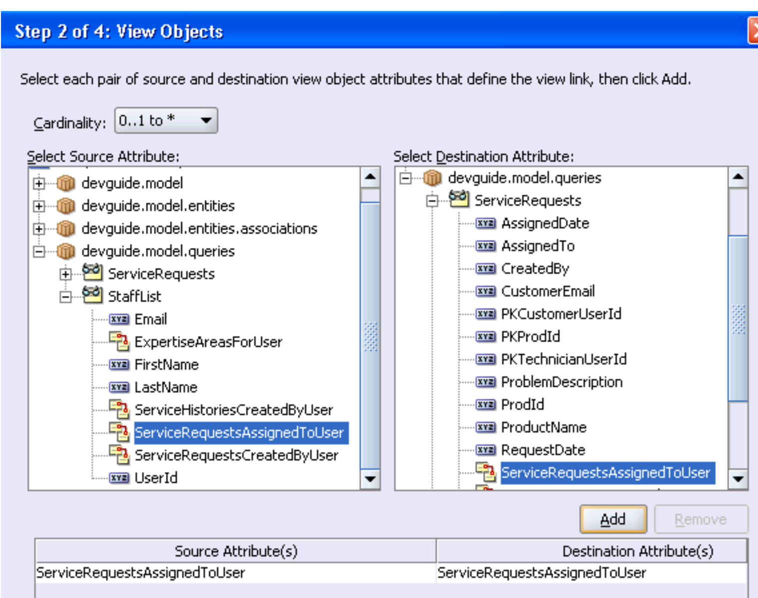
To create an association-based view link

1. From the **New Gallery** in the **Business Tier > ADF Business Components** category, select the **Create View Link wizard.**

2. In step 1 on the **Name** page, supply a package and a component name. Assume you want to create the view link in the `devguide.model.queries.viewlinks` package and call it `RequestsAssignedToTechnician`.

3. In step 2 on the **View Objects** page, in the **Select Source Attribute** tree expand the source `StaffList` view object in the `devguide.model.queries` package. In the **Select Destination Attribute** tree expand the `ServiceRequests` view object. Notice that in addition to the view object attributes, for entity-based view objects relevant associations also appear in the list. As shown in Figure 7–12, select the same `ServiceRequestsAssignedToUser` association in both **Source** and **Destination** trees, then click **Add** to add the association to the table below. Click **Next** and **Finish** to complete creating the new view link.

Figure 7–12 Selecting an Association Relating the Source View Object's Entity Usage to the Destination's



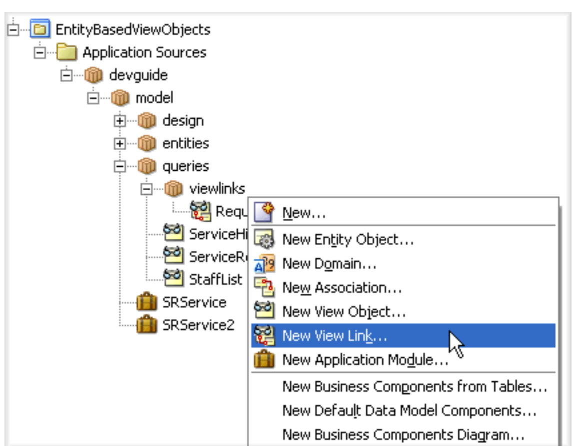
4. Next, create another association-based view link between the `ServiceRequests` view object and a view object that displays the detail information about the service request's history entries. You already have the master `ServiceRequests` view object, but you need to first create a view object for the detail before you can link them. Using the shortcut you learned above, in the Application Navigator select the `ServiceHistory` entity object in the `devguide.model.entities` package and choose **New Default View Object...** from the context menu to create a view object named `ServiceHistories` in the `devguide.model.queries` based on this entity.

Note: In the Create Default View Object you can use the combobox to select `devguide.model.queries` from the list of existing packages.

---

5. Finally, repeat the steps used above to create an association-based view link between the `ServiceRequests` view object and the new `ServiceHistories` view object based on the association that relates their respective primary entity usages. Name the view link `HistoryLinesForRequest` in the `devguide.model.queries.viewlinks` package. As an additional shortcut, to avoid having to type in the package name, as shown in Figure 7–13 you can use the **New View Link...** on the context menu of the `viewlinks` package node in the Application Navigator.

Figure 7–13 Shortcut for Creating a View Link in an Existing Package



7.4.2 What Happens When You Create an Association-Based View Link

When you create an association-based view link, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. In the example above, the view links were named `RequestsAssignedToTechnician` and `HistoryLinesForRequest` in the `devguide.model.queries.viewlinks` package, so the XML files created will be `/RequestsAssignedToTechnician.xml` and `/HistoryLinesForRequest.xml` in the `./devguide/model/queries/viewlinks` directory under the project's source path. This XML file contains the declarative information about the association that relates the source and target view objects you've specified. In addition to saving the view link component definitions themselves, JDeveloper also updates the XML definition of the *source* view objects in the view link relationships to add information about the view link accessor attribute.

7.5 Testing Entity-Based View Objects Interactively

You test entity-based view objects interactively in the same way as read-only ones. Just add instances of the desired view objects to the data model of some application module, and then test that application module using the Business Components Browser.

7.5.1 Overview of Business Component Browser Functionality for an Updatable Data Model

You'll find the Business Components Browser tool invaluable in quickly testing and debugging your application modules. Figure 7–14 gives an overview of the operations that all of the Business Components Browser toolbar buttons perform.

Figure 7–14 Business Component Browser Functionality for Updatable Data Models

Commit

Rollback

Navigate Rows

Create Row

Remove Row

Search Using Query by Example

Validate Row

Edit Bind Variable Values

Close Tab

Display Tab in Separate Window

7.5.2 Adding View Object Instances to the Data Model

Following the same steps as you learned in Section 5.10.4.3, "How to Enable Active Master/Detail Coordination in the Data Model", add the following view object instances to the data model of the `SRService` application module to end up with the hierarchy of master/detail view objects shown in Figure 7–15:

- Select existing `ServiceRequests` view object instance in the **Data Model** tree first, then add a detail instance named `ServiceHistories` of the `ServiceHistories` view object that appears as a child of `ServiceRequests` in the **Available** list.
- Select existing `StaffList` view object instance in the **Data Model** tree first, then add a detail instance named `AssignedServiceRequests` of the `ServiceRequests` view object that appears as a child of `StaffList` in the **Available** list.
- Select the new `AssignedServiceRequests` view object instance in the **Data Model** tree first, then add a detail instance named `AssignedServiceHistories` of the `ServiceHistories` view object that appears as a child of `ServiceRequests` in the **Available** list.

Figure 7–15 Business Components Browser Showing Editable Results of an Entity-Based View Object

7.5.3 How to Test Entity-Based View Objects Interactively

Assuming that you've set up the `SRService` application module's data model as shown in Figure 7–15, to test it do the following:

To test entity-based view objects:

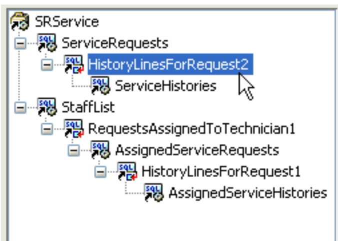
1. Select the application module in the Application Navigator and choose **Test...** from the context menu.
2. Click **Connect** on the Business Components Browser **Connect** dialog to use the default `SRServiceLocal` configuration for testing.

> **Note:** By default, an application module has only its default, local configuration, named *`AppModuleName`*`Local`. If you have created additional configurations for your application module and want to test it using one of those instead, just select the desired configuration from the **Business Components Configuration** dropdown list on the **Connect** dialog before clicking **Connect**.

7.5.4 What Happens When You Test Entity-Based View Objects Interactively

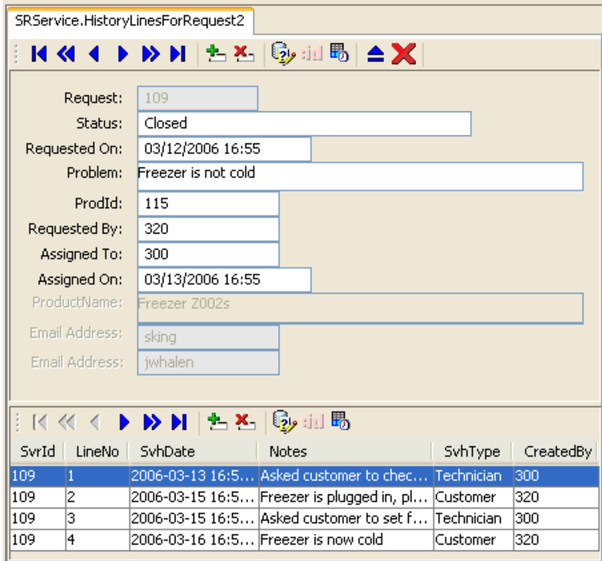
When you launch the Business Components Browser, JDeveloper starts the tester tool in a separate process and the **Business Component Browser** appears. As shown in Figure 7–16 the tree at the left of the display shows the hierarchy of the view object instances in the data model, and includes additional nodes between a master view object instance and a detail view object instance that represent the view link instance that performs the active master/detail coordination as the current row changes in the master.

Figure 7–16 SRService Data Model in the Business Components Tester



Double-clicking on the `HistoryLinesForRequest2` view link instance in the tree executes the master objects — if it has not been executed so far in the testing session — and displays the master/detail tester panel shown in Figure 7–17. Additional context menu items on the view object node allow you to reexecute the query if needed, remove the tester panel, and perform other tasks. You saw a similar master/detail panel when you used the Business Components Browser in Section 5.5, "Testing View Objects Using the Business Components Browser". You can see and scroll through the query results. One important difference is a direct result of using an entity-based view object this time. Instead of seeing *disabled* UI controls showing read-only data, you now see editable fields and are free to experiment with creating, inserting, updating, validating, committing and rolling back.

Figure 7–17 Instances of Entity-Based View Objects are Fully Editable by the Tester



Try experimenting with the multiple levels of master/detail hierarchies, opening multiple tester panels at the same time, and using the **Display Results in a Window** toolbar button to "pop" a tab out of the frame into a separate window to see multiple view object's data at the same time.

7.5.5 Simulating End-User Interaction with Your Application Module Data Model

Using the Business Components Browser, you can simulate an end user interacting with your application module data model before you have started to build any custom user interface of your own. Even *after* you have your UI pages constructed, you will come to appreciate using the Business Components Browser to assist in diagnosing problems when they arise. You can reproduce the issues in the Business Components Browser to discover if the issue lies in the view or controller layers of the application, or is instead a problem in the business service layer application module itself.

Using just the master/detail tester page shown in Figure 7–17, you can test several functional areas of your application.

7.5.5.1 Testing Master/Detail Coordination

Use the navigation buttons on the toolbar, you can see that the service history rows for the current service request are correctly co-ordinated.

7.5.5.2 Testing UI Control Hints

The prompts displayed in the testing panels help you see whether you have correctly defined a user-friendly **Label Text** control hint for each attribute. For example, the `RequestDate` attribute in the `ServiceRequests` view object instance has the prompt **Requested On**. Hovering your mouse over the edit field for the `RequestDate` field, you'll shortly see the Tooltip control hint text appear if you've defined it. The tooltip that appears says, "**The date on which the service request was created**", which is how you set up the hint on your entity object back in Section 6.5.1, "How to Add Attribute Control Hints". Since you didn't specifically define any new control hints for the `ServiceRequests` view object, this illustrates that the entity-based view object attributes inherit their control hints from those on the underlying entity object attribute.

7.5.5.3 Testing View Objects That Reference Entity Usages

By scrolling through the data — or using **Specify View Criteria** toolbar button to search — you can verify whether service requests that have not yet been assigned are correctly displaying. If you correctly altered the query's WHERE clause to use an outer join, these row will appear as expected.

By changing the `AssignedTo` attribute to a different technician's user ID — double-click on the `StaffList` view object instance in the Business Components Browser to browse for some valid staff user ID's — you can verify that the corresponding reference information is automatically updated to reflect the new technician.

By observing the values of the `ProductName` field and the two **Email Address** fields in some example rows of the `ServiceRequests`, you can see the corresponding user-friendly product name, and the email addresses of both the customer who created the request, and the technician to whom it is assigned. You can also notice a problem that both the customer's email address and the technician's email address are inheriting the **Label Text** hint from the `User` entity object of **Email Address**. It won't be clear to the end user which email address is which.

To remedy the situation where both the technician's email address and the customer's email address display with the same, inherited **Email Address** label, edit the `ServiceRequests` view object and define a **Label Text** control hint for both at the view object level. Set the **Label Text** hint to **Technician Email Address** for the `TechnicianEmail` attribute and **Customer Email Address** for the `CustomerEmail` attribute. Use the Business Components Browser to verify that these control hints defined at the view object level override the ones it would normally inherit from the underlying entity object.

7.5.5.4 Testing Business Domain Layer Validation

Try to change the `Status` attribute value of a `Closed` service request to have a value of `Reopened`. When you try to tab out of the field, you'll get an exception:

```
(oracle.jbo.AttrSetValException) The status must be Open, Pending, or Closed
```

Based on the other simple declarative validation rules defined in Section 6.7, "Using Declarative Validation Rules", you can try to update a `ProblemDescription` value to contain the word `urgent` surrounded by spaces, to receive the error:

```
(oracle.jbo.AttrSetValException)
Problem Description cannot contain the word urgent
```

Lastly, you can try to enter a `ProdId` value of `1234` to violate the range validation rule and see:

```
(oracle.jbo.AttrSetValException) Valid product codes are between 100 and 999
```

Click on the rollback button in the toolbar to revert data to the previous state.

7.5.5.5 Testing Alternate Language Message Bundles and Control Hints

By opening the **Properties** tab on the **Connect** dialog when you launch the Business Components Browser, you can override the default locale settings to change:

- `jbo.default.country` = `IT`
- `jbo.default.language` = `it`

With these properties set, you can see whether the Italian language translations of the `ServiceRequest` entity object control hints are correctly located. You'll notice **Stato**, **Aperto Il**, and **Problema** labels instead of **Status**, **Requested On**, and **Problem** (among others). You also will see that the format of `RequestDate` changes from a value like `03/12/2006 16:55` to `12/03/2006 16:55`.

7.5.5.6 Testing Row Creation and Default Value Generation

Click on the **Create Row** button in the toolbar for the `ServiceRequests` view object instance to create a new blank row. Any fields that have a declarative default value will appear with that value in the blank row. The `DBSequence`-valued `SvrId` attribute appears read-only in the new row with its temporary negative number. After entering all the required fields — try `100` for the **ProdId** and `300` for the **Requested By** field — click on the commit button to commit the transaction. The actual, trigger-assigned primary key appears in the `SvrId` field after successful commit.

7.5.5.7 Testing New Detail Rows Have Correct Foreign Keys

If you try adding a new service history row to an existing service request, you'll notice that the view link automatically ensures the foreign key attribute value for `SvrId` in the new `ServiceHistories` row is set to the value of the current master service request row.

7.6 Adding Calculated and Transient Attributes to an Entity-Based View Object

In addition to having attributes that map to underlying entity objects, your view objects can include calculated attributes that don't map to any entity object attribute value. The two kinds of calculated attributes are known as:

- *SQL-calculated attributes*, when their value is retrieved as an expression in the SQL query's SELECT list
- *Transient attributes*, when their value is not retrieved as part of the query

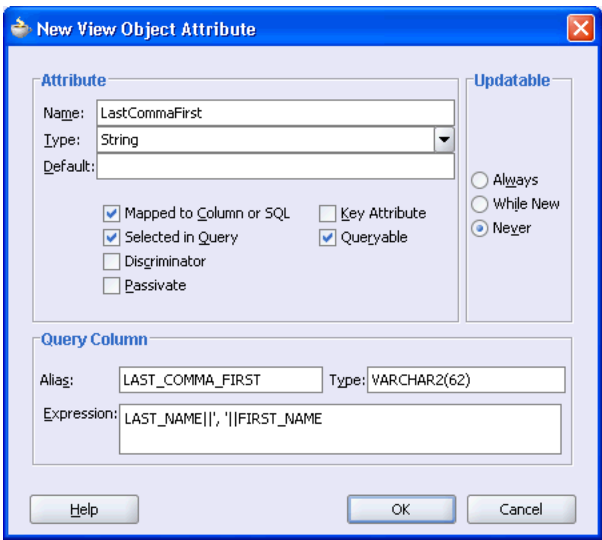
This section explains how to add both kinds, first illustrating how to add a SQL-calculated `LastCommaFirst` attribute and then a transient-calculated attribute named `FirstDotLast` to the `StaffList` view object. Finally, you'll see that a view object can include an entity-mapped attribute which itself is a transient attribute at the entity object level just to ensure that all of the supported combinations are clear.

7.6.1 How to Add a SQL-Calculated Attribute

To add a SQL-calculated attribute to an entity-based view object:

1. Open the **Attributes** page in the View Object Editor and click **New**.
2. Enter a name for the attribute, such as `LastCommaFirst`.
3. Set the Java **Attribute Type** to an appropriate value, like `String`.
4. Check the **Mapped to Column of SQL** checkbox.
5. Provide a SQL expression in the **Expression** field like `LAST_NAME||', '||FIRST_NAME`
6. Consider changing the SQL column alias to match the name of the attribute
7. Verify the database **Query Column Type** and adjust the length (or precision/scale) as appropriate.
8. Click **OK** to create the attribute.

Figure 7–18 Adding a New SQL-Calculated Attribute



7.6.2 What Happens When You Add a SQL-Calculated Attribute

When you add a SQL-calculated attribute and finish the View Object Editor, JDeveloper updates the XML component definition for the view object to reflect the new attribute. Whereas an entity-mapped attribute like `LastName` looks like this in the XML, inheriting most of it properties from the underlying entity attribute to which it is mapped:

```
<ViewAttribute
   Name="LastName"
   IsNotNull="true"
   EntityAttrName="LastName"
   EntityUsage="User1"
   AliasName="LAST_NAME" >
</ViewAttribute>
```

in contrast, a SQL-calculated attribute's <ViewAttribute> tag looks like the following. As expected, it has no `EntityUsage` or `EntityAttrName` property, and includes datatype information along with the SQL expression:

```
<ViewAttribute
   Name="LastCommaFirst"
   IsUpdatable="false"
   IsPersistent="false"
   Precision="62"
   Type="java.lang.String"
   ColumnType="VARCHAR2"
   AliasName="FULL_NAME"
   Expression="LAST_NAME||&#39;, &#39;||FIRST_NAME"
   SQLType="VARCHAR" >
</ViewAttribute>
```

Note: The `'` is the XML character reference for the apostrophe, referencing it by its numerical ASCII code of 39 (decimal). Other characters in literal text that require similar construction in XML are the less-than, greater-than, and ampersand characters.

7.6.3 How to Add a Transient Attribute

To add a transient attribute to an entity-based view object:

1. Open the **Attributes** page in the View Object Editor and click **New**.
2. Enter a name for the attribute, like `FirstDotLast`.
3. Set the Java **Attribute Type** to `String`.
4. Leave the **Mapped to Column of SQL** checkbox unchecked.
5. Click **OK** to create the attribute.

Figure 7–19 Adding a New Transient Attribute

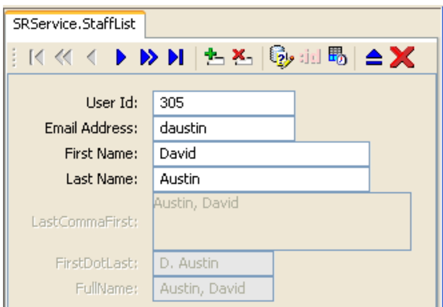
New View Object Attribute
Attribute
Name: FirstDotLast
Type: String
Default:
Mapped to Column or SQL
Key Attribute
Selected in Query
Queryable
Discriminator
Passivate
Updatable
Always
While New
Never
Query Column
Alias: VIEW_ATTR
Type: VARCHAR2
Expression:
Help
OK
Cancel

7.6.3.1 Adding an Entity-Mapped Transient Attribute to a View Object

To add a transient entity object attribute to an entity-based view object, first ensure that you have an entity usage for the entity on the **Entity Objects** page of the View Object Editor. Then go to **Attributes** page and the desired attribute from the **Available** list into the **Selected** list. Using these steps, you can add the `FullName` calculated attribute from the `User` entity object to the `StaffList` view object.

If you use the Business Components Browser to test the `SRService` data model after adding these three attributes to the `StaffList` view object lists, you can see their effect as shown in Figure 7–20:

Figure 7–20 StaffList View Object with Three Kinds of Calculated Attributes



7.6.4 What Happens When You Add a Transient Attribute

When you add a transient attribute and finish the View Object Editor, JDeveloper updates the XML component definition for the view object to reflect the new attribute. A transient attribute's <ViewAttribute> tag in the XML is similar to the SQL-calculated one, but lacks an `Expression` property.

7.6.5 Adding Java Code in the View Row Class to Perform Calculation

A transient attribute is a placeholder for a data value. If you change the **Updatable** property of the transient attribute to **While New** or **Always**, the end user can enter a value for the attribute. If you want the transient attribute to display a calculated value, then you'll typically leave the **Updatable** property set to **Never** and write custom Java code that calculates the value.

After adding a transient attribute to the view object, to make it a *calculated* transient attribute you need to:

- Enable a custom view row class on the **Java** page of the View Object Editor, choosing to generate accessor methods
- Write Java code inside the accessor method for the transient attribute to return the calculated value

For example, after enabling the generation of the `StaffListRowImpl.java` view row class, the Java code to return its calculated value would reside in the `getLastCommaFirst()` method like this:

```
// In StaffListRowImpl.java
public String getFirstDotLast() {
  // Commented out this original line since we're not storing the value
  // return (String) getAttributeInternal(FIRSTDOTLAST);
  return getFirstName().substring(0,1)+". "+getLastName();
}
```

Note: In Section 26.8, "Implementing Automatic Attribute Recalculation", you'll learn a coding technique to cause calculated attributes at the entity row level to be re-calculated when one of the attribute values on which they depend is modified. You could adopt a very similar strategy at the view row level to cause automatic recalculation of calculated view object attributes, too.

7.6.6 What You May Need to Know About Transient Attributes

The view object includes the SQL expression for your SQL-calculated attribute in the `SELECT` list of its query at runtime. The database is the one that evaluates the expression and it returns the result as the value of that column in the query. The value gets reevaluated each time you execute the query.

7.7 Understanding How View Objects and Entity Objects Cooperate at Runtime

On their own, view objects and entity objects simplify two important jobs that every enterprise application developer needs to do:

- Work with SQL query results
- Modify and validate rows in database tables

Entity-based view objects can query any selection of data your end user needs to see or modify. Any data they are allowed to change is validated and saved by your reusable business domain layer. The key ingredients you provide as the developer are the ones that only *you* can know:

- You decide what business logic should be enforced in your business domain layer
- You decide what queries describe the data you need to put on the screen

These are the things that make *your* application unique. The built-in functionality of your entity-based view objects handle the rest of the implementation details. You've experimented above with entity-based view objects in the Business Components Browser and witnessed some of the benefits they offer, but now it's time to understand exactly how they work. This section walks step by step through a scenario of retrieving and modifying data through an entity-based view object, and points out the interesting aspects of what's going on behind the scenes. But before diving in deep, you need a bit of background on row keys and on what role the entity cache plays in the transaction, after which you'll be ready to understand the entity-based view object in detail.

7.7.1 Each View Row or Entity Row Has a Related Key

As shown in Figure 7–21, when you work with view rows you use the `Row` interface in the `oracle.jbo` package. It contains a method called `getKey()` that you can use to access the `Key` object that identifies any row. Notice that the `Entity` interface in the `oracle.jbo.server` package extends the `Row` interface. This relationship provides a concrete explanation of why the term *entity row* is so appropriate. Even though an entity row supports additional features for encapsulating business logic and handling database access, you can still treat any entity row as a `Row`.

Recall that both view rows and entity rows support either single-attribute or multi-attribute keys, so the `Key` object related to any given `Row` will encapsulate all of the attributes that comprise its key. Once you have a `Key` object, you can use the `findByKey()` method on any row set to find a row based on its `Key` object.

Figure 7–21 Any View Row or Entity Row Supports Retrieving Its Identifying Key

---

Note: When you define an entity-based view object, by default the primary key attributes for all of its entity usages are marked with their **Key Attribute** property set to `true`. It is best practice to subsequently disable the **Key Attribute** property for the key attributes from reference entity usages. Since view object attributes related to the primary keys of *updatable* entity usages must be part of the composite view row key, their **Key Attribute** property cannot be disabled.

7.7.2 What Role Does the Entity Cache Play in the Transaction

An application module is a transactional container for a logical unit of work. At runtime, it acquires a database connection using information from the named configuration you supply, and it delegates transaction management to a companion `Transaction` object. Since a logical unit of work may involve finding and modifying multiple entity rows of different types, the `Transaction` object provides an entity cache as a "work area" to hold entity rows involved in the current user's transaction. Each entity cache contains rows of a single entity type, so a transaction involving both the `User` and `ServiceHistory` entity objects holds the working copies of those entity rows in two separate caches.

By using an entity object's related entity definition, you can write code in an application module to find and modify existing entity rows. As shown in Figure 7–22, by calling `findByPrimaryKey()` on the entity definition for the `ServiceRequest` entity object, you can retrieve the row with that key. If it is not already in the entity cache, the entity object executes a query to retrieve it from the database. This query selects all of the entity object's persistent attributes from its underlying table, and find the row using an appropriate `WHERE` clause against the column corresponding to the entity object's primary key attribute. Subsequent attempts to find the same entity row by key during the same transaction will find it in the cache, avoiding a trip to the database. In a given entity cache, entity rows are indexed by their primary key. This makes finding and entity row in the cache a fast operation.

When you access related entity rows using association accessor methods, they are also retrieved from the entity cache, or are retrieved from the database if they are not in the cache. Finally, the entity cache is also the place where new entity rows wait to be saved. In other words, when you use the `createInstance2()` method on the entity definition to create a new entity row, it is added to the entity cache.

Figure 7–22 During the Transaction ServiceRequest, Entity Rows are Stored In ServiceRequest Entity Cache

When an entity row is created, modified, or removed, it is automatically enrolled in the transaction's list of pending changes. When you call `commit()` on the `Transaction` object, it processes its pending changes list, validating new or modified entity rows that might still be invalid. When the entity rows in the pending list are all valid, the `Transaction` issues a database `SAVEPOINT` and coordinates saving the entity rows to the database. If all goes successfully, it issues the final database `COMMIT` statement. If anything fails, the `Transaction` performs a `ROLLBACK TO SAVEPOINT` to allow the user to fix the error and try again.

The `Transaction` object used by an application module represents the working set of entity rows for a single end-user transaction. By design, it is *not* a shared, global cache. The database engine itself is an extremely efficient shared, global cache for multiple, simultaneous users. Rather than attempting to duplicate the 30+ years of fine-tuning that has gone into the database's shared, global cache functionality, ADF Business Components consciously embraces it. To refresh a single entity object's data from the database at any time, you can call its `refresh()` method. You can `setClearCacheOnCommit()` or `setClearCacheOnRollback()` on the `Transaction` object to control whether entity caches are cleared at commit or rollback. The defaults are `false` and `true`, respectively. The `Transaction` object also provides a `clearEntityCache()` method you can use to programmatically clear entity rows of a given entity type (or all types). By clearing an entity cache, entity rows of that type will be retrieved from the database fresh the next time they are found by primary key, or retrieved by an entity-based view object, as you'll see in the following sections.

7.7.3 Metadata Ties Together Cleanly Separated Roles of Data Source and Data Sink

When you want to venture beyond the world of finding an entity row by primary key and navigating related entities via association accessors, you turn to the entity-based view object to get the job done. In an entity-based view object, the view object and entity object play cleanly separated roles:

- The view object is the data *source*: it retrieves the data using SQL.
- The entity object is the data *sink*: it handles validating and saving data changes.

Because view objects and entity objects have cleanly separated roles, you can build a hundred different view objects — projecting, filtering, joining, sorting the data in whatever way your user interfaces require application after application — without any changes to the reusable entity object. In fact, in some larger development organizations, the teams responsible for the core business domain layer of entity objects might be completely separate from the ones who build specific application modules and view objects to tackle an end-user requirement. This extremely flexible, symbiotic relationship is enabled by metadata an entity-based view object encapsulates about how the `SELECT` list columns related to the attributes of one or more underlying entity objects.

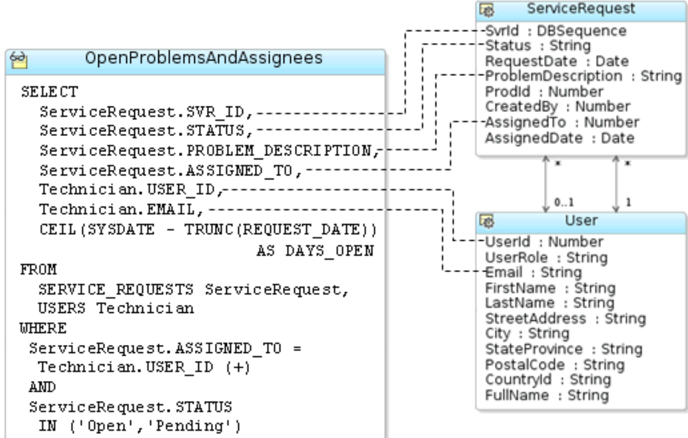
Imagine a new requirement arises where your end users are demanding a page to quickly see open and pending service requests. They want to see only the service request ID, status, and problem description; the technician assigned to resolve the request; and the number of days the request has been open. It should be possible to update the status and the assigned technician. Figure 7–23 shows a new entity-based view object named `OpenProblemsAndAssignees` that can support this new requirement.

The dotted lines in the figure represent the metadata captured in the view object's XML component definition that maps `SELECT` list columns in the query to attributes of the entity objects used in the view object.

A few things to notice about the view object and its query are:

- It joins data from a primary entity usage (ServiceRequest) with that from a secondary reference entity usage (User), based on the association related to the assigned technician you've seen in examples above
- It's using an outer join of ServiceRequest.ASSIGNED_TO = Technician.USER_ID (+)
- It includes a SQL-calculated attribute DaysOpen based on the SQL expression CEIL(SYSDATE - TRUNC(REQUEST_DATE))

Figure 7–23 View Object Encapsulates a SQL Query and Entity Attribute Mapping Metadata



7.7.4 What Happens When a View Object Executes Its Query

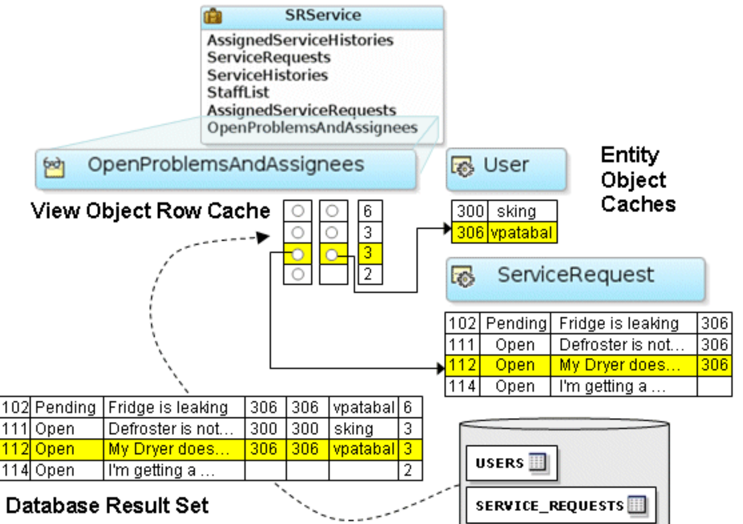
After adding an instance of OpenProblemsAndAssignees with the same name to the SRService's data model, you can see what happens at runtime when you execute the query. Like a read-only view object, an entity-based view object sends its SQL query straight to the database using the standard Java Database Connectivity (JDBC) API, and the database produces a result set. In contrast to its read-only counterpart, however, as the entity-based view object retrieves each row of the database result set, it partitions the row attributes based on which entity usage they relate to. This partitioning occurs by creating an entity object row of the appropriate type for each of the view object's entity usages, populating them with the relevant attributes retrieved by the query, and storing each of these entity rows in its respective entity cache. Then, rather than storing duplicate copies of the data, the view row simply *points* at the entity row parts that comprise it. As shown in Figure 7–24, the highlighted row in the result set is partitioned into a User entity row with primary key 306 and a ServiceRequest entity row with primary key 112. Since the SQL-calculated DaysOpen attribute is not related to any entity object, its value is stored directly in the view row.

The ServiceRequest entity row that was brought into the cache above using findByPrimaryKey() contained *all* attributes of the ServiceRequest entity object. In contrast, a ServiceRequest entity row created by partitioning rows from the OpenProblemsAndAssignees query result contains values only for attributes that appear in the query. It does *not* include the complete set of attributes. This partially populated entity row represents an important runtime performance optimization.

Since the ratio of rows retrieved to rows modified in a typical enterprise application is very high, bringing only the attributes into memory that you need to display can represent a big memory savings over bringing all attributes into memory all the time.

Finally, notice that in the queried row for service request `114` there is no assigned technician, so in the view row it has a `null` entity row part for its `User` entity object.

Figure 7–24 View Rows Are Partitioned into Entity Rows in Entity Caches



By partitioning queried data this way into its underlying entity row constituent parts, the first benefit you gain is that all of the rows that include some data queried about the user with `UserId` = `306` will display a consistent result when changes are made in the current transaction. In other words, if one view object allows the `Email` attribute of user `306` to be modified, then all rows in any entity-based view object showing the `Email` attribute for user `306` will update instantly to reflect the change. Since the data related to user `306` is stored exactly once in the `User` entity cache in the entity row with primary key `306`, any view row that has queried the user's `Email` attribute is just pointing at this single entity row.

Luckily, these implementation details are completely hidden from a client working with the rows in a view object's row set. Just as you did in the Business Components Browser, the client works with a view row, getting and setting the attributes, and is unaware of how those attributes might be related to entity rows behind the scenes.

7.7.5 What Happens When You Modify a View Row Attribute

You see above that among other rows, the `OpenProblemsAndAssignees` result set includes a row related to service request `112`. When a client attempts to update the status of service request `112` to the value `Closed`, ultimately a `setStatus("Closed")` method gets called on the view row. Figure 7–25 illustrates the steps that will occur to automatically coordinate this view row attribute modification with the underlying entity row:

1. The client attempts to set the `Status` attribute to the value `Closed`
2. Since `Status` is an entity-mapped attribute from the `ServiceRequest` entity usage, the view row delegates the attribute set to the appropriate underlying entity row in the `ServiceRequest` entity cache having primary key `112`.

3. Any attribute-level validation rules on the `Status` attribute at the `ServiceRequest` entity object get evaluated and will fail the operation if they don't succeed.

   Assume that some validation rule for the `Status` attribute programmatically references the `RequestDate` attribute (for example, to enforce a business rule that a `ServiceRequest` cannot be closed the same day it is opened). The `RequestDate` was not one of the `ServiceRequest` attributes retrieved by the query, so it is not present in the partially populated entity row in the `ServiceRequest` entity cache.

4. To ensure that business rules can always reference all attributes of the entity object, the entity object detects this situation and "faults-in" the entire set of `ServiceRequest` entity object attributes for the entity row being modified using the primary key (which must be present for each entity usage that participates in the view object.

5. After the attribute-level validations all succeed, the entity object attempts to acquire a lock on the row in the `SERVICE_REQUESTS` table before allowing the first attribute to be modified.

6. If the row can be locked, the attempt to set the `Status` attribute in the row succeeds and the value is changed in the entity row.

Note: The `jbo.locking.mode` configuration property controls how rows are locked. The default value is `pessimistic,` whose behavior corresponds to the steps described here. In `pessimistic` locking mode, the row must be lockable before any change is allowed to it in the entity cache. Typically, web applications will set this property to `optimistic` instead, so that rows aren't locked until transaction commit time.

Figure 7–25 Updating a View Row Attribute Delegates to Entity

7.7.6 What Happens When You Change a Foreign Key Attribute

If the user also updates the technician assigned to service request `112`, then something else interesting occurs. The request is currently assigned to `vpatabal`, who has user ID `306`. Assume that the end user sets the `AssignedTo` attribute to `300` to reassign the request to `sking`. As shown in Figure 7–26, behind the scenes, the following occurs:

1. The client attempts to set the `AssignedTo` attribute to the value `300`.
2. Since `AssignedTo` is an entity-mapped attribute from the `ServiceRequest` entity usage, the view row delegates the attribute set to the appropriate underlying entity row in the `ServiceRequest` entity cache having primary key `112`.
3. Any attribute-level validation rules on the `AssignedTo` attribute at the `ServiceRequest` entity object get evaluated and will fail the operation if they don't succeed.
4. The row is already locked, so the attempt to set the `AssignedTo` attribute in the row succeeds and the value is changed in the entity row.
5. Since the `AssignedTo` attribute on the `ServiceRequest` entity usage is associated to the reference entity usage named `Technician` to the `User` entity object, this change of foreign key value causes the view row to replace its current entity row part for user 306 with the entity row corresponding to the new `UserId = 300`. This effectively makes the view row for service request `112` point to the entity row for `sking`, so the value of the `Email` in the view row updates to reflect the correct reference information for this newly assigned technician.

Figure 7–26 After Updating a Foreign Key, View Row Points to a New Entity

7.7.7 What Happens When You Re-query Data

When you reexecute a view object's query, by default the view rows in its current row set are "forgotten" in preparation for reading in a fresh result set. This view object operation does not directly affect the entity cache, however. The view object then sends the SQL to the database and the process begins again to retrieve the database result set rows and partition them into entity row parts.

Note: Typically when you re-query, you are doing it in order to see the latest database information. If instead you want to avoid a database roundtrip by restricting your view object to querying only over existing entity rows in the cache, or over existing rows already in the view object's row set, Section 27.5, "Performing In-Memory Sorting and Filtering of Row Sets" explains how to do this.

7.7.7.1 Unmodified Attributes in Entity Cache are Refreshed During Re-query

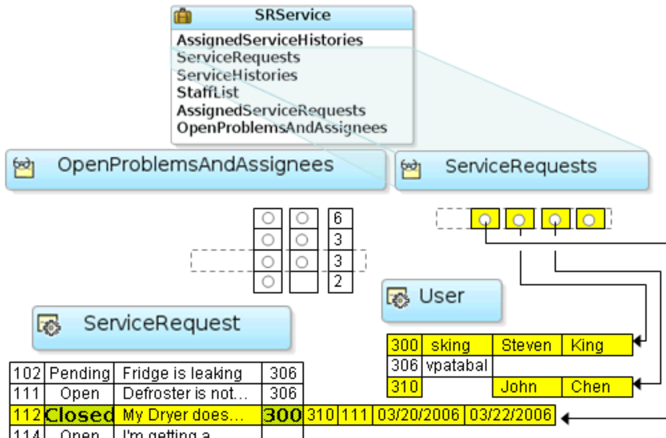
As part of this entity row partitioning process during a re-query, if an attribute on the entity row is unmodified, then its value in the entity cache is updated to reflect the newly queried value.

7.7.7.2 Modified Attributes in Entity Cache are Left Intact During Re-query

If the value of an entity row attribute has been *modified* in the current transaction, then during a re-query the entity row partitioning process does not refresh its value. Uncommitted changes in the current transaction are left intact so the end-user's logical unit of work is preserved. As with any entity attribute value, these pending modifications continue to be consistently displayed in any entity-based view object rows that reference the modified entity rows.

Figure 7–27 illustrates this scenario. Imagine that in the context of the current transaction's pending changes, a user "drills down" to a different page that uses the `ServiceRequests` view object instance to retrieve all details about service request `112`. That view object has four entity usages: a primary `ServiceRequest` usage, and three reference usages for `Product`, `User` (Technician), and `User` (Customer). When its query result is partitioned into entity rows, it ends up pointing at the same `ServiceRequest` entity row that the previous `OpenProblemsAndAssignees` view row had modified. This means the end user will correctly see the pending change, that the service request is assigned to `Steven King` in this transaction.

Figure 7–27 Overlapping Sets of Entity Attributes from Different View Objects are Merged in Entity Cache



7.7.7.3 Overlapping Subsets of Attributes are Merged During Re-query

Figure 7–27 also illustrates the situation that the `ServiceRequests` view object's query retrieves a *different* subset of reference information about users than the `OpenProblemsAndAssignees` did. The `ServiceRequests` queries up `FirstName` and `LastName` for a user, while the `OpenProblemsAndAssignees` view object queried the user's `Email`. The figure shows what happens at runtime in this scenario. If while partitioning the retrieved row, the entity row part contains a different set of attributes than the partially populated entity row that is already in the cache, the attributes get "merged". The result is a partially populated entity row in the cache with the *union* of the overlapping subsets of user attributes. In contrast, for John Chen (user 308) who wasn't in the cache already, the resulting new entity row contains only the `FirstName` and `LastName` attributes, but not the `Email`.

7.7.8 What Happens When You Commit the Transaction

Suppose the user is happy with her changes, and commits the transaction. As shown in Figure 7–28, there are two basic steps:

1. The `Transaction` object validates any invalid entity rows in its pending changes list.
2. The entity rows in the pending changes list are saved to the database.

The figure depicts a loop in step 1 before the act of validating one modified entity object might programmatically affect changes to other entity objects. Once the transaction has processed its list of invalid entities on the pending changes list, if the list is still nonempty, it will complete another pass through the list of invalid ones. It will attempt up to ten passes through the list. If by that point there are still invalid entity rows, it will throw an exception since this typically means you have an error in your business logic that needs to be investigated.

Figure 7–28 Committing the Transaction Validates Invalid Entities, Then Saves Them

7.7.9 Interactively Testing Multiuser Scenarios

The last aspect to understand about how view objects and entity objects cooperate involves two exceptions that can occur when working in a multiuser environment. Luckily, these are easy to simulate for testing purposes by simply starting up the Business Components Browser two times on the `SRService` application module (without exiting from the first instance of course). Try the following two tests to see how these multiuser exceptions can arises:

- In one Business Components Browser tester modify the status of an existing service request and tab out of the field. Then, in the other Business Components Browser window, try to modify the same service request in some way. You'll see that the second user gets the `oracle.jbo.AlreadyLockedException`

  Try repeating the test, but after overriding the value of `jbo.locking.mode` to be `optimistic` on the **Properties** page of the Business Components Browser **Connect** dialog. You'll see the error occurs at commit time for the second user instead of immediately.

- In one Business Components Browser tester modify the status of an existing service request and tab out of the field. Then, in the other Business Components Browser window, retrieve (but don't modify) the same status request. Back in the first window, commit the change. If the second user then tries to modify that same service request, you'll see that the second user gets the `oracle.jbo.RowInconsistentException`. The row has been modified and committed by another user since the second user retrieved the row into the entity cache

7.8 Working Programmatically with Entity-Based View Objects

From the point of view of a client accessing your application module's data model, the API's to work with a read-only view object and an entity-based view object are identical. The key functional difference is that entity-based view objects allow the data in a view object to be fully updatable. The application module that contains the entity-based view objects defines the unit of work and manages the transaction. This section presents four simple test client programs that work with the `SRService` application module to illustrate:

- Iterating master/detail/detail hierarchy
- Finding a row and updating a foreign key value
- Creating a new service request
- Retrieving the row Key identifying a row

7.8.1 Example of Iterating Master/Detail/Detail Hierarchy

Example 7–1 performs the following basic steps:

1. Finds the `StaffList` view object instance
2. Executes the query
3. Iterate over the resulting `StaffList` rows
4. Print the staff member's full name by getting the value of the calculated `FullName` attribute
5. Get related row set of `ServiceRequests` using a view link accessor attribute
6. Iterate over the `ServiceRequests` rows

7. Print out some service request attribute values
8. If the status is not `Closed`, then get the related row set of `ServiceHistories` using a view link accessor attribute
9. Iterate over the `ServiceHistories` rows
10. Print out some service request history attributes

Note: Other than having one additional level of nesting, this example uses the same API's that you saw in the `TestClient2` program that was iterating over master/detail read-only view objects in Section 5.10.4.2, "How to Access a Detail Collection Using the View Link Accessor".

Example 7–1 Iterating Master/Detail/Detail Hierarchy

```
package devguide.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
public class TestClient {
  public static void main(String[] args) {
    String        amDef = "devguide.model.SRService";
    String        config = "SRServiceLocal";
    ApplicationModule am =
      Configuration.createRootApplicationModule(amDef,config);
    // 1. Find the StaffList view object instance.
    ViewObject staffList = am.findViewObject("StaffList");
    // 2. Execute the query
    staffList.executeQuery();
    // 3. Iterate over the resulting rows
    while (staffList.hasNext()) {
      Row staffMember = staffList.next();
      // 4. Print the staff member's full name
      System.out.println("Staff Member: "+staffMember.getAttribute("FullName"));
      // 5. Get related rowset of ServiceRequests using view link accessor
      RowSet reqs = (RowSet)staffMember.getAttribute("ServiceRequests");
      // 6. Iterate over the ServiceRequests rows
      while (reqs.hasNext()) {
        Row svcreq = reqs.next();
        // 7. Print out some service request attribute values
        System.out.println(" ["+svcreq.getAttribute("Status")+"] "+
                           svcreq.getAttribute("SvrId")+": "+
                           svcreq.getAttribute("ProblemDescription"));
        if(!svcreq.getAttribute("Status").equals("Closed")) {
          // 8. Get related rowset of ServiceHistories
          RowSet hists = (RowSet)svcreq.getAttribute("ServiceHistories");
          // 9. Iterate over the ServiceHistories rows
          while (hists.hasNext()) {
            Row hist = hists.next();
            // 10. Print out some service request history attributes
            System.out.println("  "+hist.getAttribute("LineNo")+": "+
                               hist.getAttribute("Notes"));
          }
        }
      }
    }
```

```
    Configuration.releaseRootApplicationModule(am,true);
  }
}
```

Running the program produces the following output:

```
Staff Member: David Austin
 [Open] 104: Spin cycle not draining
  1: Researching issue
Staff Member: Bruce Ernst
 [Closed] 101: Agitator does not work
 [Pending] 102: Washing Machine does not turn on
  1: Called customer to make sure washer was plugged in...
  2: We should modify the setup instructions to include...
 [Open] 108: Freezer full of frost
  1: Researching issue
Staff Member: Alexander Hunold
 [Closed] 100: I have noticed that every time I do a...
 [Closed] 105: Air in dryer not hot
   :
```

7.8.2 Example of Finding a Row and Updating a Foreign Key Value

Example 7–2 performs the following basic steps:

1. Finds the `ServiceRequests` view object instance
2. Constructs a `Key` object to look up the row for service request number `101`
3. Uses `findByKey()` to find the row
4. Prints some service request attribute values
5. Tries to assign the *illegal* value `Reopened` to the `Status` attribute

   Since view object rows cooperate with entity objects, the validation rule on the `Status` attribute throws an exception, preventing this illegal change.
6. Sets the `Status` to a legal value of `Open`
7. Prints the value of the `Status` attribute to show it was updated successfully
8. Prints the current value of the assigned technician's email
9. Reassigns the service request to technician number `303` (Alexander Hunold) by setting the `AssignedTo` attribute
10. Shows the value of the reference information (`TechnicianEmail`) reflecting a new technician
11. Cancels the transaction by issuing a rollback

Example 7–2 Finding and Updating a Foreign Key Value

```
package devguide.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
public class TestFindAndUpdate {
  public static void main(String[] args) {
    String        amDef = "devguide.model.SRService";
    String        config = "SRServiceLocal";
    ApplicationModule am =
      Configuration.createRootApplicationModule(amDef,config);
    // 1. Find the ServiceRequests view object instance
    ViewObject vo = am.findViewObject("ServiceRequests");
    // 2. Construct a new Key to find ServiceRequest# 101
    Key svcReqKey = new Key(new Object[]{101});
    // 3. Find the row matching this key
    Row[] reqsFound = vo.findByKey(svcReqKey,1);
    if (reqsFound != null && reqsFound.length > 0) {
      Row svcReq = reqsFound[0];
      // 4. Print some service request information
      String curStatus = (String)svcReq.getAttribute("Status");
      System.out.println("Current status is: "+curStatus);
      try {
        // 5. Try setting the status to an illegal value
        svcReq.setAttribute("Status","Reopened");
      }
      catch (JboException ex) {
        System.out.println("ERROR: "+ex.getMessage());
      }
      // 6. Set the status to a legal value
      svcReq.setAttribute("Status","Open");
      // 7. Show the value of the status was updated successfully
      System.out.println("Current status is: "+svcReq.getAttribute("Status"));
      // 8. Show the current value of the assigned technician
      System.out.println("Assigned: "+svcReq.getAttribute("TechnicianEmail"));
      // 9. Reassign the service request to technician # 303
      svcReq.setAttribute("AssignedTo",303); // Alexander Hunold (technician)
      // 10. Show the value of the reference info reflects new technician
      System.out.println("Assigned: "+svcReq.getAttribute("TechnicianEmail"));
      // 11. Rollback the transaction
      am.getTransaction().rollback();
      System.out.println("Transaction cancelled");
    }
    Configuration.releaseRootApplicationModule(am,true);
  }
}
```

Running this example produces the following output:

```
Current status is: Closed
ERROR: The status must be Open, Pending, or Closed
Current status is: Open
Assigned: bernst
Assigned: ahunold
Transaction cancelled
```

7.8.3 Example of Creating a New Service Request

Example 7–3 performs the following basic steps:

1. Find the `ServiceRequests` view object instance
2. Creates a new row and inserts it into the row set
3. Shows the effect of entity object related defaulting for `Status` attribute
4. Sets values of some required attributes in the new row
5. Commits the transaction
6. Retrieves and displays the trigger-assigned service request ID

Example 7–3 Creating a New Service Request

```
package devguide.client;
import java.sql.Timestamp;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.DBSequence;
import oracle.jbo.domain.Date;
public class TestCreatingServiceRequest {
  public static void main(String[] args) throws Throwable {
    String amDef = "devguide.model.SRService";
    String config = "SRServiceLocal";
    ApplicationModule am =
      Configuration.createRootApplicationModule(amDef, config);
    // 1. Find the ServiceRequests view object instance.
    ViewObject svcReqs = am.findViewObject("ServiceRequests");
    // 2. Create a new row and insert it into the row set
    Row newSvcReq = svcReqs.createRow();
    svcReqs.insertRow(newSvcReq);
    // 3. Show effect of entity object defaulting for Status attribute
    System.out.println("Status defaults to: "+newSvcReq.getAttribute("Status"));
    // 4. Set values for some of the required attributes
    newSvcReq.setAttribute("CreatedBy",308); // Nancy Greenberg (user)
    Date now = new Date(new Timestamp(System.currentTimeMillis()));
    newSvcReq.setAttribute("RequestDate",now);
    newSvcReq.setAttribute("ProdId",119); // Ice Maker
    newSvcReq.setAttribute("ProblemDescription","Cubes melt immediately");
    // 5. Commit the transaction
    am.getTransaction().commit();
    // 6. Retrieve and display the trigger-assigned service request id
    DBSequence id = (DBSequence)newSvcReq.getAttribute("SvrId");
    System.out.println("Thanks, reference number is "+id.getSequenceNumber());
    Configuration.releaseRootApplicationModule(am, true);
  }
}
```

Running this example produces the following results:

```
Status defaults to: Open
Thanks, reference number is 200
```

7.8.4 Example of Retrieving the Row Key Identifying a Row

Example 7–4 performs the following basic steps:

1. Finds the `ServiceRequests` view object
2. Constructs a key to find service request number 101
3. Finds the `ServiceRequests` row with this key
4. Displays the key of the `ServiceRequests` row
5. Accesses the row set of `ServiceHistories` using the view link accessor attribute
6. Iterates overs the `ServiceHistories` row
7. Displays the key of each `ServiceHistories` row

Example 7–4 Retrieving the Row Key Identifying a Row

```
package devguide.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
public class TestFindAndShowKeys {
  public static void main(String[] args) {
    String amDef = "devguide.model.SRService";
    String config = "SRServiceLocal";
    ApplicationModule am =
      Configuration.createRootApplicationModule(amDef, config);
    // 1. Find the ServiceRequests view object
    ViewObject vo = am.findViewObject("ServiceRequests");
    // 2. Construct a key to find service request number 101
    Key svcReqKey = new Key(new Object[] { 101 });
    // 3. Find the ServiceRequests row with this key
    Row[] reqsFound = vo.findByKey(svcReqKey, 1);
    if (reqsFound != null && reqsFound.length > 0) {
      Row svcReq = reqsFound[0];
      // 4. Display the key of the ServiceRequests row
      showKeyFor(svcReq);
      // 5. Access row set of ServiceHistories using view link accessor
      RowSet histories = (RowSet)svcReq.getAttribute("ServiceHistories");
      // 6. Iterate over the ServiceHistories row
      while (histories.hasNext()) {
        Row historyRow = histories.next();
        // 7. Display the key of the current ServiceHistories row
        showKeyFor(historyRow);
      }
    }
    Configuration.releaseRootApplicationModule(am, true);
  }
  private static void showKeyFor(Row r) {
    // get the key for the row passed in
    Key k = r.getKey();
    // format the key as "(val1,val2)"
    String keyAttrs=formatKeyAttributeNamesAndValues(k);
    // get the serialized string format of the key, too
    String keyStringFmt = r.getKey().toStringFormat(false);
    System.out.println("Key "+keyAttrs+" has string format "+keyStringFmt);
```

```
  }
  // Build up "(val1,val2)" string for key attributes
  private static String formatKeyAttributeNamesAndValues(Key k) {
    StringBuffer sb = new StringBuffer("(");
    int attrsInKey = k.getAttributeCount();
    for (int i = 0; i < attrsInKey;i++) {
      if (i > 0 ) sb.append(",");
      sb.append(k.getAttributeValues()[i]);
    }
    sb.append(")");
    return sb.toString();
  }
}
```

Running the example produces the following results. Notice that the serialized string format of a key is a hexadecimal number that includes information in a single string that represents all the attributes in a key.

```
Key (101) has string format 000100000003313031
Key (101,1) has string format 000200000003C2020200000002C102
Key (101,2) has string format 000200000003C2020200000002C103
```

7.9 Summary of Difference Between Entity-Based View Objects and Read-Only View Objects

You now know that view objects can either be related to underlying entity objects or not. This section helps summarize the difference between the runtime behavior of these two fundamental kinds of view objects.

7.9.1 Runtime Features Unique to Entity-Based View Objects

When a view object has one or more underlying entity usages you can create new rows, and modify or remove queried rows. The entity-based view object coordinates with underlying entity objects to enforce business rules and to permanently save the changes. In addition, you've seen that entity-based view objects:

- Immediately reflect pending changes made to relevant entity object attributes made through other view objects in the same transaction
- Initialize attribute values in newly created rows to the values from the underlying entity object attributes
- Reflect updated reference information when foreign key attribute values are changed

7.9.2 View Objects with No Entity Usage Are Read-Only

View objects with no entity usage are read-only, do not pick up entity-derived default values, do not reflect pending changes, and do not reflect updated reference information. You need to decide what kind of functionality your application requires and design the view object accordingly. Typically view objects used for SQL-based validation purposes, and for displaying the list of valid selections in a dropdown list, can be read-only. There is a small amount of runtime overhead associated with the coordination between view object rows and entity object rows, so if you don't need any of the functionality offered by an entity-mapped view object, you can slightly increase performance by using a read-only view object with no related entity objects.

7.9.3 What You May Need to Know About Enabling View Object Key Management for Read-Only View Objects

An entity-based view object delegates the task of finding rows by key to its underlying entity row parts. When you use the `findByKey()` method to find a view row by key, the view row turns around and uses the entity definition's `findByPrimaryKey()` to find each entity row contributing attributes to the view row key.

This scheme is not possible for a read-only view object since it has no underlying entity row to which to delegate the job. Since you might use read-only view objects to quickly iterate over query results without needing any of the additional features provided by the entity-based view object, a read-only view object does not assume you want to incur the slight additional overhead of managing rows by key at the level of the view object's row set.

In order to successfully be able to use the `findByKey()` method on a read-only view object, you need to perform two additional steps:

1. Ensure that at least one attribute in the view object has the **Key Attribute** property set
2. Enable a custom Java class for the view object, and override its `create()` method to call `setManageRowsByKey(true)`after calling `super.create()` like this:

```
// In custom Java class for read-only view object
public void create() {
  super.create();
  setManageRowsByKey(true);
}
```

Note: In an application using the ADF Model layer for data binding to a read-only view object, the successful operation of the ADF Faces table — or other controls that allow the end user to set the current row by clicking in the page — require this additional step. This also applies to the successful use of built-in binding layer actions like `setCurrentRowWithKey` or `setCurrentRowWithKeyValue` on a read-only view object. These all boil down to calling `findByKey()` under the covers.

Section 25.3.2, "Implementing Generic Functionality Using Runtime Metadata" describes a generic technique you can use to avoid having to remember to do this on all of your read-only view objects on which you want `findByKey()` to work as expected.

8 Implementing Business Services with Application Modules

This chapter describes how to implement the overall functionality of your business services using the combination of application modules, view objects, and entity objects.

This chapter includes the following sections:

- Section 8.1, "Introduction to Application Modules"
- Section 8.2, "Creating an Application Module"
- Section 8.3, "Adding a Custom Service Method"
- Section 8.4, "Publishing Custom Service Methods to Clients"
- Section 8.5, "Working Programmatically with an Application Module's Client Interface"
- Section 8.6, "Overriding Built-in Framework Methods"
- Section 8.7, "Creating an Application Module Diagram for Your Business Service"
- Section 8.8, "Supporting Multipage Units of Work"
- Section 8.9, "Deciding on the Granularity of Application Modules"

8.1 Introduction to Application Modules

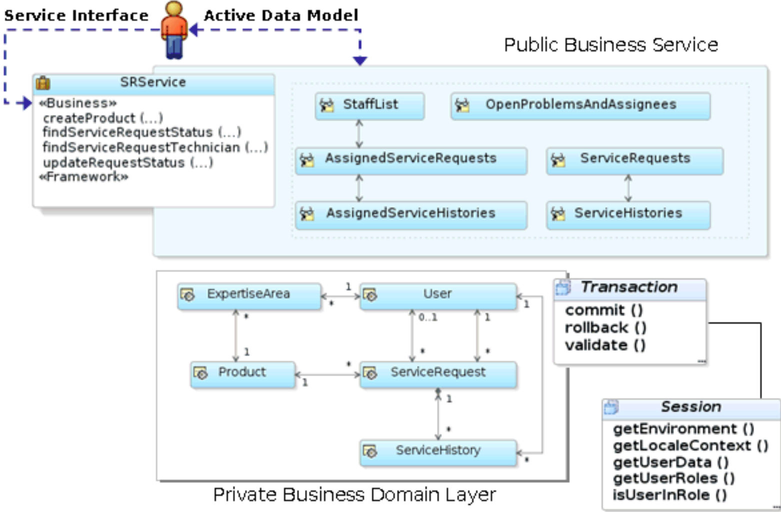
An application module is Oracle ADF Business Component component that encapsulates the business service methods and active data model for a logical unit of work related to an end-user task.

By the end of this chapter, you'll understand all the concepts illustrated in Figure 8–1, and more:

- You use instances of view objects in an application module to define its active data model.
- You write service methods to encapsulate task-level business logic.
- You expose selected methods on the service interface for clients to call.
- You use application modules from a pool during a logical transaction that can span multiple web pages.

- Your application module works with a `Transaction` object that acquires a database connection and coordinates saving or rolling back changes made to entity objects.
- The related `Session` object provides runtime information about the current application user

Figure 8–1 Application Module Is a Business Service Component Encapsulating a Unit of Work



Previous chapters explained the details of including view object instances in an application module's data model and of how client-accessible view objects cooperate internally with the entity objects in your reusable business domain layer. This chapter describes how to combine business service methods with that data model to implement a complete business service.

> **Note:** The examples in this chapter use the same basic `SRService` application module from Chapter 7, "Building an Updatable Data Model With Entity-Based View Objects", including the entity-based view objects shown in Figure 8–1. To experiment with a working version, download the `DevGuideExamples` workspace from *Example Downloads* page at
> `http://otn.oracle.com/documentation/jdev/b25947_01`
> and see the `ApplicationModules` project.

8.2 Creating an Application Module

In a large application, you typically create one application module to support each coarse-grained end-user task. In a smaller-sized application like the SRDemo application, you may decide that creating a single application module is adequate to handle the needs of the complete set of application functionality. Section 8.9, "Deciding on the Granularity of Application Modules" provides additional guidance on this subject.

8.2.1 Creating an Application Module

To create an application module:

1. Open **Create Application Module Wizard**. The wizard is available from the **New Gallery** in the **Business Tier > ADF Business Components** category.
2. In step 1 on the **Name** pane, provide a package name and an application module name.
3. In step 2 on the **Data Model** page, include instances of view object you have previously defined and adjust the view object instance names to be exactly what you want clients to see. Then click **Finish**.

For more step by step details, see Section 5.3, "Using a View Object in an Application Module's Data Model".

8.2.2 What Happens When You Create an Application Module

When you create an application module, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, given an application module named `SRService` in the `devguide.model` package, the XML file created will be `./devguide/model/SRService.xml` under the project's source path. This XML file contains the information needed at runtime to re-create the view object instances in the application module's data model. If you're curious to see its contents, you can see the XML file for the application module by selecting the view object in the **Application Navigator** and looking in the corresponding **Sources** folder in the Structure Window. Double-clicking on the `SRService.xml` node will open the XML in an editor so that you can inspect it.

> **Note:** If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom application module class `SRServiceImpl.java`.

8.2.3 Editing an Existing Application Module

After you've created a new application module, you can edit any of its settings by using the Application Module Editor. Choose the **Edit** menu option on the right-mouse context menu in the Application Navigator, or double-click on the application module, to launch the editor. By visiting the different panels of the editor, you can adjust the data model, Java generation settings, remote deployment options, client interface methods, and custom properties.

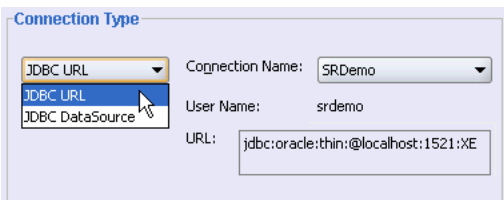
8.2.4 Configuring Your Application Module Database Connection

You configure your application module to use a database connection by identifying either a Java Database Connectivity (JDBC) URL or a JDBC Datasource name in the **Connection Type** section of the **Configuration** editor (see Figure 5–12).

8.2.4.1 Using a JDBC URL Connection Type

The default *YouAppModule*Local configuration uses a JDBC URL connection. It is based on the named connection definition set on the **Business Components** page of the Project Properties dialog for the project containing your application module. Figure 8–2 shows what this section would look like in a configuration using a JDBC URL connection based on the SRDemo connection name. When you use a JDBC URL connection type as the SRServiceLocalTesting configuration does in the SRDemo application, you can use the application module in any context where Java can run. In other words, it is not restricted to running inside a J2EE application server with this connection type.

Figure 8–2 Connection Type Setting in Configuration Editor



Note: See Section 29.5.2, "Database Connection Pools" and Section 29.7, "Database Connection Pool Parameters" for more information on how database connection pools are used and how you can tune them.

8.2.4.2 Using a JDBC Datasource Connection Type

The other type of connection you can use is a JDBC datasource. You define a JDBC datasource as part of your application server configuration information, and then the application module looks up the resource at runtime using a logical name. You define the datasource connection details in two parts:

- a logical part that uses standard J2EE deployment descriptors to define the datasource name the application will use at runtime,
- a physical part that is application-server specific which maps the logical datasource name to the physical connection details.

For example, Example 8–1 shows the <resource-ref> tags in the web.xml file of the SRDemo application. These define two logical datasources named jdbc/SRDemoDS and jdbc/SRDemoCoreDS. When you use a JDBC datasource connection for your application module, you reference this logical connection name after the prefix java:comp/env. Accordingly, if you inspect the SRServiceLocal configuration for the SRService application module in the SRDemo application, you'll see that the value of its **JDBC Datasource Name** field is java:comp/env/jdbc/SRDemoDS.

Example 8–1 Logical Datasource Resource Names Defined in web.xml

```
<!-- In web.xml -->
<resource-ref>
  <res-ref-name>jdbc/SRDemoDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
<resource-ref>
  <res-ref-name>jdbc/SRDemoCoreDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Once you've defined the logical datasource names in `web.xml` and referenced them in a configuration, you then need to include additional, server-specific configuration files to map the logical datasource names to physical connection definitions in the target application server. Example 8–2 shows the contents of the SRDemo application's `data-sources.xml` file. This file is specific to the Oracle Containers for J2EE (OC4J) and defines the physical details of the datasource connection pools and connection names. You would need to provide a similar file for different J2EE application servers, and the file would have a vendor-specific format.

Example 8–2 Datasource Connection Details Defined External to Application

```
<data-sources ... >
    <connection-pool name="jdev-connection-pool-SRDemo">
        <connection-factory
            factory-class="oracle.jdbc.pool.OracleDataSource"
            password="->DataBase_User_8PQ34e6j3MDg3UcQD-BZktUAK-QpepGp"
            user="srdemo" url="jdbc:oracle:thin:@localhost:1521:XE"/>
    </connection-pool>
    <managed-data-source name="jdev-connection-managed-SRDemo"
                        jndi-name="jdbc/SRDemoDS"
                        connection-pool-name="jdev-connection-pool-SRDemo"/>
    <native-data-source name="jdev-connection-native-SRDemo"
                        jndi-name="jdbc/SRDemoCoreDS" ... />
</data-sources>
```

The last step in the process involves mapping the physical connection details to the logical resource references for the datasource. In the OC4J server, you accomplish this step using the `orion-web.xml` file shown in Example 8–3.

Example 8–3 Server-Specific File Maps Logical Datasource to Physical Datasource

```
<orion-web-app ... >
  <resource-ref-mapping location="jdbc/SRDemoDS" name="jdbc/SRDemoDS"/>
  <resource-ref-mapping location="jdbc/SRDemoCoreDS" name="jdbc/SRDemoCoreDS"/>
</orion-web-app>
```

Once these datasource configuration details are in place, you can successfully use your application module in a J2EE application server as part of a web application.

8.2.5 Managing Your Application Module's Runtime Configurations

In addition to creating the application module XML component definition, JDeveloper also adds a default configuration named `SRServiceLocal` to the `bc4j.xcfg` file in the subdirectory named `common` relative to directory containing the `SRService.xml` file. To manage your application module's configurations, select it in the Application Navigator and choose **Configurations...** from the right-mouse context menu.

8.2.6 What You Might Need to Know About Application Module Connections

When testing your application module with the Business Component Browser, you should be aware of the connection configuration.

8.2.6.1 The Business Components Browser Requires a JDBC URL Connection

In previous chapters you've learned how valuable the Business Components Browser tool can be for testing your application module's data model interactively. Since it runs outside the context of a J2EE application server, it cannot test application modules using a configuration that depends on a JDBC datasource. The solution is simply to test the application module by selecting a configuration that uses a JDBC URL connection. You do this by choosing it from the **Business Component Configuration Name** dropdown list on the Connect dialog of the Business Components Browser.

8.2.6.2 Testing the SRService Application Module in the Business Components Browser

To test the SRDemo application’s SRService application module using the Business Components Browser, choose the `SRServiceLocalTesting` configuration. Incidentally, this is the same configuration used by the JUnit tests in the `UnitTests` project in the workspace. The tests also run outside of a J2EE application server so for the same reason as the Business Components Browser cannot use a configuration with a JDBC datasource connection type.

8.3 Adding a Custom Service Method

An application module can expose its data model of view objects to clients without requiring any custom Java code. This allows client code to use the `ApplicationModule`, `ViewObject`, `RowSet`, and `Row` interfaces in the `oracle.jbo` package to work directly with any view object in the data model. However, just because you *can* programmatically manipulate view objects any way you want to in client code doesn't mean that doing so is always a best practice.

Whenever the programmatic code that manipulates view objects is a logical aspect of implementing your complete business service functionality, you should encapsulate the details by writing a custom method in your application module's Java class. This includes code that:

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

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

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

8.3.1 How to Generate a Custom Class for an Application Module

To add a custom service method to your application module, you must first enable a custom Java class for it. If you have configured your IDE-level **Business Components** Java generation preferences to automatically generate an application module class, you're set. If you're not sure whether your application module has a custom Java class, as shown in Figure 8–3, check the context menu for the **Go to Application Module Class** option. This option lets you quickly navigate to your application module's custom class, but if you don't see the option in the menu then that means your application module is currently an XML-only component.

Figure 8–3 Quickly Navigating to an Application Module's Custom Java Class

To enable the class, open the Application Module Editor, visit the **Java** page, select the **Generate Java File** checkbox for an **Application Module Class**, then click **OK** to finish the wizard.

8.3.2 What Happens When You Generate a Custom Class for an Application Module

For an application module named `devguide.model.SRService`, the default name for its custom Java file will be `SRServiceImpl.java`. The file is created in the same `./devguide/model` directory as the component's XML component definition file.

The Java generation options for the application module continue to be reflected on the **Java** page on subsequent visits to the Application Module Editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom java classes up to date with any changes you make in the editor. If later you decide you didn't require a custom Java file for any reason, unchecking the relevant option in the **Java** page causes the custom Java file to be removed.

8.3.3 What You May Need to Know About Default Code Generation

By default, the application module Java class will look similar to what you see in Example 8–4 when you've first enabled it. Of interest, it contains:

- Getter methods for each view object instance in the data model
- A `main()` method allowing you to debug the application module using the Business Components Browser

Example 8–4 Default Application Module Generated Code

```
package devguide.model;
import devguide.model.common.SRService;
import oracle.jbo.server.ApplicationModuleImpl;
import oracle.jbo.server.ViewLinkImpl;
import oracle.jbo.server.ViewObjectImpl;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
public class SRServiceImpl extends ApplicationModuleImpl {
  /** This is the default constructor (do not remove) */
  public SRServiceImpl() { }
  /** Sample main for debugging Business Components code using the tester */
  public static void main(String[] args) {
    launchTester("devguide.model", /* package name */
                 "SRServiceLocal"  /* Configuration Name */);
  }
  /** Container's getter for YourViewObjectInstance1 */
  public ViewObjectImpl getYourViewObjectInstance1() {
    return (ViewObjectImpl)findViewObject("YourViewObjectInstance1");
  }

  // ... Additional ViewObjectImpl getters for each view object instance

  // ... ViewLink getters for view link instances here
}
```

As shown in Figure 8–4, your application module class extends the base ADF `ApplicationModuleImpl` class to inherit all the default behavior before adding your custom code.

Figure 8–4 Your Custom Application Module Class Extends ApplicationModuleImpl

8.3.4 Debugging the Application Module Using the Business Components Tester

As you learn more about the overall business service role of the application module in this chapter, you'll find it useful to exercise your application module under the JDeveloper debugger while using the Business Components Browser as the testing interface. To debug an application module using the tester, select the application module in the Application Navigator and then either:

- In the Application Navigator, right-click the application module and choose **Go to Application Module Class** from the context menu.
- Select **Debug** from the right-mouse context menu in the code editor

8.3.5 How to Add a Custom Service Method to an Application Module

To add a custom service method to an application module, simply navigate to application module's custom class and type in the Java code for a new method into the application module's Java implementation class. Use the following guidelines to decide on the appropriate visibility for the method. If you will use the method only inside this component's implementation as a helper method, make the method `private`. If you want to allow eventual subclasses of your application module to be able to invoke or override the method, make it `protected`. If you need clients to be able to invoke it, it must be `public`. See the examples of the `SRServiceImpl` methods in previous chapters.

Note: The `SRService` application module in this chapter is using the strongly typed, custom entity object classes that you saw illustrated in the `SRServiceImpl2.java` example at the end of Chapter 6, "Creating a Business Domain Layer Using Entity Objects".

Example 8–5 shows a `private retrieveServiceRequestById()` helper method in the `SRServiceImpl.java` class for the `SRService` application module. It uses the `static getDefinition()` method of the `ServiceRequestImpl` entity object class to access its related entity definition, uses the `createPrimaryKey()` method on the entity object class to create an appropriate Key object to look up the service request, then used the `findByPrimaryKey()` method on the entity definition to find the entity row in the entity cache. It returns an instance of the strongly typed `ServiceRequestImpl` class, the custom Java class for the `ServiceRequest` entity object.

Example 8–5 Private Helper Method in Custom Application Module Class

```
// In devguide.model.SRServiceImpl class
/*
 * Helper method to return a ServiceRequest by Id
 */
private ServiceRequestImpl retrieveServiceRequestById(long requestId) {
  EntityDefImpl svcReqDef = ServiceRequestImpl.getDefinitionObject();
  Key svcReqKey =
    ServiceRequestImpl.createPrimaryKey(new DBSequence(requestId));
  return (ServiceRequestImpl)svcReqDef.findByPrimaryKey(getDBTransaction(),
                                                         svcReqKey);
}
```

Example 8–6 shows a `public createProduct()` method that allows the caller to pass in a name and description of a product to be created. It uses the `getDefinition()` method of the `ProductImpl` entity object class to access its related entity definition, uses the `createInstance2()` method to create a new `ProductImpl` entity row whose `Name` and `Description` attributes it populates with the parameter values passed in before committing the transaction.

Example 8–6 Public Method in Custom Application Module Class

```
/*
 * Create a new Product and Return its new id
 */
public long createProduct(String name, String description) {
  EntityDefImpl productDef = ProductImpl.getDefinitionObject();
  ProductImpl newProduct =
    (ProductImpl)productDef.createInstance2(getDBTransaction(),null);
  newProduct.setName(name);
  newProduct.setDescription(description);
  try {
    getDBTransaction().commit();
  }
  catch (JboException ex) {
    getDBTransaction().rollback();
    throw ex;
  }
  DBSequence newIdAssigned = newProduct.getProdId();
  return newIdAssigned.getSequenceNumber().longValue();
}
```

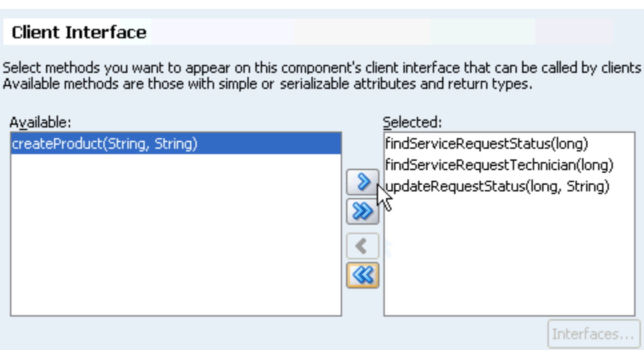
8.4 Publishing Custom Service Methods to Clients

When you add a `public` custom method to your application module class, if you want clients to be able to invoke it, you need to include the method on the application module's client interface.

8.4.1 How to Publish Custom Service Methods to Clients

To include a public method from your application module's custom Java class on the client interface, use the **Client Interface** page of the Application Module Editor. As shown in Figure 8–5, select one or more desired methods from the **Available** list and press **>** to shuttle them into the **Selected** list. Then click **OK** to dismiss the editor.

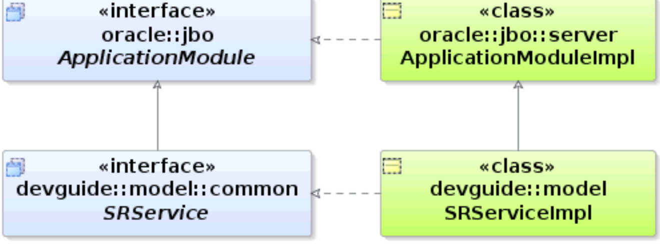
Figure 8–5 Adding a Public Method to an Application Module's Client Interface



8.4.2 What Happens When You Publish Custom Service Methods to Clients

When you publish custom service methods on the client interface, as illustrated in Figure 8–6, JDeveloper creates a Java interface with the same name as the application module in the `common` subpackage of the package in which your application module resides. For an application module named `SRService` in the `devguide.model` package, this interface will be named `SRService` and reside in the `devguide.model.common` package. The interface extends the base `ApplicationModule` interface in the `oracle.jbo` package, reflecting that a client can access all of the base functionality that your application module inherits from the `ApplicationModuleImpl` class.

Figure 8–6 Custom Service Interface Extends the Base ApplicationModule Interface



As shown in Example 8–7, the `SRService` interface includes the method signatures of all of the methods you've selected to be on the client interface of your application module.

Example 8–7 Custom Service Interface Based on Methods Selected in the Client Interface Panel

```
package devguide.model.common;
import oracle.jbo.ApplicationModule;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---------------------------------------------------------------------
public interface SRService extends ApplicationModule {
  long createProduct(String name, String description);
  String findServiceRequestStatus(long requestId);
  String findServiceRequestTechnician(long requestId);
  void updateRequestStatus(long requestId, String newStatus);
}
```

Each time you add or remove methods from the **Selected** list in the **Client Interface** page, the corresponding service interface file is updated automatically. JDeveloper also generates a companion client proxy class that is used when you deploy your application module for access by a remote client. For the `SRService` application module in this example, the client proxy file is called `SRServiceClient` and it is created in the `devguide.model.client` subpackage.

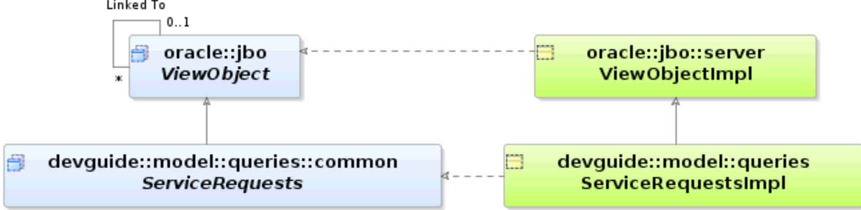
Note: After adding new custom methods to the client interface, if your new custom methods do not appear to be available using JDeveloper's *Code Insight* context-sensitive statement completion when trying to use the custom interface from client code, try recompiling the generated client interface. To do this, select the application module in the Application Navigator, select the source file for the interface of the same name in the Structure window, and choose **Rebuild** from the context menu. Consider this tip for new custom methods added to view objects and view rows as well.

8.4.3 How to Generate Client Interfaces for View Objects and View Rows

In addition to generating a client interface for your application module, it's also possible to generate strongly typed client interfaces for working with the other key client objects that you can customize. In a manner identical to how you have learned to do for application modules, you can open the **Client Interface** and **Client Row Interface** pages of the View Object Editor to add custom methods to the view object client interface and the view row client interface, respectively.

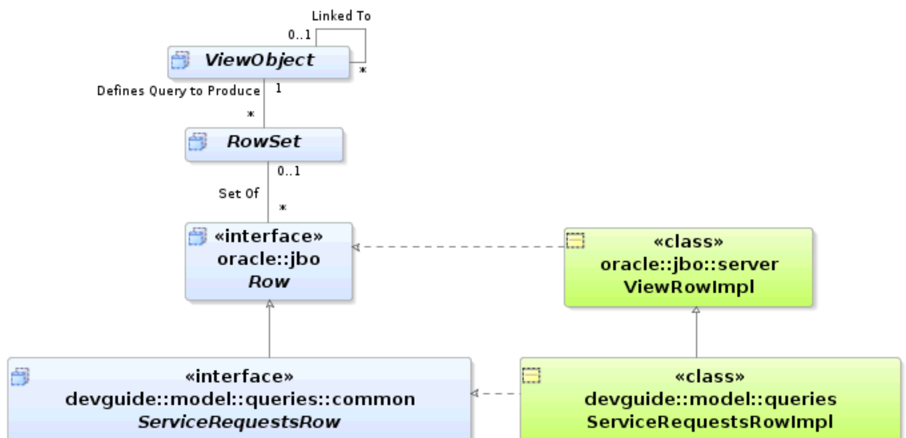
If for the `ServiceRequests` view object in the `devguide.model.queries` package you were to enable the generation of a custom view object Java class and add one or more custom methods to the view object client interface, JDeveloper would generate the `ServiceRequestsImpl` class and `ServiceRequests` interface, as shown in Figure 8–7. As with the application module custom interface, notice that it gets generated in the `common` subpackage.

Figure 8–7 Custom View Object Interface Extends the Base ViewObject Interface



Likewise, if for the same view object you were to enable the generation of a custom view row Java class and add one or more custom methods to the view row client interface, JDeveloper would generate the `ServiceRequestsRowImpl` class and `ServiceRequestsRow` interface, as shown in Figure 8–8.

Figure 8–8 Custom View Row Interface Extends the Base Row Interface



8.4.4 What You May Need to Know About Method Signatures on the Client Interface

You can include any custom method in the client interface that obeys these rules:

- If the method has a non-`void` return type, the type must be serializable.
- If the method accepts any parameters, all their types must be serializable.
- If the method signature includes a throws clause, the exception must be an instance of `JboException` in the `oracle.jbo` package.

In other words, all the types in its method signature must implement the `java.io.Serializable` interface, and any checked exceptions must be `JboException` or its subclass. Your method can throw any unchecked exception — `java.lang.RuntimeException` or a subclass of it — without disqualifing the method from appearing on the application module's client interface.

> **Note:** If the method you've added to the application module class doesn't appear in the **Available** list, first check to see that it doesn't violate any of the rules above. If it seems like it should be a legal method to appear in the list, try recompiling the application module class before visiting the Application Module Editor again.

8.4.5 What You May Need to Know About Passing Information from the Data Model

The private implementation of an application module custom method can easily refer to any view object instance in the data model using the generated accessor methods. By calling the `getCurrentRow()` method on any view object, it can access the same current row for any view object that the client user interface sees as the current row. Due to this benefit, in some cases while writing application module business service methods there is no need to pass in parameters from the client if they would only be passing in values from the current rows of other view object instances in the same application module's data model.

For example, the `createServiceRequest()` method in the SRDemo application's `SRService` application module accepts no parameters. Internally it calls `getGlobals().getCurrentRow()` to access the current row of the Globals view object instance. Then it uses the strongly typed accessor methods on the row to access the values of the `ProblemDescription` and `ProductId` attributes to set them as the values of corresponding attributes in a newly-created `ServiceRequest` entity object row.

Example 8–8 Using View Object Accessor Methods to Access a Current Row

```
// In SRServiceImpl.java, createServiceRequest() method
GlobalsRowImpl globalsRow = (GlobalsRowImpl)getGlobals().getCurrentRow();
newReq.setProblemDescription(globalsRow.getProblemDescription());
newReq.setProdId(globalsRow.getProductId());
```

8.5 Working Programmatically with an Application Module's Client Interface

After publishing methods on your application module's client interface, you can invoke those methods from a client.

8.5.1 How to Work Programmatically with an Application Module's Client Interface

To work programmatically with an application module's client interface, do the following:

- Cast `ApplicationModule` to the more specific client interface.
- Call any method on the interface.

> **Note:** For simplicity, this section focuses on working only with the custom application module interface; however; the same downcasting approach works on the client to use a `ViewObject` interface as a view object interface like `ServiceRequests` or a `Row` interface as a custom view row interface like `ServiceRequestsRow`.

Example 8–9 illustrates a `TestClientCustomInterface` class that puts these two steps into practice. You might recognize it from Section 6.8, "Working Programmatically with Entity Objects and Associations" where you tested some example application module methods from within the `main()` method of the `SRServiceImpl` class itself. Here it's calling all of the same methods from the client using the `SRService` client interface.

The basic logic of the example follows these steps:

1. Acquire the application module instance and cast it to the more specific `SRService` client interface.

> **Note:** If you work with your application module using the default `ApplicationModule` interface in the `oracle.jbo` package, you won't have access to your custom methods. Make sure to cast the application module instance to your more specific custom interface like the `SRService` interface in this example.

2. Call `findRequestStatus()` to find the status of service request `101`.
3. Call `findServiceRequestTechnician()` to find the name of the technician assigned to service request 101.
4. Call `updateRequestStatus()` to try updating the status of request 101 to the illegal value `Reopened`.
5. Call `createProduct()` to try creating a product with a missing product name attribute value, and display the new product ID assigned to it.

Example 8–9 Using the Custom Interface of an Application Module from the Client

```
package devguide.client;
import devguide.model.common.SRService;
import oracle.jbo.JboException;
import oracle.jbo.client.Configuration;
public class TestClientCustomInterface {
  public static void main(String[] args) {
    String        amDef = "devguide.model.SRService";
    String        config = "SRServiceLocal";
    /*
     * This is the correct way to use application custom methods
     * from the client, by using the application module's automatically-
     * maintained custom service interface.
     */
    // 1. Acquire instance of application module, cast to client interface
    SRService service =
      (SRService)Configuration.createRootApplicationModule(amDef,config);
    // 2. Find the status of service request 101
    String status = service.findServiceRequestStatus(101);
    System.out.println("Status of SR# 101 = " + status);
    // 3. Find the name of the technician assigned to service request 101
    String techName = service.findServiceRequestTechnician(101);
    System.out.println("Technician for SR# 101 = " + techName);
     try {
       // 4. Try updating the status of service request 101 to an illegal value
       service.updateRequestStatus(101,"Reopened");
     }
     catch (JboException ex) {
       System.out.println("ERROR: "+ex.getMessage());
     }
     long id = 0;
     try {
       // 5. Try creating a new product with a missing required attribute
       id = service.createProduct(null,"Makes Blended Fruit Drinks");
     }
     catch (JboException ex) {
```

```
      System.out.println("ERROR: "+ex.getMessage());
    }
    // 6. Try creating a new product with a missing required attribute
    id = service.createProduct("Smoothie Maker","Makes Blended Fruit Drinks");
    System.out.println("New product created successfully with id = "+id);
   Configuration.releaseRootApplicationModule(service,true);
  }
}
```

8.5.2 What Happens When You Work with an Application Module's Client Interface

If the client layer code using your application module is located in the same tier of the J2EE architecture, this configuration is known as using your application module in "local mode." In local mode, the client interface is implemented directly by your custom application module Java class. Typical situations that use an application module in local mode are:

- A JavaServer Faces application, accessing the application module in the web tier
- A JSP/Struts application, accessing the application module in the web tier
- A Swing application, accessing the application module in the client tier (2-tier client/server style)

In contrast, when the client layer code accessing your application module is located in a *different* tier of the J2EE architecture, this is known as using the application module in "remote mode." In remote mode, the generated client proxy class described above implements your application module service interface on the client side and it handles all of the communications details of working with the remotely-deployed application module service. The typical situation that uses "remote mode" is a thin-client Swing application accessing the application module on a remote application server.

A unique feature of ADF Business Components is that by adhering to the best-practice interface-only approach for working with client service methods, you can be sure your client code works unchanged regardless of your chosen deployment mode. Even if you only plan to work in "local mode", it is still the best practice in the J2EE development community to adopt an interface-based approach to working with your services. Using application modules, it's extremely easy to follow this best practice in your applications.

> **Note:** Whether you plan to use your application modules in local deployment mode or remote mode, as described in Section 8.4.4, "What You May Need to Know About Method Signatures on the Client Interface", the JDeveloper design time enforces that your custom interface methods use Serializable types. This allows you to switch at any time between local or remote deployment mode, or to support both at the same time, with no code changes.

8.5.3 How to Access an Application Module Client Interface

The `Configuration` class in the `oracle.jbo.client` package makes it very easy to get an instance of an application module for *testing*. You've seen it used in numerous test client programs in this guide, and you'll see it again in the chapter on testing your application module services as part of the JUnit regression testing fixture. Because it is easy, however, it is tempting for developers to use its `createRootApplicationModule()` and `releaseApplicationModule()` methods *everywhere* they want to access an application module.

However, for web applications you should resist this temptation because there is an even easier way.

8.5.3.1 How to Access an Application Module Client Interface in a JSF Web Application

When working with JSF or Struts/JSP applications using the ADF Model layer for data binding, JDeveloper configures a servlet filter in your `ViewController` project called the `ADFBindingFilter`. It orchestrates the automatic acquisition and release of an appropriate application module instance based on declarative binding metadata, and insures that the service is available to be looked up as a data control. You'll learn more about the ADF `BindingContext` and data controls in later chapters, however here it's enough to remember that you can access the application module's client interface from this `BindingContext`. Since the `BindingContext` is available during each web page request by referencing the request-scoped attribute named `data`, you can reference the binding context in a JSF managed bean.

For instance, if you want to access the custom interface of the `devguide.model.SRService` application module, follow these basic steps shown in Example 8–10:

1. Access the JSF `FacesContext`.
2. Create value binding for the `#{data}` EL expression.
3. Evaluate the value binding, casting the result to `BindingContext`.
4. Find the data control by name from the `BindingContext` .
5. Access the application module data provider from the data control.
6. Cast the application module to its client interface.
7. Call any method on the client interface.

Example 8–10 Accessing the Application Module Client Interface in a JSF Backing Bean

```
package demo.view;
import devguide.model.common.SRService;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
import oracle.adf.model.BindingContext;
import oracle.adf.model.binding.DCDataControl;
import oracle.jbo.ApplicationModule;
public class YourBackingBean {
  public String commandButton_action() {
    // 1. Access the FacesContext
    FacesContext fc = FacesContext.getCurrentInstance();
    // 2. Create value binding for the #{data} EL expression
    ValueBinding vb = fc.getApplication().createValueBinding("#{data}");
    // 3. Evaluate the value binding, casting the result to BindingContext
    BindingContext bc = (BindingContext)vb.getValue(fc);
    // 4. Find the data control by name from the binding context
    DCDataControl dc  = bc.findDataControl("SRServiceDataControl");
    // 5. Access the application module data provider
    ApplicationModule am = (ApplicationModule)dc.getDataProvider();
    // 6. Cast the ApplicationModule to its client interface
    SRService service    = (SRService)am;
    // 7. Call a method on the client interface
    service.doSomethingInteresting();
    return "SomeNavigationRule";
  }
}
```

The SRDemo application includes a `JSFUtils` class that encapsulates steps of evaluating an EL expression using a JSF value binding, and you can use the dot notation in an EL expression to chain method invocations together on successive beans and to look up beans in maps. So, putting these two ideas together, you can reduce the above steps to the single line like:

```
// Access the SRService custom interface with a single EL expression
SomeService service = (SomeService)JSFUtils.resolveExpression("#{data.SRServiceDat
aControl.dataProvider}");
```

Note: ISection 10.3.2, "How to Change the Data Control Name Before You Begin Building Pages" explains how to rename the data control for an application module. If, as done in the SRDemo application, you use the technique described there to rename the data control for the `SRService` from the default name `SRServiceDataControl` to the shorter `SRService` name that matches the name of the application module itself, then the line of code above becomes the following:

```
// Access SRService custom interface with a single EL expression
// NOTE: SRService app module data control renamed to "SRService"
SomeService service = (SomeService)JSFUtils.resolveExpression(
                                 "#{data.SRService.dataProvider}");
```

8.5.3.2 How to Access an Application Module Client Interface in a JSP/Struts Web Application

If you use Struts and JSP for your view and controller layers, you can access the `BindingContext` and your application module custom interface from your custom `PageController` using code like what you see in Example 8–11. Notice that you can directly cast the result of `getDataProvider()` to the application module client interface without first needing to retrieve it as `ApplicationModule`.

Example 8–11 Accessing the Application Module Client Interface in ADF Page Controller

```
package demo.view;
import devguide.model.common.SRService;
import oracle.adf.controller.v2.context.LifecycleContext;
import oracle.adf.controller.v2.lifecycle.PageController;
import oracle.adf.model.BindingContext;
import oracle.adf.model.binding.DCDataControl;
public class YourPageController extends PageController {
  public void prepareModel(LifecycleContext lcContext) {
    super.prepareModel(lcContext);
    BindingContext bc = lcContext.getBindingContext();
    DCDataControl  dc = bc.findDataControl("SRServiceDataControl");
    SRService service = (SRService)dc.getDataProvider();
    service.doSomethingInteresting();
  }
}
```

8.5.3.3 How to Access an Application Module Client Interface in an ADF Swing Application

If you use Swing to create desktop-fidelity applications, you can access the `BindingContext` and your application module custom interface from inside your Swing panels using code like what you see in Example 8–12.

Example 8–12 Accessing the Application Module Client Interface in ADF Swing Panel

```
package demo.view.panels;
import devguide.model.common.SRService;
import oracle.adf.model.BindingContext;
import oracle.adf.model.binding.DCDataControl;
// etc.
public class YourPanel extends JPanel implements JUPanel {
  // etc.
  private void jButton1_actionPerformed(ActionEvent e) {
    BindingContext bc = getPanelBinding().getBindingContext();
    DCDataControl dc = bc.findDataControl("SRServiceDataControl");
    SRService service = (SRService)dc.getDataProvider();
    service.doSomethingInteresting();
  }
}
```

8.6 Overriding Built-in Framework Methods

The `ApplicationModuleImpl` base class provides a number of built-in methods that implement its functionality. While Appendix D, "Most Commonly Used ADF Business Components Methods" provides a quick reference to the most common code that you will typically write, use, and override in your custom application module classes, this section focuses on helping you understand the basic steps to override one of these built-in framework methods to augment the default behavior.

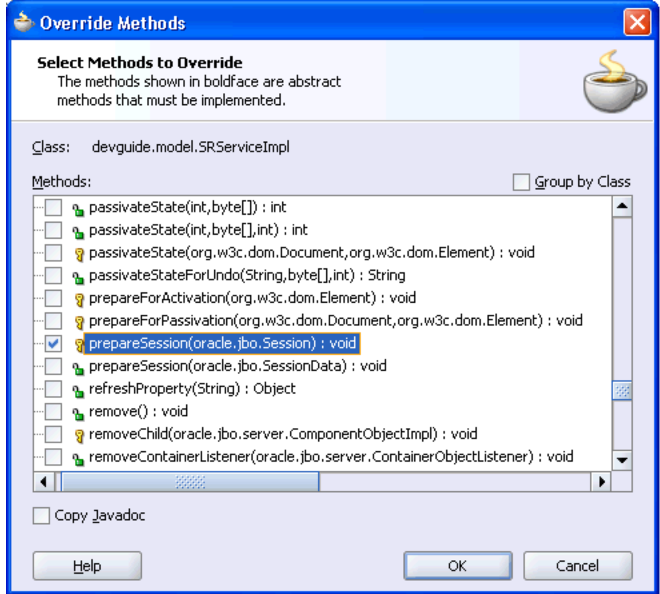
8.6.1 How to Override a Built-in Framework Method

To override a built-in framework method for an application module, go to the application module Java class and then choose **Source | Override Methods...** from the JDeveloper main menu. As shown in Figure 8–9, the **Override Methods** dialog appears. You can scroll the list of methods using the scrollbar, but if you know the method you want to override, begin typing the first few letters of its name to perform an incremental search to find it more quickly in the list.

Assume that you want to override the application module's `prepareSession()` method to augment the default functionality when a new user session begins working with an application module service component for the first time. Check the checkbox next to the `prepareSession(oracle.jbo.Session)` method, and click **OK**.

Note: Here you've only selected one method, but the **Override Methods** dialog allows you to select any number of methods to override simultaneously.

Figure 8–9 Overriding a Built-in Framework Method



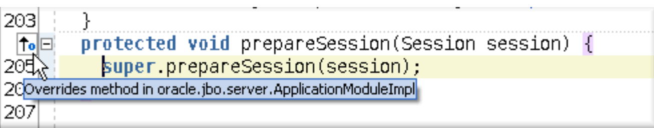
8.6.2 What Happens When You Override a Built-in Framework Method

When you dismiss the **Override Methods** dialog, you return to the code editor with the cursor focus in the overridden method, as shown in Figure 8–10. Notice that the method appears with a single line that calls `super.prepareSession()`. This is the syntax in Java for invoking the default behavior the base class would have normally performed for this method. By adding code before or after this line in the custom application module class, you can augment the default behavior before or after the default functionality.

Also notice that when you override a method in a base class, the code editor's left margin displays a small "up arrow" icon. This is a clue that this method is related to an interface or a base class. Hovering the mouse over the icon as shown in the figure gives positive feedback that this method has the correct signature to override a method in the base class.

While this method was generated by the **Override Methods** dialog, if you begin typing overridden method names from memory, it's important to have the signature *exactly* the same as the base class method you want to override. If you make a spelling mistake in what you think is an overridden method, or if you have the wrong number or types of arguments or return value, then you won't get any compile errors but neither will you be overriding the base class method you anticipated. The positive feedback makes it clear you got the method name exactly right. To quickly navigate to the base class to double-check what it's doing when you call the superclass, choose **Go to Overridden Method** in the right-mouse context menu menu on the override icon in the margin.

Figure 8–10 Code Editor Margin Gives Visual Feedback About Overridden Methods



Note: In addition to the *design time* visual method override confirmation described above, you can consider adding the JDK 5.0 `@Override` annotation just before any method in your class whose intent it is to override its superclass. This causes the compiler to generate a compile-time error if it does not match the signature of any method in the superclass.

8.6.3 How to Override prepareSession() to Set Up an Application Module for a New User Session

Since the `prepareSession()` method gets invoked by the application module when it is used for the first time by a new user session, it's a nice method to override in your custom application module class to perform setup tasks that are specific to each new user that uses your application module. Example 8–13 illustrates an overridden `prepareSession()` method in the `devguide.model.SRServiceImpl` class that invokes a `findLoggedInUserByEmailInStaffList()` helper method to initialize the `StaffList` view object instance to display the row corresponding to the currently logged-in user.

The helper method does the following:

1. Calls `super.prepareSession()` to perform the default processing
2. Accesses the `StaffList` view object instance using the generated `getStaffList()` getter method
3. Calls the `getUserPrincipalName()` method to get name of the currently authenticated user

---

Note: Until you learn about enabling J2EE security in your application in Section 30.4, "Configuring the ADF Business Components Application to Use Container-Managed Security", the `getUserPrincipalName()` API in the sample below will return `null` instead of the authenticated user's name. So, the example contains the fallback code that assigns a fixed email ID of `sking` for testing purposes.

---

4. Defines a `CurrentUser` named bind variable, with the `currentUserName` member variable as its default value
5. Sets an additional WHERE clause to find the current user's row by email
6. Executes the query to retrieve the `StaffList` row for the current user

After overriding the `prepareSession()` method in this way, if you test the `SRService` application module using the Business Components Browser, you'll see that the `StaffList` view object instance has the single row corresponding to Steven King (`email = 'sking'`).

Example 8–13 Initializing the StaffList View Object Instance to Display a Current User's Information

```
// In devguide.model.SRServiceImpl class
protected void prepareSession(Session session) {
  // 1. Call the superclass to perform the default processing
  super.prepareSession(session);
  findLoggedInUserByEmailInStaffList();
}
private void findLoggedInUserByEmailInStaffList() {
  // 2. Access the StaffList vo instance using the generated getter method
  ViewObject staffList = getStaffList();
  // 3. Get the name of the currently authenticated user
  String currentUserName = getUserPrincipalName();
  /*
   * Until later when we learn how to integrate J2EE security,
   * this API will return null. For testing, we can default it
   * to the email of one of the staff members like "sking".
   */
  if (currentUserName == null) {
    currentUserName = "sking";
  }
 /*
  * We can't use a simple findByKey since the key for the
  * StaffList view object is the numerical userid of the staff member
  * and we want to find the user by their email address. We could build
  * an "EMAIL = :CurrentUser" where clause directly into the view object
  * at design time, but here let's illustrate doing it dynamically to
  * see this alternative.
  */
  // 4. Define named bind variable, with currentUserName as default value
 staffList.defineNamedWhereClauseParam("CurrentUser",    // bindvar name
                                       currentUserName, // default value
                                       null);
 // 5. Set an additional WHERE clause to find the current user's row by email
 staffList.setWhereClause("EMAIL = :CurrentUser");
 // 6. Execute the query to retrieve the StaffList row for the current user
 staffList.executeQuery();
 /*
  * If the view object needs to be also used during this session
  * without the additional where clause, you would use
  * setWhereClause(null) and removeNamedWhereClauseParam("CurrentUser") to
  * leave the view object instance back in it's original state.
  */
}
```

8.7 Creating an Application Module Diagram for Your Business Service

Since your business service's data model and service interface are a key asset to your team, it is often convenient to visualize it using a UML model. JDeveloper supports easily creating a diagram for your application module that you and your colleagues can use for reference.

8.7.1 How to Create an Application Module Diagram

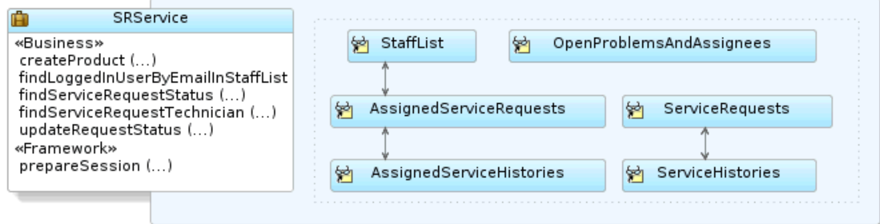
To create a diagram of your application module:

1. Open the **Create Business Components Diagram** dialog from the **New Gallery** in the **Business Tier > ADF Business Components** category.
2. The dialog prompts you for a diagram name, and a package name in which the diagram will be created. Enter a diagram name like "`SRService Data Model`" and a package name like `devguide.model.design`.

   clicking **OK** creates the empty diagram and opens the diagrammer.
3. To add your existing application module to the diagram, select them all in the **Application Navigator** and drop them onto the diagram surface.
4. Use the property inspector to:
   - Hide the package name,
   - Change the font
   - Turn off the grid and page breaks
   - Turn off the display of the role names on the view links ("Master"/"Detail")

After completing these steps, the diagram looks like what you see in: Figure 8–11

Figure 8–11 UML Diagram of Application Module in a Drag and Drop and a Few Clicks



8.7.2 What Happens When You Create an Application Module Diagram

When you create a business components diagram, JDeveloper creates an XML file representing the diagram in a subdirectory of the project's model path that matches the package name in which the diagram resides. For the `Business Domain Objects` diagram above, it would create a matching `*.oxd_bc4j` file in the `./devguide/model/design` subdirectory of the model path. By default the **Application Navigator** unifies the display of the project contents paths so that ADF components and Java files in the source path appear in the same package tree as the UML model artefacts in the project model path. You can use the **Toggle Directories** toolbar icon in the Application Navigator to switch between the unified directory view and seeing the distinct project content path folders.

8.7.3 What You May Need to Know About Application Module Diagrams

You can do a number of tasks directly on the diagram, such as editing the application module, controlling display options, filtering methods names, showing related objects and files, publishing the application, and launching the Business Components Browser.

8.7.3.1 Using the Diagram for Editing the Application Module

The UML diagram of business components is not just a static picture that reflects the point in time when you dropped the application module onto the diagram. Rather, it is a UML-based rendering of the current component definitions so it will always reflect the current state of affairs. The UML diagram is both a visualization aid and a visual navigation and editing tool. You can bring up the Application Module Editor for any application module in a diagram by selecting **Properties...** from the right-mouse context menu (or double-clicking). You can also perform some application module editing tasks directly on the diagram like renaming view object instances, dropping view object definitions onto the data model to create a new view object instance, and removing view object instances by pressing the Delete key.

8.7.3.2 Controlling Display Options

After selecting the application module in the diagram, use the Property Inspector to control its display options. In the **Display** category, toggle properties like the following:

- **Show Stereotype —** to display the type of object (e.g. "<<application module>>")
- **Show Operations —** to display service methods
- **Show Package —** to display the package name

Note: The term *operation* is a more generic, UML name for methods.

In the **Operations** category, you will typically consider changing the following properties depending on the amount of detail you want to provide in the diagram:

- **Show Method Parameters**
- **Show Return Types**
- **Show Visibility** (public, private, etc.)

On the right-mouse context menu, you can also select to **View As**:

- **Standard —** to show service operations
- **Expanded —** to show operations and data model (*default*)
- **Compact —** to show only the icon and the name

8.7.3.3 Filtering Method Names

Initially, if you show the operations for the application module the diagram displays all the methods. Any method it recognizes as overridden framework methods display in the **<<Framework>>** operations category. The rest display in the **<<Business>>** methods category.

The **Exclude Operations Filter** property is a regular expression that you can use to filter out methods you don't want to display on the diagram. For example, by setting the **Exclude Operations Filter** property to

```
findLoggedInUser.*|retrieveService.*|get.*
```

you can filter out all of the following application module methods:

- `findLoggedInUserByEmailInStaff`
- `retrieveServiceRequestById`
- All the generated view object getter methods

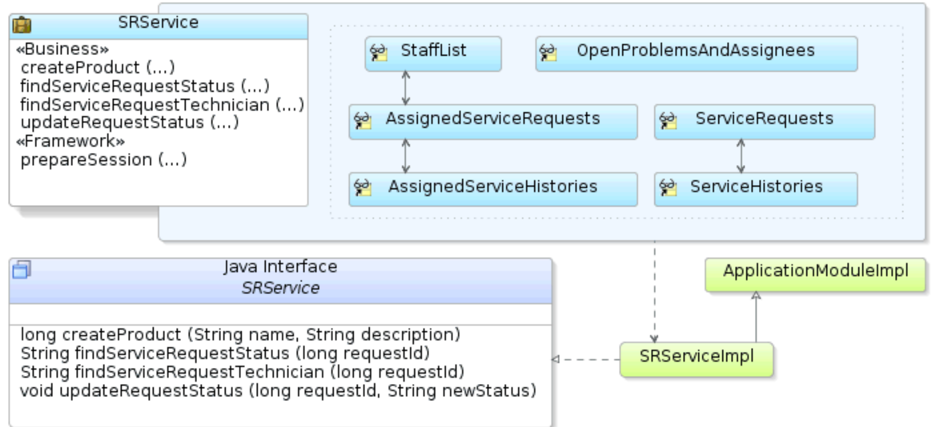
8.7.3.4 Show Related Objects and Implementation Files

After selecting the application module on the diagram — or any set of individual view object instances in its data model — you can choose **Show > Related Elements** from the right-mouse context menu to display related component definitions on the diagram. In a similar fashion, selecting **Show > Implementation Files** includes the files that implement the application module on the diagram. You can repeat these options on the additional diagram elements that appear until the diagram includes the level of detail you want to convey.

Note: Deleting components from the diagram only removes their visual representation on the diagram surface. The components and classes remain on the file system and in the Application Navigator.

Figure 8–12 illustrates what you'll see if you include the implementation files for the `devguide.model.SRService` application module, showing the related elements for the `SRServiceImpl` class, and drawing an additional dependency line between the `SRService` application module and the `SRServiceImpl` class. Note the generated `SRService` client interface that you used above.

Figure 8–12 Adding Detail to a Diagram Using Show Related Elements and Show Implementation Files



8.7.3.5 Publishing the Application Module Diagram

To publish the diagram to PNG, JPG, SVG, or compressed SVG format, choose **Publish Diagram...** from the right-mouse context menu on the diagram surface.

8.7.3.6 Testing the Application Module From the Diagram

To launch the Business Components Browser for an application module in the diagram, choose **Test...** from the right-mouse context menu.

8.8 Supporting Multipage Units of Work

During the span of time your end user is interacting with your application, she might:

- Visit the same pages multiple times, expecting fast response times
- Perform a logical unit of work that requires visiting many different pages to complete
- Need to perform a partial "rollback" of a pending set of changes they've made but haven't saved yet.
- Unwittingly be the victim of an application server failure in a server farm before saving pending changes

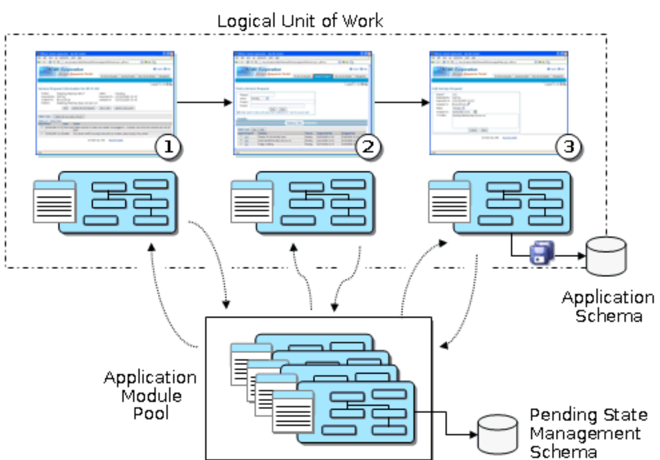
This section provides a brief overview of the application module pooling and state management features that simplify implementing scalable, well-performing applications that address these requirements.

8.8.1 Overview of Application Module Pooling and State Management

Applications you build that leverage an application module as their business service take advantage of an automatic application module pooling feature. This facility manages a configurable set of application module instances that grows and shrinks as the end user load on your application changes during the day. Due to the natural "think time" inherent in the end user's interaction with your application user interface, the number of application module instances in the pool can be smaller than the overall number of active users using the system.

As shown in Figure 8–13, as a given end user visits multiple pages in your application to accomplish her logical task, on each page request an application module instance in the pool is acquired automatically from the pool for the lifetime of that one request. At the end of the request, the instance is automatically returned to the pool for use by another user session. In order to protect the end user's work against application server failure, the application module supports the ability to "dehydrate" the set of pending changes in its entity caches to a persistent store by saving an XML "snapshot" describing the change set. For scalability reasons, this state snapshot is typically saved in a state management schema that is a different database schema than the one containing the application data.

Figure 8–13 Using Pooled Application Modules Throughout a Multipage, Logical Unit of Work



The pooling algorithm affords a tunable optimization whereby a certain number of application module instances will attempt to stay "sticky" to the last user session that returned them to the pool. The optimization is not an iron-clad guarantee, but when a user can benefit by the optimization they continue to work with the same application module instance from the pool as long as system load allows. When load is too high, the pooling algorithm uses any available instance in the pool to service the user's request and the dehydrated "snapshot" of their logical unit of work is rehydrated from the persistent store to allow the new instance of the application module to continue where the last one left off. The end user continues to work in this way until they commit or rollback their changes.

Using these facilities, the application module delivers the productivity of a stateful development paradigm that can easily handle multi-page work flows, in an architecture that delivers the runtime performance near that of a completely stateless application. You will learn more about these application module features in Chapter 28, "Application Module State Management" and about how to tune them in Chapter 29, "Understanding Application Module Pooling".

Note: This application module pooling and state management is also available for thin-client, desktop-fidelity Swing applications and web-style user interfaces.

8.8.2 Experimenting with State Management in the Business Components Browser

For a quick taste of what the state management functionality does, you can launch the Business Components Browser on the `devguide.model.SRService` application module and try the following steps:

1. Double-click on the `OpenProblemsAndAssignees` view object instance to query its data.
2. Make a note of the current values of the **Status** and the **Assigned To** attributes for a few rows.
3. Update those rows to have a different value for **Status** and **Assigned To**, but do not commit the changes.

4. Choose **File | Save Transaction State** from the Business Components Browser main menu.

   A **Passivated Transaction State** dialog appears, indicating a numerical transaction ID number. Make a note of this number.

   Exit the Business Components Browser and then restart it.

5. Exit out of the Business Components Browser completely.

6. Restart the Business Components Browser and double-click on the `OpenProblemsAndAssignees` view object instance to query its data.

7. Notice that the data is *not* changed. The queried data from the data reflects the current state of the database without your changes.

8. Choose **File | Restore Transaction State** from the Business Components Browser main menu, and enter the transaction ID you noted in step 4.

At this point you'll see that your pending change set is reflected again in the rows you modified. If you commit the transaction now, your changes are permanently saved to the database.

8.9 Deciding on the Granularity of Application Modules

A common question from developers related to application modules is, "How big should my application module be?" In other words, "Should I build one big application module to contain the entire data model for my enterprise application, or many smaller application modules?" The answer is different for different situations. This section provides some tips about how to answer this question for your own application.

In general, application modules should be as big as necessary to support the specific use case you have in mind for them to accomplish. They can be assembled from finer-grained application module components using a "nesting" feature described below. Since a complex business application is not really a single use case, a complex business application implemented using ADF will typically not be just a single application module.

In the case of a sample like the SRDemo application, there is really only one main use case that it is implementing which allows users to manage service requests. You could argue that the application's functionality of managing technicians skills is a separate "back-end" use case, and you'd be right. However, in practice, modeling the demo that way implied having a second application module with just two view object instances in it, so the SRDemo developers took the liberty of including everything for this small-sized sample application into a single application module for simplicity's sake.

8.9.1 Use Cases Assist in Planning Your Application Modules

In the early analysis phases of application development, often architects and designers use UML use case techniques to iteratively refine a high-level description of the different kinds of end-user functionalities that the system being built will need to support.

Each high-level, end-user use case identified during this design phase typically identifies two kinds of logical outputs:

- The domain business objects involved

  What core business data is relevant to the use case?

- The user-oriented view of business data required

  What subset of columns, what filtered set of rows, sorted in what way, grouped in what way, is needed to support the use case?

The identified domain objects involved in the use case help you know which entity objects from your business domain layer will participate in the use case. The user-oriented view of business data required helps developers define the right SQL queries captured as view objects to retrieve the data for the end user in the exactly way that they expect. For best performance, this includes retrieving the minimum required details necessary to support the use case as well. In addition to leveraging view object queries to shape the data, you've learned how to use view links to setup natural master/detail hierarchies in your data model to match exactly the kind of end-user experience you want to offer the user to accomplish the use case.

The application module is the "workunit" container that includes instances of the reusable view objects required for the use case in question, related through metadata to the underlying entity objects in your reusable business domain layer whose information the use case is presenting or modifying.

8.9.2 Application Modules Are Designed to Support Assembly

Use cases map to modular functions. Certain higher-level functions might "reuse" a "subfunction" that is common to several business work flows. Application modules support the ability to create software components that mimic this natural modularity by creating composite application modules that you assemble using instances of other application modules. When you nest an instance of an application module inside another, you aggregate not only the view objects in its data model, but also any custom service methods it defines. While this feature of "nesting" or reusing an instance of one application module inside of another is not typically the focus of simple sample like the SRDemo application, it is one of the most powerful design aspects of the ADF Business Components layer of Oracle ADF for implementing larger-scale, real-world application systems.

Using the basic logic that an application module represents an end-user use case or work flow, you can build application modules that cater to the data required by some shared, modular use case, and then reuse those application modules inside of other more complicated application modules that are designed to support a more complex use case. For example, imagine that after creating application modules `SRService` and `ProductMaintenaceService`, you later need to build an application that uses both of these services as an integral part of a new `CompositeService`. Figure 8–14 illustrates what this `CompositeService` would look like in a JDeveloper business components diagram. Notice that an application module like `CompositeService` can contain a combination of view object instances and application module instances.

Figure 8–14 Application Module Instances Can Be Reused to Assemble Composite Services

If you leverage nested application modules in your application, make sure to read Section 10.3.7, "How Nested Application Modules Appear in the Data Control Palette" as well to avoid common pitfalls when performing data binding involving them.

8.9.3 Root Application Modules Versus Nested Application Module Usages

At runtime, your application works with a *main* — or what's known as a *root* — application module. Any application module can be used as a root application module, however in practice the application modules that are used as root application modules are the ones that map to a more complex end-user use case, assuming you're not just building a straightforward CRUD application. When a root application module contains other nested application modules, they all participate in the root application module's transaction and share the same database connection and a single set of entity caches. This sharing is handled for you automatically by the root application module and its `Transaction` object.

At runtime, each top-level application module that your application uses will get an application module pool created for it. More sophisticated customers like Oracle Applications who are building large applications with many, many business functions, often have written custom code to make more efficient use of a "generic" pool of container application modules to avoid ending up with hundreds of different application module pools. Based on the customer workflow that needs to be started, they have written code that takes a generic application module — which is basically an application module with no view object instances of its own or any nested application modules of its own defined at design time — and they programmatically use the `createApplicationModule()` API on this "generic" application module to dynamically nest an instance of the appropriate "use case"-based application module at runtime. When the user is done, they call `remove()` on that dynamically nested application module instance, leaving the generic application module "empty" again, ready to be used by another end user for whatever "use case" they need to accomplish.

9 Implementing Programmatic Business Rules in Entity Objects

This chapter explains the key entity object events and features for implementing the most common kinds of business rules.

This chapter includes the following sections:

- Section 9.1, "Introduction to Programmatic Business Rules"
- Section 9.2, "Understanding the Validation Cycle"
- Section 9.3, "Using Method Validators"
- Section 9.4, "Assigning Programmatically-Derived Attribute Values"
- Section 9.5, "Undoing Pending Changes to an Entity Using the Refresh Method"
- Section 9.6, "Using View Objects for Validation"
- Section 9.7, "How to Access Related Entity Rows Using Association Accessors"
- Section 9.8, "How to Reference Information About the Authenticated User"
- Section 9.9, "How to Access Original Attribute Values"
- Section 9.10, "How to Store Information About the Current User Session"
- Section 9.11, "How to Access the Current Date and Time"
- Section 9.12, "How to Send Notifications Upon a Successful Commit"
- Section 9.13, "How to Conditionally Prevent an Entity Row from Being Removed"
- Section 9.14, "How to Implement Conditional Updatability for Attributes"
- Section 9.15, "Additional Resources"

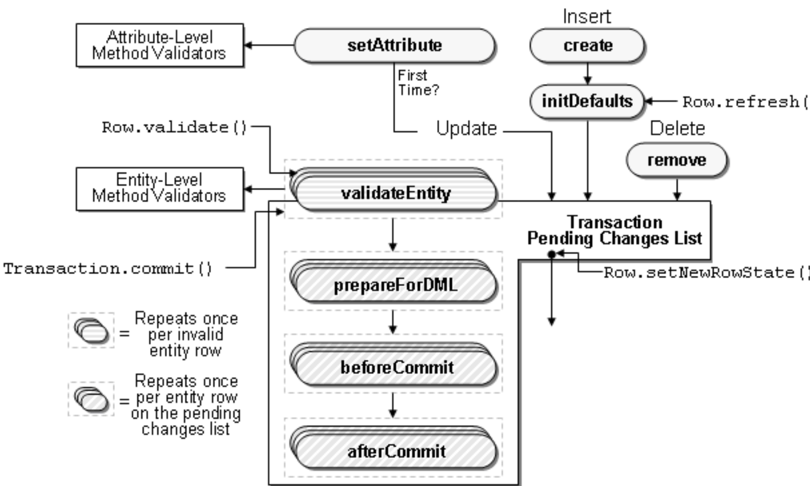
9.1 Introduction to Programmatic Business Rules

Complementing its built-in *declarative* validation features, entity objects have method validators and several events you can handle to implement encapsulated business logic using Java code. By the end of this chapter, you'll understand all the concepts illustrated in Figure 9–1, and more:

- Attribute-level method validators trigger validation code when an attribute value changes.
- Entity-level method validators trigger validation code when an entity row is validated.

- You can override the following key methods in a custom Java class for an entity:
  - `create()`, to assign default values when a row is created
  - `initDefaults()`, to assign defaults either when a row is created or when a new row is refreshed
  - `isAttributeUpdateable()`, to make attributes conditionally updatable
  - `remove()`, to conditionally disallow deleting
  - `prepareForDML()`, to assign attribute values before an entity row is saved
  - `beforeCommit()`, to enforce rules that must consider all entity rows of a given type
  - `afterCommit()`, to send notifications about a change to an entity object's state

Figure 9–1 Key Entity Objects Features and Events for Programmatic Business Logic



9.2 Understanding the Validation Cycle

Each entity row tracks whether or not its data is valid. When an existing entity row is retrieved from the database, the entity is assumed to be valid. When the first persistent attribute of an existing entity row is modified, or when a new entity row is created, the entity is marked invalid.

In addition, since a composed child entity row is considered an integral part of its composing parent entity object, any change to composed child entity rows causes the parent entity to be marked invalid.

When an entity is in an invalid state, the declarative validation you have configured and the programmatic validation rules you have implemented are evaluated again before the entity can be consider valid again. You can determine whether a given entity row is valid at runtime by calling the `isValid()` method on it.

9.2.1 Types of Entity Object Validation Rules

Entity object validation rules fall into two basic categories: attribute-level and entity-level.

9.2.1.1 Attribute-Level Validation Rules

Attribute-level validations are triggered for a particular entity object attribute when either the end user or program code attempts to modify the attribute's value. Since you cannot determine the order in which attributes will be set, attribute-level validation rules should be only used when the success or failure of the rule depends exclusively on the candidate value of that single attribute.

The following examples are attribute-level validations:

- The value of the `AssignedDate` of a service request should not be a date in the past.
- The `ProdId` attribute of a service request should represent an existing product.

9.2.1.2 Entity-Level Validation Rules

All other kinds of validation rules are entity-level validation rules. These are rules whose implementation requires considering two or more entity attributes, or possibly composed children entity rows, in order to determine the success or failure of the rule.

The following examples are attribute-level validations.

- The value of the `AssignedDate` of a service request should be a data that comes after the `RequestDate`.
- The `ProdId` attribute of a service request should represent an existing product.

Entity-level validation rules are triggered by calling the `validate()` method on a `Row`. This will occur when:

- You call the method explicitly on the entity object
- You call the method explicitly on a view row with an entity row part that is invalid
- A view object's iterator calls the method on the current row in the view object before allowing the current row to change
- Transaction commit processing validates an invalid entity in the pending changes list before proceeding with posting the changes to the database.

9.2.2 Understanding Commit Processing and Validation

Transaction commit processing happens in three basic phases:

1. Ensure any invalid entity rows on the pending changes list are valid.
2. Post the pending changes to the database by performing appropriate DML operations.
3. Commit the transaction.

If you have business validation logic in your entity objects that executes queries or stored procedures that depends on seeing the posted changes in the `SELECT` statements they execute, they should be coded in the `beforeCommit()` method described in Section 9.6.3, "Validating Conditions Related to All Entities of a Given Type". This method fires after all DML has been applied so queries or stored procedures invoked from that method can "see" all of the pending changes that have been saved, but not yet committed.

Caution: The transaction-level `postChanges()` method that exists to force the transaction to post unvalidated changes without committing them is not recommended for use in web applications unless you can guarantee that the transaction will definitely be committed or rolled-back during the same HTTP request. Failure to heed this advice can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

9.2.3 Avoiding Infinite Validation Cycles

If your validation rules contain code that updates attributes of the current entity or other entities, then the act of validating the entity can cause that or other entities to become invalid. As part of the transaction commit processing phase that attempts to validate all invalid entities in the pending changes list, the transaction will perform up to 10 passes on the pending changes list in an attempt to reach a state where all pending entity rows are valid.

If after 10 passes, there are still invalid entities in the list, you will see the following exception:

```
JBO-28200: Validation threshold limit reached. Invalid Entities still in cache
```

This is as sign that you need to debug your validation rule code to avoid inadvertently invalidating entities in a cyclic fashion.

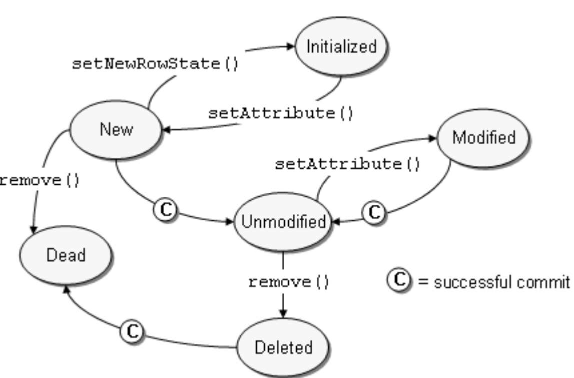
9.2.4 What Happens When Validations Fail

When an entity object's validation rules throw exceptions, the exceptions are bundled and returned to the client. If the validation failures are thrown by methods you've overridden to handle events during the transaction `postChanges` processing, then the validation failures cause the transaction to rollback any database `INSERT`, `UPDATE`, or `DELETE` statements that might have been performed already during the current `postChanges` cycle.

9.2.5 Understanding Entity Objects Row States

When an entity row is in memory, it has an entity state that reflects the logical state of the row. Figure 9–2 illustrates the different entity row states and how an entity row can transition from one state to another. When a entity row is first created, its status is `New`. You can use the `setNewRowState()` method to mark the entity as being `Initialized`, which removes it from the transaction's list of pending changes until the user sets at least one of its attributes, at which time it returns to the `New` state. The `Unmodified` state reflects an entity that has been retrieved from the database and has not yet been modified. It is also the state that a `New` or `Modified` entity transitions too after the transaction successfully commits. During the transaction in which it is pending to be deleted, an `Unmodified` entity row transitions to the `Deleted` state. Finally, if a row that was `New` and got removed before the transaction commits, or `Unmodified` and got successfully deleted, the row transition to the `Dead` state.

Figure 9–2 Diagram of Entity Row States and Transitions



You can use the `getEntityState()` method to access the current state of an entity row in your business logic code.

> **Note:** If you use the `postChanges()` method to post pending changes without committing the transaction yet, the `getPostState()` method returns the entity's state from the point of view of it's being posted to the database or not. For example, a new entity row that has been inserted into the database due to your calling the `postChanges()` method programmatically — but which has not yet been committed — will have a different value for `getPostState()` and `getEntityState()`. The `getPostState()` method will reflect an "Unmodified" status since the new row has been posted, however the `getEntityState()` will still reflect that the entity is `New` in the current transaction.

9.3 Using Method Validators

Method validators are the primary way Oracle recommends supplementing declarative validation rules using your own Java code. Method validators trigger Java code that you write in your own validation methods at the appropriate time during the entity object validation cycle. You can add any number of attribute-level method validators or entity-level method validators, provided they each trigger a distinct method name in your code. All validation method names must begin with the word `validate`; however, following that rule you are free to name them in any way that most clearly identifies there functionality.

9.3.1 How to Create an Attribute-Level Method Validation

To create an attribute-level method validator:

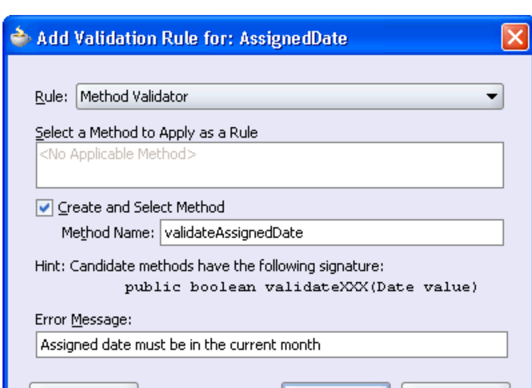
1. Open the Entity Object Editor
2. If your entity object does not yet have a custom Java class, then first open the **Java** page and enable the generation of an **Entity Object Class**, and click **Apply** to generate the `*.java` file.
3. Open the **Validation** page and select the attribute that you want to validate.
4. Click **New** to add a validation rule.
5. Select the **Method Validator** type from the **Rule** dropdown list, as shown in Figure 9–3.

The **Add Validation Rule** dialog displays the expected method signature for an attribute-level validation method. You have two choices:

- If you already have a method in your entity object's custom Java class of the appropriate signature, it will appear in the list and you can select it after unchecking the **Create and Select Method** checkbox.
- If you leave the **Create and Select Method** checked, you can enter any method name in the **Method Name** box that begins with the word `validate` and when you click **OK,** JDeveloper will add that method to your entity object's custom Java class with the appropriate signature.

Finally, supply the text of the error message for the default locale that the end user should see if this validation rule fails.

Figure 9–3 Adding an Attribute-Level Method Validator



9.3.2 What Happens When You Create an Attribute-Level Method Validator

When you add a new method validator, JDeveloper updates the XML component definition to reflect the new validation rule. If you asked to have the method created, the method is added to the entity object's custom Java class. Example 9–1 illustrates a simple attribute-level validation rule that ensures the `AssignedDate` of a service request is a date in the current month. Notice that the method accepts an argument of the same type as the corresponding attribute, and that its conditional logic is based on the value of this incoming parameter. When the attribute validator fires, the attribute value has not yet been set to the new value in question, so calling the `getAssignedDate()` method inside the attribute validator for the `AssignedDate` attribute would return the attribute's *current* value, rather than the *candidate* value that the client is attempting to set.

Example 9–1 Simple Attribute-Level Method Validator

```
// In ServiceRequestImpl.java in SRDemo Sample
public boolean validateAssignedDate(Date  data) {
  if (data != null && data.compareTo(getFirstDayOfCurrentMonth()) <= 0) {
    return false;
  }
  return true;
}
```

Note: The return value of the `compareTo()` method is zero (`0`) if the two dates are equal, negative one (`-1`) if the first date is less than the second, or positive one (`1`) if the first date is greater than the second.

9.3.3 How to Create an Entity-Level Method Validator

To create an entity-level method validator:

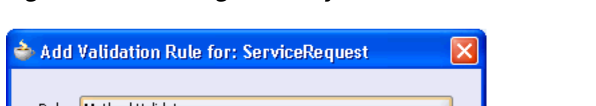
1. Open the Entity Object Editor.
2. If your entity object does not yet have a custom Java class, then first open the **Java** page and enable the generation of an **Entity Object Class**, and click **Apply** to generate the `*.java` file.
3. Open the **Validation** page and select root node in the tree that represents the entity object itself.
4. Click **New** to add a validation rule.
5. Select the **Method Validator** type from the **Rule** dropdown list, as shown in Figure 9–4.

The **Add Validation Rule** dialog displays the expected method signature for an entity-level validation method. You have two choices:

- If you already have a method in your entity object's custom Java class of the appropriate signature, it will appear in the list and you can select it after unchecking the **Create and Select Method** checkbox
- If you leave the **Create and Select Method** checked, you can enter any method name in the **Method Name** box that begins with the word `validate` and when you click **OK** JDeveloper will add that method to your entity object's custom Java class with the appropriate signature.

Finally, supply the text of the error message for the default locale that the end user should see if this validation rule fails.

Figure 9–4 Adding an Entity-Level Method Validator



9.3.4 What Happens When You Create an Entity-Level Method Validator

When you add a new method validator, JDeveloper updates the XML component definition to reflect the new validation rule. If you asked to have the method created, the method is added to the entity object's custom Java class. Example 9–2 illustrates a simple entity-level validation rule that ensures the `AssignedDate` of a service request comes after the `RequestDate`.

Example 9–2 Simple Entity-Level Method Validator

```
public boolean validateAssignedDateAfterRequestDate() {
  Date assignedDate = getAssignedDate();
  Date requestDate  = getRequestDate();
  if (assignedDate != null && assignedDate.compareTo(requestDate) < 0) {
    return false;
  }
  return true;
}
```

9.3.5 What You Might Need To Know About Translating Validation Rule Error Messages

Like the locale-specific UI control hints for entity object attributes, the validation rule error messages are added to the entity object's component message bundle file. These represent the strings for the default locale for your application. To provide translated versions of the validation error messages, follow the same steps as for translating the UI control hints that you've seen in previous chapters.

9.3.6 What You May Need to Know About Referencing the Invalid Value in an Attribute-Level Validation Error Message

The validation error message you supply when adding an attribute-level validation rule can reference the invalid value by referencing the message parameter token "`{3}`" in the string. The other error parameters supplied are useful for programmatic processing of the `ValidationException`, but not typically useful in the message string itself.

9.4 Assigning Programmatically-Derived Attribute Values

When declarative defaulting falls short of your needs, you can perform programmatic defaulting in your entity object:

- When an entity row is first created
- When the entity row is first created or when refreshed to blank again
- When the entity row is saved to the database
- When an entity attribute value is set

9.4.1 Defaulting Values for New Rows at Create Time

The `create()` method provides the entity object event you can handle to initialize default values the first time an entity row is created. Example 9–3 shows the overridden create method of the `ServiceHistory` entity object in the SRDemo application. It calls the attribute setter methods to populate the `SvhType`, `CreatedBy`, and `LineNo` attributes in a new service history entity row.

Example 9–3 Programmatically Defaulting Attribute Values for New Rows

```
// In ServiceHistoryImpl.java in SRDemo sample
protected void create(AttributeList nameValuePair) {
  super.create(nameValuePair);
  setSvhType(getDefaultNoteType());
  setCreatedBy(getCurrentUserId());
  setLineNo(new Number(getServiceRequest().getMaxHistoryLineNumber()+1));
}
```

9.4.1.1 Choosing Between create() and initDefaults() Methods

If an entity row has `New` status and you call the `refresh()` method on it, if you do not supply either the `REFRESH_REMOVE_NEW_ROWS` or `REFRESH_FORGET_NEW_ROWS` flag, then the entity row is returned to an `Initialized` status. As part of this process, the entity object's `initDefaults()` method is invoked, but not its `create()` method again. So override the `initDefaults()` method for programmatic defaulting logic that you want to fire both when the row is first created, as well as when it might be refreshed back to initialized status.

9.4.1.2 Eagerly Defaulting an Attribute Value from a Database Sequence

Section 6.6.3.7, "Synchronization with Trigger-Assigned Values" explained how to use the `DBSequence` type for primary key attributes whose values need to be populated by a database sequence at *commit* time. Sometimes you may want to eagerly allocate a sequence number at entity row creation time so that the user can see its value and so that this value does not change when the data is saved. To accomplish this, use the `SequenceImpl` helper class in the `oracle.jbo.server` package in an overridden `create()` method as shown in Example 9–4. It shows code from the custom Java class of the SRDemo application's `Product` entity object. After calling `super.create()`, it creates a new instance of the `SequenceImpl` object, passing the sequence name and the current transaction object. Then it calls the `setProdId()` attribute setter method with the return value from `SequenceImpl`'s `getSequenceNumber()` method.

Example 9–4 Eagerly Defaulting an Attribute's Value from a Sequence at Create Time

```
// In ProductImpl.java
import oracle.jbo.server.SequenceImpl;
// Default ProdId value from PRODUCTS_SEQ sequence at entity row create time
protected void create(AttributeList nameValuePair) {
  super.create(nameValuePair);
  SequenceImpl sequence = new SequenceImpl("PRODUCTS_SEQ",getDBTransaction());
  setProdId(sequence.getSequenceNumber());
}
```

9.4.2 Assigning Derived Values Before Saving

If you want to assign programmatic defaults for entity object attribute values before a row is saved, override the `prepareForDML()` method and call the appropriate attribute setter methods to populate the derived attribute values. In order to perform the assignment only during `INSERT`, `UPDATE`, or `DELETE`, you can compare the value of the `operation` parameter passed to this method against the integer constants `DML_INSERT`, `DML_UPDATE`, `DML_DELETE` respectively.

Example 9–5 shows the overridden `prepareForDML()` method used by the `ServiceHistory` entity object in the SRDemo application to automatically change the status of a service request when a service history note of certain types are created. When a new service history entry is inserted, this code changes the status of:

- A pending or closed service request to open if the new history note is added by a customer
- An open service request to pending if the new history note is added by a technician

Example 9–5 Assigning Derived Values Before Saving Using PrepareForDML

```
// In ServiceHistoryImpl.java
protected void prepareForDML(int operation, TransactionEvent e) {
  super.prepareForDML(operation, e);
  // If we are inserting a new service history entry...
  if (operation == DML_INSERT) {
    ServiceRequestImpl serviceReq = getServiceRequest();
    String historyType = getSvhType();
    // If request is pending or closed and customer adds note, status => Open
    if ((serviceReq.isPending() || serviceReq.isClosed())
        && CUSTOMER_TYPE.equals(historyType)) {
        serviceReq.setOpen();
    }
    // If request is open & technician adds a non-hidden note, status => Pending
    if (serviceReq.isOpen() && TECHNICIAN_TYPE.equals(historyType)) {
        serviceReq.setPending();
    }
  }
}
```

9.4.3 Assigning Derived Values When an Attribute Value is Set

To assign derived attribute values whenever another attribute's value is set, add code to the latter attribute's setter method. Example 9–6 shows the setter method for the `AssignedTo` attribute in the SRDemo application's `ServiceRequest` entity object. After the call to `setAttributeInternal()` to set the value of the `AssignedTo` attribute, it uses the setter method for the `AssignedDate` attribute to set its value to the current date and time.

Example 9–6 Setting the Assigned Date Whenever the AssignedTo Attribute Changes

```
// In ServiceRequestImpl.java
public void setAssignedTo(Number value) {
  setAttributeInternal(ASSIGNEDTO, value);
  setAssignedDate(getCurrentDateWithTime());
}
```

Note: It is safe to add custom code to the generated attribute getter and setter methods as shown here. When JDeveloper modifies code in your class, it intelligently leaves your custom code in place.

9.5 Undoing Pending Changes to an Entity Using the Refresh Method

You can use the `refresh(int flag)` method on any `Row` to refresh pending changes it might have. The `refresh()` method's behavior depends on the flag that you pass as a parameter. The three key flag values that control its behavior are the following constants in the `Row` interface.

- `REFRESH_WITH_DB_FORGET_CHANGES` forgets modifications made to the row in the current transaction and the row's data is refreshed from database. The latest data from database replaces data in the row regardless of whether the row was modified or not.
- `REFRESH_WITH_DB_ONLY_IF_UNCHANGED` works just like `REFRESH_WITH_DB_FORGET_CHANGES`, but for *unmodified* rows. If a row was already modified by this transaction, the row is not refreshed.
- `REFRESH_UNDO_CHANGES` works the same as `REFRESH_WITH_DB_FORGET_CHANGES` for *unmodified* rows. For a modified row, this mode refreshes the row with attribute values at the beginning of this transaction. The row remains in a modified state.

9.5.1 Controlling What Happens to New Rows During a Refresh

By default, any entity rows with `New` status that you `refresh()` are reverted back to blank rows in the `Initialized` state. Declarative defaults are reset, as well as programmatic defaults coded in the `initDefaults()` method, but the entity object's `create()` method is not invoked during this blanking-out process.

You can change this default behavior by combining one of the following two flags with one from the above section (using the bitwise-`OR` operator):

- `REFRESH_REMOVE_NEW_ROWS`, new rows are removed during refresh.
- `REFRESH_FORGET_NEW_ROWS`, new rows are marked `Dead`.

9.5.2 Cascading Refresh to Composed Children Entity Rows

You can cause a `refresh()` operation to cascade to composed child entity rows by bitwise-`OR`'ing the `REFRESH_CONTAINEES` flag with any of the valid flag combinations above. This causes the entity to invoke `refresh()` using the same mode on any composed child entities it contains.

9.6 Using View Objects for Validation

When your business logic requires performing SQL queries, the natural choice is to use a view object to perform that task. Keep in mind that SQL statements you execute for validation will "see" pending changes in the entity cache if they are entity-based view objects; read-only view objects will only retrieve data that has been posted to the database.

9.6.1 Creating View Objects at Runtime for Validation

Since entity objects are designed to be reused in any application scenario, they should not depend *directly* on a view object instance in any specific application module's data model. Doing so would prevent them from being reused in other application modules, which is highly undesirable.

Instead, your entity object can access the root application module in which it finds itself at runtime, and use that application module instances's `createViewObject()` method to create an instance of the "validation" view object it requires. As with view object instances added to the data model at design time, this API allows you to assign an instance name to the view object so you can use `findViewObject()` to find it again when needed.

Since the SQL-based validation code may be executed multiple times, it would not be the most efficient approach to create the view object instance each time it's needed and remove it when you are done using it. Instead, you can implement a helper method like what you see in Example 9–7 to use the view object instance if it already exists, or otherwise create it the first time it's needed. In order to ensure that the instance name of the runtime-created view object instance will not clash with the name of any design-time-specified ones in the data model, you can adopt the convention of constructing a name based on the view object definition name, prefixed by a string like "`Validation_`". This is just one approach. As long as the name doesn't clash with a design time supplied name, you can use any naming scheme.

Example 9–7 Helper Method to Access View Object for Validation

```
/**
 * Find instance of view object used for validation purposes in the
 * root application module. By convention, the instance of the view
 * object will be named Validation_your_pkg_YourViewObject.
 *
 * If not found, create it for the first time.
 *
 * @return ViewObject to use for validation purposes
 * @param viewObjectDefName
 */
protected ViewObject getValidationVO(String viewObjectDefName) {
  // Create a new name for the VO instance being used for validation
  String name = "Validation_" + viewObjectDefName.replace('.', '_');
  // Try to see if an instance of this name already exists
  ViewObject vo = getDBTransaction().getRootApplicationModule()
                                    .findViewObject(name);
  // If it doesn't already exist, create it using the definition name
  if (vo == null) {
    vo = getDBTransaction().getRootApplicationModule()
                           .createViewObject(name,viewObjectDefName);
  }
  return vo;
}
```

With a helper method like this in place, your validation code can call `getValidationVO()` and pass it the fully qualified name of the view object definition that it wants to use. Then you can write code like what you see in Example 9–8.

Example 9–8 Using a Validation View Object in a Method Validator

```
// Sample entity-level validation method
public boolean validateSomethingUsingViewObject() {
  Number numVal = getSomeEntityAttr();
  String stringVal = getAnotherEntityAttr();
  // Get the view object instance for validation
  ViewObject vo = getValidationVO("devguide.example.SomeViewObjectName");
  // Set it's bind variables (which it will typically have!)
  vo.setNamedBindWhereClauseParam("BindVarOne",numVal);
  vo.setNamedBindWhereClauseParam("BindVarTwo",stringVal);
  vo.executeQuery();
  if ( /* some condition */) {
    /*
     * code here returns true if the validation succeeds
     */
  }
  return false;
}
```

As the sample code suggests, view objects used for validation will typically have one or more named bind variables in them. Depending on the kind of data your view object retrieves, the "`/* some condition */`" expression above will look different. For example, if your view object's SQL query is selecting a `COUNT()` or some other aggregate, the condition will typically use the `vo.first()` method to access the first row, then use the `getAttribute()` method to access the attribute value to see what the database returned for the count.

If the validation succeeds or fails based on whether the query has returned zero or one row, the condition might simply test whether `vo.first()` returns `null` or not. If `vo.first()` returns `null`, there is no "first" row. In other words, the query retrieved no rows.

In other cases, you may be iterating over one or more query results retrieved by the view object to determine whether the validation succeeds or fails.

9.6.2 Implementing an Efficient Existence Check

One common kind of SQL-based validation is a simple test that a candidate foreign key value exists in a related table. This type of validation can be implemented using the `findByPrimaryKey()` method on the entity definition, however that will retrieve all attributes of the entity if the entity exists. An alternative approach to perform a lighter-weight existing check involves using a view object for validation.

Example 9–9 shows the `exists()` method that the SRDemo application's `Product` entity object contains in its custom entity definition class. First, it uses a variant of the helper method above that accepts a `DBTransaction` as a parameter to return the instance of the appropriate validation view object. This is encapsulated inside the `getProductExistsVO()` method in the same class.

This read-only view object used for validation is named `ProductExistsById` in the `oracle.srdemo.model.entities.validationqueries` package. Since this view object has a custom Java class (`ProductExistsByIdImpl`), the code in the `exists()` method can use the strongly-typed `setTheProductId()` method to set the named bind variable that the view object defines. Then the code executes the query and sets the boolean `foundIt` variable based on whether a row was found, or not.

Example 9–9 Efficient Existence Check Using View Object and Entity Cache

```
// In ProductDefImpl.java in SRDemo sample
public boolean exists(DBTransaction t, Number productId) {
  boolean foundIt = false;
  ProductExistsByIdImpl vo = getProductExistsVO(t);
  vo.setTheProductId(productId);
  vo.setForwardOnly(true);
  vo.executeQuery();
  foundIt = (vo.first() != null);
  /*
   * If we didn't find it in the database,
   * try checking against any new employees in cache
   */
  if (!foundIt) {
    Iterator iter = getAllEntityInstancesIterator(t);
    while (iter.hasNext()) {
      ProductImpl product = (ProductImpl)iter.next();
      /*
       * NOTE: If you allow your primary key attribute to be modifiable
       *       then you should also check for entities with entity state
       *       of Entity.STATUS_MODIFIED here as well.
       */
      if (product.getEntityState() == Entity.STATUS_NEW
          && product.getProdId().equals(productId)) {
        foundIt = true;
        break;
      }
    }
  }
  return foundIt;
}
```

Even though the SRDemo application currently does not allow the end user to create new products, it's good to implement the validation in a way that assumes the user might be able to do this in some future screens that are implemented. The code that follows the `executeQuery()` tests to see whether the candidate product ID is for a new `Product` entity that exists in the cache.

Recall that since the validation view object has no entity usages, its query will only "see" rows that are currently in the database. So, if the `foundIt` variable is false after trying the database, the remaining code gets an iterator for the `ProductImpl` entity rows that are currently in the cache and loops over them to see if any new `Product` entity row has the candidate product ID. If it does, the `exists()` method still returns `true`.

9.6.3 Validating Conditions Related to All Entities of a Given Type

The `beforeCommit()` method is invoked on *each* entity row in the pending changes list after the changes have been posted to the database, but before they are committed. This can be a perfect method in which to execute view object-based validations that must assert some rule over all entity rows of a given type.

Note: If your `beforeCommit()` logic can throw a `ValidationException`, you must set the `jbo.txn.handleafterpostexc` property to `true` in your configuration to have the framework automatically handle rolling back the in-memory state of the other entity objects that may have already successfully posted to the database (but not yet been committed) during the current commit cycle.

9.7 How to Access Related Entity Rows Using Association Accessors

Often your validation rules or programmatic defaulting of derived values may require consulting the values of associated entity rows. The association accessor methods in your entity object custom Java class make this task extremely easy. By calling the accessor method, you can easily access any related entity row — or `RowSet` of entity rows — depending on the cardinality of the association.

Example 9–10 shows an example of programmatic defaulting logic in use in the SRDemo application's `ServiceHistory` entity object. The line number of the new service history row is calculated by accessing the containing parent entity row of type `ServiceHistoryImpl`, and invoking a helper method called `getMaxHistoryLineNumber()` on it, before incrementing that value by one. If the parent entity row is already in the cache, the association accessor accesses the row from there. If not, it is brought into the cache using the primary key.

Example 9–10 Accessing Composing Parent Entity Row In a Create Method

```
// In ServiceHistoryImpl.java in SRDemo sample
protected void create(AttributeList nameValuePair) {
  super.create(nameValuePair);
  setSvhType(getDefaultNoteType());
  setCreatedBy(getCurrentUserId());
  setLineNo(new Number(getServiceRequest().getMaxHistoryLineNumber()+1));
}
```

Example 9–11 illustrates the code for the `getMaxHistoryLineNumber()` in the `ServiceRequest` entity object's custom Java class. It shows another use of an association accessor to retrieve the `RowSet` of children `ServiceHistory` rows (of type `ServiceHistoryImpl`) in order to calculate the maximum value of the `LineNo` attributes in the existing service history rows.

Example 9–11 Accessing Composed Children Entity Rows in a Calculation Using Association Accessor

```
// In ServiceRequestImpl.java in SRDemo Sample
public long getMaxHistoryLineNumber() {
  long max = 0;
  RowSet histories = (RowSet)getServiceHistories();
  if (histories != null) {
    while (histories.hasNext()) {
      long curLine = ((ServiceHistoryImpl)histories.next()).getLineNo()
                                                           .longValue();
      if (curLine > max) {
        max = curLine;
      }
    }
  }
  histories.closeRowSet();
  return max;
}
```

9.8 How to Reference Information About the Authenticated User

If you have set the `jbo.security.enforce` runtime configuration property to the value `Must` or `Auth`, the `oracle.jbo.server.SessionImpl` object provides methods you can use to get information about the name of the authenticated user and information about the roles of which they are a member. This is the implementation class for the `oracle.jbo.Session` interface that clients can access.

9.8.1 Referencing Role Information About the Authenticated User

The `oracle.jbo.Session` interface provides the two methods:

- `String[] getUserRoles()`, returns array of role names to which the user belongs
- `boolean isUserInRole(String roleName)`, returns `true` if user belongs to specified role

Your entity object code can access the `Session` by calling:

```
Session session = getDBTransaction().getSession();
```

Example 9–12 shows a helper method that uses this technique. It determines whether the current user is a technician by using the `isUserInRole()` method to test whether the user belongs to the `technician` role.

Example 9–12 Helper Method to Test Whether Authenticated User is in a Given Role

```
protected boolean currentUserIsTechnician() {
  return getDBTransaction().getSession().isUserInRole("technician");
}
```

After refactoring the constants into a separate `SRConstants` class, the SRDemo application contains helper methods like this in its base `SREntityImpl` class that all entity objects in the sample extend to inherit this common functionality:

```
protected boolean currentUserIsTechnician() {
  return getDBTransaction().getSession()
                          .isUserInRole(SRConstants.TECHNICIAN_ROLE);
}
protected boolean currentUserIsManager() {
  return getDBTransaction().getSession()
                          .isUserInRole(SRConstants.MANAGER_ROLE);
}
protected boolean currentUserIsCustomer() {
  return getDBTransaction().getSession()
                          .isUserInRole(SRConstants.USER_ROLE);
}
protected boolean currentUserIsStaffMember() {
  return currentUserIsManager() || currentUserIsTechnician();
}
```

These are then used by the `create()` method to conditionally default the service request type based on the role of the current user. The `getDefaultNoteType()` helper method:

```
// In ServiceHistoryImpl.java in SRDemo sample
private String getDefaultNoteType() {
  return currentUserIsStaffMember() ? TECHNICIAN_TYPE : CUSTOMER_TYPE;
}
```

is used by the `ServiceHistory` entity object's overridden `create()` method to default the service history type based on the role of the current user.

```
// In ServiceHistoryImpl.java in SRDemo sample
protected void create(AttributeList nameValuePair) {
  super.create(nameValuePair);
  setSvhType(getDefaultNoteType());
  setCreatedBy(getCurrentUserId());
  setLineNo(new Number(getServiceRequest().getMaxHistoryLineNumber()+1));
}
```

9.8.2 Referencing the Name of the Authenticated User

In order to access the name of the authenticated user, you need to cast the `Session` interface to its `SessionImpl` implementation class. Then you can use the `getUserPrincipalName()` method. Example 9–13 illustrates a helper method you can use in your entity object to retrieve the current user name.

Example 9–13 Helper Method to Access the Current Authenticated User Name

```
protected String getCurrentUserName() {
  SessionImpl session = (SessionImpl)getDBTransaction().getSession();
  return session.getUserPrincipalName();
}
```

9.9 How to Access Original Attribute Values

If an entity attribute's value has been changed in the current transaction, when you call the attribute getter method for it you will get this pending changed value. Using the `getPostedAttribute()` method, your entity object business logic can consult the original value for any attribute as it was read from the database before the entity row was modified. The method takes the attribute *index* as an argument, so pass the appropriate generated attribute index constants that JDeveloper maintains for you.

9.10 How to Store Information About the Current User Session

If you need to store information related to the current user session in a way that entity object business logic can reference, you can use the user data hashtable provided by the `Session` object. Consider how the SRDemo application is using it. When a new user accesses an application module for the first time, the `prepareSession()` method is called. As shown in Example 9–14, the `SRService` application module overrides `prepareSession()` to automatically retrieve information about the authenticated user by calling the `retrieveUserInfoForAuthenticatedUser()` method on the `LoggedInUser` view object instance. Then, it calls the `setUserIdIntoUserDataHashtable()` helper method to save the user's numerical ID into the user data hashtable.

Example 9–14 Overriding prepareSession() to Automatically Query User Information

```
// In SRServiceImpl.java in SRDemo Sample
protected void prepareSession(Session session) {
  super.prepareSession(session);
  /*
   * Automatically query up the correct row in the LoggedInUser VO
   * based on the currently logged-in user, using a custom method
   * on the LoggedInUser view object component.
   */
  getLoggedInUser().retrieveUserInfoForAuthenticatedUser();
  setUserIdIntoUserDataHashtable();
}
```

Example 9–15 shows the code for the `LoggedInUser` view object's `retrieveUserInfoForAuthenticatedUser()` method. It sets its own `EmailAddress` bind variable to the name of the authenticated user from the session and then calls `executeQuery()` to retrieve the additional user information from the `USERS` table.

Example 9–15 Accessing Authenticated User Name to Retrieve Additional User Details

```
// In LoggedInUserImpl.java
public void retrieveUserInfoForAuthenticatedUser() {
  SessionImpl session = (SessionImpl)getDBTransaction().getSession();
  setEmailAddress(session.getUserPrincipalName());
  executeQuery();
  first();
}
```

One of the pieces of information about the authenticated user that the `LoggedInUser` view object retrieves is the user's numerical ID number, which that method returns as its result. For example, the user `sking` has the numeric `UserId` of `300`.

Example 9–16 shows the `setUserIdIntoUserDataHashtable()` helper method — used by the `prepareSession()` code above — that stores this numerical user ID in the user data hashtable, using the key provided by the string constant `SRConstants.CURRENT_USER_ID`.

Example 9–16 Setting Information into the UserData Hashtable for Access By Entity Objects

```
// In SRServiceImpl.java
private void setUserIdIntoUserDataHashtable() {
  Integer userid = getUserIdForLoggedInUser();
  Hashtable userdata = getDBTransaction().getSession().getUserData();
  if (userdata == null) {
    userdata = new Hashtable();
  }
  userdata.put(SRConstants.CURRENT_USER_ID, userid);
}
```

Both the `ServiceRequest` and the `ServiceHistory` entity objects have an overridden `create()` method that references this numerical user ID using a helper method like the following to set the `CreatedBy` attribute programmatically to the value of the currently authenticated user's numerical user ID.

```
protected Number getCurrentUserId() {
  Hashtable userdata = getDBTransaction().getSession().getUserData();
  Integer userId = (Integer)userdata.get(SRConstants.CURRENT_USER_ID);
  return userdata != null ? Utils.intToNumber(userId):null;
}
```

9.11 How to Access the Current Date and Time

You might find it useful to reference the current date and time in your entity object business logic. Example 9–17 shows a helper method you can use to access the current date without any time information.

Example 9–17 Helper Method to Access the Current Date with No Time

```
/*
 * Helper method to return current date without time
 *
 * Requires import: oracle.jbo.domain.Date
 */
protected Date getCurrentDate() {
  return new Date(Date.getCurrentDate());
}
```

In contrast, if you need the information about the current time included as part of the current date, use the helper method shown in Example 9–18.

Example 9–18 Helper Method to Access the Current Date with Time

```
/*
 * Helper method to return current date with time
 *
 * Requires imports: oracle.jbo.domain.Date
 *                   java.sql.Timestamp
 */
protected Date getCurrentDateWithTime() {
  return new Date(new Timestamp(System.currentTimeMillis()));
}
```

9.12 How to Send Notifications Upon a Successful Commit

The `afterCommit()` method is invoked on *each* entity row that was in the pending changes list and got successfully saved to the database. You might use this method to send an email notification about the change in state of an entity.

9.13 How to Conditionally Prevent an Entity Row from Being Removed

The `remove()` method is invoked on an entity row before it is removed. You can conditionally throw a `JboException` in this method to prevent a row from being removed if the appropriate conditions are not met.

> **Note:** The entity object offers declarative prevention of deleting a master entity row that has existing, composed children rows. You configure this option on the **Association Properties** page of the **Association Editor** for the composition.

9.14 How to Implement Conditional Updatability for Attributes

You can override the `isAttributeUpdateable()` method in your entity object class to programmatically determine whether a given attribute is updatable or not at runtime based on appropriate conditions. Example 9–19 shows how the `ServiceHistory` entity object in the SRDemo application overrides this method to enforce that its `SvhType` attribute is updatable only if the current authenticated user is a staff member. Notice that when the entity object fires this method, it passes in the integer attribute index whose updatability is being considered. You implement your conditional updatability logic for a particular attribute inside an `if` or `switch` statement based on the attribute index. Here `SVHTYPE` is referencing the integer attribute index constants that JDeveloper automatically maintains in your entity object custom Java class.

Example 9–19 Conditionally Determining an Attribute's Updatability at Runtime

```
// In ServiceHistoryImpl.java
public boolean isAttributeUpdateable(int index) {
  if (index == SVHTYPE) {
    if (!currentUserIsStaffMember()) {
      return super.isAttributeUpdateable(index);
    }
    return CUSTOMER_TYPE.equals(getSvhType()) ? false : true;
  }
  return super.isAttributeUpdateable(index);
}
```

Note: Entity-based view objects inherit this conditional updatability as they do everything else encapsulated in your entity objects. Should you need to implement this type of conditional updatability logic in a way that is specific to a transient view object attribute, or to enforce some condition that involves data from multiple entity objects participating in the view object, you can override this same method in a view object's view row class to achieve the desired result.

9.15 Additional Resources

The *Business Rules in ADF Business Components* whitepaper by Oracle Consulting outlines a formal approach to classifying and implementing virtually every kind of real-world business rule they have encountered in their project implementations using Oracle ADF using ADF Business Components. You can access it from the Oracle JHeadstart Product Center on OTN.

10

Overview of Application Module Data Binding

Using the SRDemo application's `SRService` as an example, this chapter describes how an application module's active data model and business service interface methods appear at design time for drag and drop data binding and how they are accessible at runtime by the ADF Model data binding layer using the application module data control.

This chapter includes the following sections:

- Section 10.1, "Overview of Data Controls and Declarative Bindings"
- Section 10.2, "Understanding the Application Module Data Control"
- Section 10.3, "How an Application Module Appears in the Data Control Palette"
- Section 10.5, "Application Module Databinding Tips and Techniques"
- Section 10.6, "Overview of How SRDemo Pages Use the SRService"

10.1 Overview of Data Controls and Declarative Bindings

The Oracle ADF Model layer is a declarative data binding facility. It implements the two concepts in the JSR-227 specification that enable decoupling the user interface technology from the business service implementation: *data controls* and *declarative bindings*. Data controls and declarative bindings enable a unified design time and runtime approach to bind any user interface to any backend business service without code.

10.1.1 Data Controls Abstract the Implementation Technology of a Business Service

Data controls abstract the implementation technology of a business service by using standard metadata interfaces to describe the service's operations and data collections. This includes information about the properties, methods, and types involved. At design time, visual tools like JDeveloper can leverage the standard service metadata to simplify binding UI components any data control operation or data collection. 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 business service.

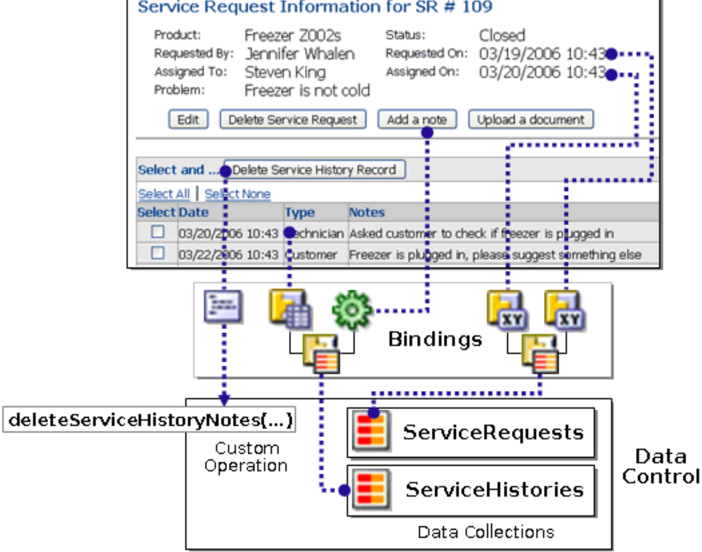
10.1.2 Bindings Connect UI Controls to Data Collections and Operations

Declarative bindings abstract the details of accessing data from data collections in a data control and of invoking its operations. There are three basic kinds of declarative binding objects that automate the key aspects of data binding that all enterprise applications require:

- *Iterator bindings*, which bind to an iterator that tracks the current row in a data collection
- *Value bindings*, which connect UI components to attributes in a data collection
- *Action bindings*, which invoke custom or built-it operations on a data control or its data collections

Iterator bindings simplify building user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information. UI components that display data use value bindings. Value bindings range from the most basic variety that works with a simple text field to more sophisticated list, table, and tree bindings that support the additional needs of list, table, and tree UI controls. An action binding is used by UI components like hyperlinks or buttons to invoke methods. An action binding allows the user to click on the component to invoke a business service without code. There are two kinds of action bindings: a regular *action binding* that invokes a built-in operation, and a *method action binding* that invokes a custom operation.

Figure 10–1 Bindings Connect UI Components to Data Control Collections and Operations



Note: *Value bindings* are bindings that have a bound attribute value. All value bindings implement the `oracle.binding.AttributeBinding` interface. The interface for action bindings is `oracle.binding.OperationBinding`. Since both of these kinds of binding interfaces are related to UI controls, they both extend the `oracle.binding.ControlBinding` interface. The term *control bindings* is used in this guide to describe things that are common to both value bindings and action bindings.

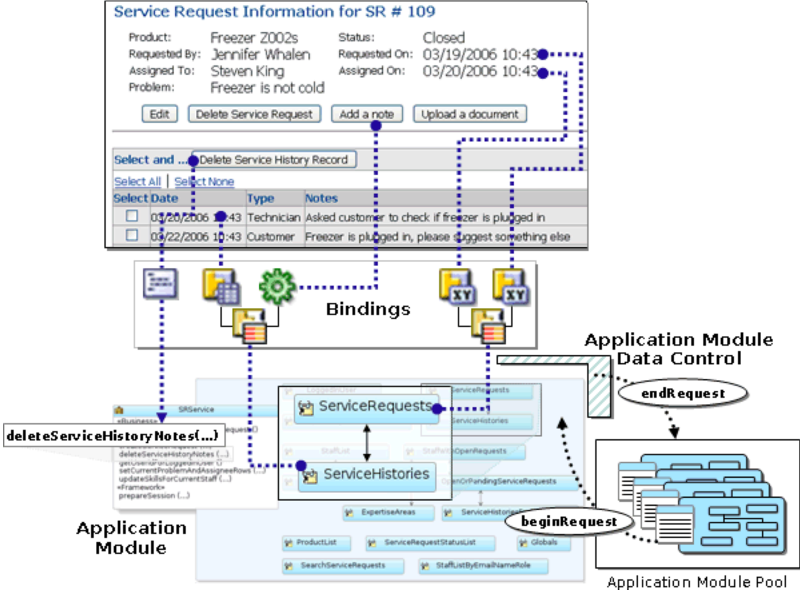
10.2 Understanding the Application Module Data Control

The application module data control is one of the several data control implementations supplied with Oracle ADF. Its job is to be a thin adapter over an application module pool that automatically acquires an available application module instance at the beginning of the request. During the current request the application module data control holds a reference to the application module instance on behalf of the current user session. At the end of the request, the application module data control releases the application module instance back to the pool. Importantly, the application module component directly implements the interfaces that the binding objects expect for data collections, built-in operations, and service methods. This allows the bindings to work *directly* with the application modules and the view object instances in its active data model. Specifically, this optimized interaction allows:

- Iterator bindings to directly bind to the default row set iterator of the default row set of any view object instance
- Action bindings to directly bind to either:
  - Custom methods on the application module's client interface
  - Built-in operations of the application module and view objects

Figure 10–2 illustrates the pool management role the application module data control plays and highlights the direct link between the bindings and the application module instance.

Figure 10–2 Bindings Connect Directly to View Objects and Methods of an Application Module from a Pool



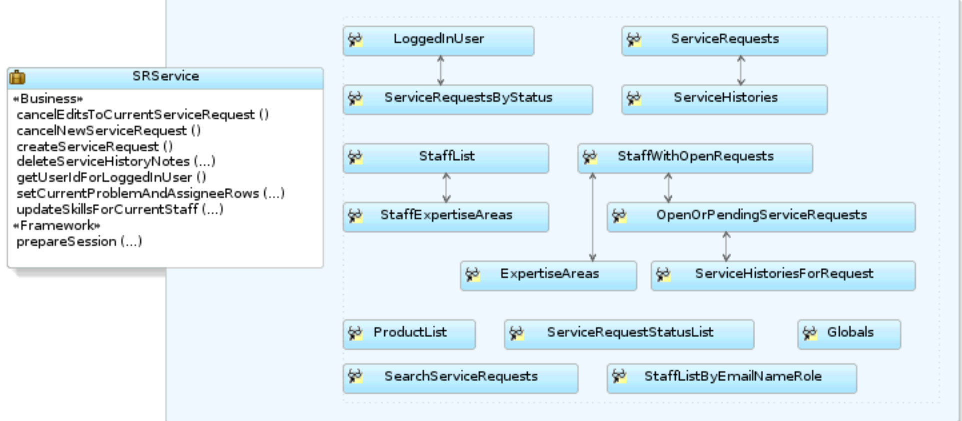
10.3 How an Application Module Appears in the Data Control Palette

At design time, you use the Data Control Palette to perform drag and drop data binding for JSF, JSP/Struts, and Swing applications. Each application module in the workspace appears automatically in the Data Control Palette. Using the `SRService` application module from the SRDemo application as an example this section highlights exactly what you'll see in the Data Control Palette when you work with your own application modules.

10.3.1 Overview of the SRService Application Module

Figure 10–3 shows the SRService application module that implements the business service layer of the SRDemo application. Notice that its data model includes numerous view object instances, including several master/detail hierarchies. The view layer of the demo consists of JSF pages whose UI components are bound to data from the view object instances in the SRService's active data model, and to built-in operations and service methods on its client interface. In Section 10.6, "Overview of How SRDemo Pages Use the SRService", you'll find an overview of exactly what aspects of this application module each page uses.

Figure 10–3 UML Diagram of the SRDemo Application's oracle.srdemo.model.SRService Application Module



10.3.2 How to Change the Data Control Name Before You Begin Building Pages

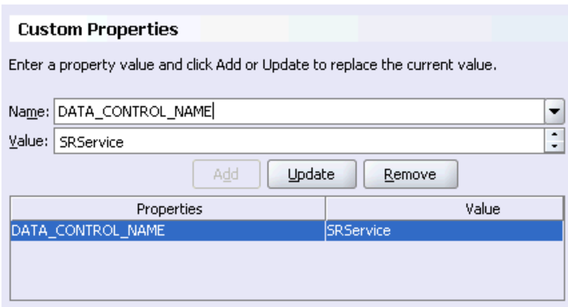
By default, an application module will appear in the Data Control Palette as a data control named *AppModuleName*DataControl. For example, the SRService originally had the name SRServiceDataControl. To change the default data control name to a shorter, or simply more preferable name, do the following:

To change the application module name:

1. Open the application module in the Application Module Editor.
2. Open the **Custom Properties** page.
3. In the **Name** combobox, select the DATA_CONTROL_NAME property from the dropdown list.
4. Enter your preferred data control name in the **Value** field and click OK to close the wizard.

Figure 10–4 shows the custom property setting that changed the data control name of the SRService from the default SRServiceDataControl to the shorter SRService name that matches the name of the application module. You'll notice the change immediately in the Data Control Palette.

Figure 10–4 Setting the Custom Application Module Property to Override the Data Control Name



Note that as you begin to bind data from the application module to your application pages or panels, the data control name for your application module will appear in the `DataBindings.cpx` file in your user interface project and in each data binding page definition XML file. In addition, you might refer to the data control name in code when needing to work programmatically with the application module service interface. For this reason, if you plan to change the name of your application module, Oracle recommends doing this change before you being building your view layer.

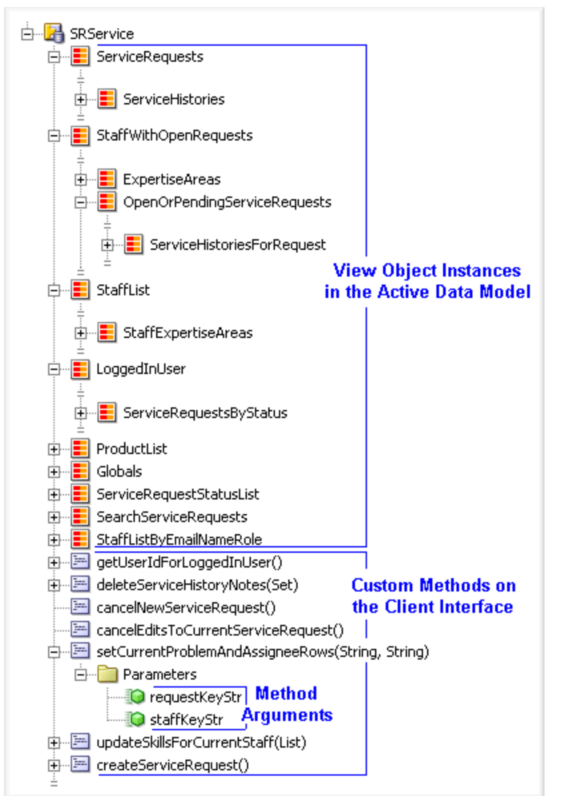
Note: In JDeveloper Release 3, if you decide to change the application module's data control name after you have already referenced it in one or more pages, you will need to open the page definition files where it is referenced and update the old name to the new name manually. Future releases of JDeveloper may extend its refactoring support to make renaming a data control simpler.

10.3.3 How the Data Model and Service Methods Appear in the Data Control Palette

Figure 10–5 illustrates how the Data Control Palette displays the view object instances in the `SRService`'s active data model. Each view object instance appears as a named data collection whose name matches the view object instance name. Note the hierarchical *structure* of the data collections, and that for viewing simplicity, the figure omits some details in the tree that appear for each view object. These additional view object details are highlighted in Section 10.3.6, "How View Objects Appear in the Data Control Palette". The Data Control Palette reflects the master/detail hierarchies in your application module data model by displaying detail data collections nested under their master data collection.

The Data Control Palette also displays each custom method on the application module's client interface as a named data control custom operation whose name matches the method name. If a method accepts arguments, they appear in a **Parameters** folder as operation parameters nested inside the operation node.

Figure 10–5 How the Active Data Model Appears in the Data Control Palette



10.3.4 How to Change View Instance Names Before You Begin Building Pages

When you initially add a view object instance to the data model, if you haven't typed in an instance name yourself, it gets added with a name composed of the view object name with a numeric suffix appended. For example, adding an instance of a `ServiceRequests` view object to the data model, the default view object instance name would be `ServiceRequests1`. You can easily rename the view object instances in the **Data Model** page of the Application Module Editor.

As you begin to bind data from the data collections in the Data Control Palette to your application pages or panels, in addition to the data control name, the data *collection* names will be referenced in the page definition XML files used by the ADF Model data binding layer. Since the names of your view object instance in the application module data model are used as the names of these data collections, Oracle recommends reviewing your view object instance names before using them to build data bound pages to ensure the names are descriptive.

Note: In JDeveloper Release 3, if you decide to change a view object instance name after you have already referenced it in one or more pages, you will need to open the page definition files where it is referenced and update the old name to the new name manually. Future releases of JDeveloper may extend its refactoring support to make renaming a view object instance simpler.

10.3.5 How Transaction Control Operations Appear in the Data Control Palette

The application module data control exposes two data control built-in operations named `Commit` and `Rollback` as shown in Figure 10–6. At runtime, when these operations are invoked by the data binding layer, they delegate to the `commit()` and `rollback()` methods of the `Transaction` object associated with the current application module instance. Note that the **Operations** folder in the data controls tree omits all of the data collections and custom operations for a more streamlined view.

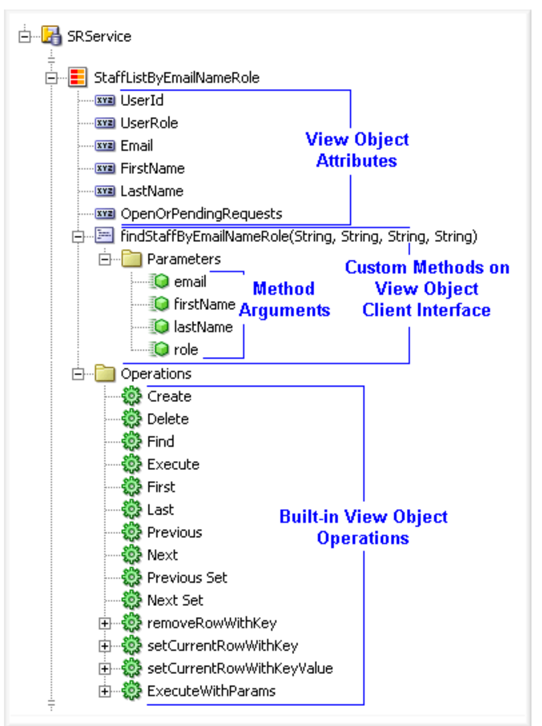
Note: In an application module with many view object instances and custom methods, you may need to scroll the Data Control Palette display to find the **Operations** folder that is the direct child node of the data control. This folder is the one that contains its built-in operations.

Figure 10–6 How Transaction Control Operations Appear in the Data Control Palette

10.3.6 How View Objects Appear in the Data Control Palette

Figure 10–7 shows how each view object instance in the application module's data model appears in the Data Control Palette. The view object attributes are displayed as immediate child nodes of the corresponding data collection. If you have selected any custom methods to appear on the view object's client interface, like the `performSearch()` method in the figure, they appear as custom operations immediately following the view object attribute at the same level. If the method accepts arguments, these appear in a nested **Parameters** folder as operation parameters.

Figure 10–7 How View Objects Appear in the Data Control Palette



10.3.6.1 Built-in Operations for View Object Data Collections

As shown in Figure 10–7, the **Operations** folder under the data collection displays all its available built-in operations. If an operation accepts one or more parameters, then they appear in a nested **Parameters** folder. At runtime, when one of these data collection operations is invoked by name by the data binding layer, the application module data control delegates the call to an appropriate method on the `ViewObject` interface to handle the built-in functionality. The built-in operations fall into three categories: operations that affect the current row, operations that refresh the data collection, and all other operations.

10.3.6.1.1 Operations that affect the current row

- `Create` — creates a new row that becomes the current row
- `Delete` — deletes the current row
- `First` — sets the current row to the first row in the row set
- `Last` — sets the current row to the last row in the row set
- `Previous` — sets the current row to the previous row in the row set
- `Next` — sets the row to the next row in the row set
- `Previous Set` — navigates forward one full page of rows
- `Next Set` — navigates backward one full page of rows
- `setCurrentRowWithKey` — tries to finds a row using the serialized string representation of row key passed as a parameter. If found, it becomes the current row.
- `setCurrentRowWithKeyValue` — tries to finds a row using the primary key attribute value passed as a parameter. If found, it becomes the current row.

10.3.6.1.2 Operations that refresh the data collection

- `Execute` — refreshes the data collection by (re)executing the view object's query, leaving any bind parameters at their current values.
- `ExecuteWithParams` — refreshes the data collection by first assigning new values to the named bind variables passed as parameters, then (re)executing the view object's query.

Note: The `ExecuteWithParams` operation only appears for view objects that have defined one or more named bind variables at design time.

10.3.6.1.3 All other operations

- `removeRowWithKey` — tries to finds a row using the serialized string representation of row key passed as a parameter. If found, the row is removed.
- `Find` — toggles "Find Mode" on and off for data collection

10.3.7 How Nested Application Modules Appear in the Data Control Palette

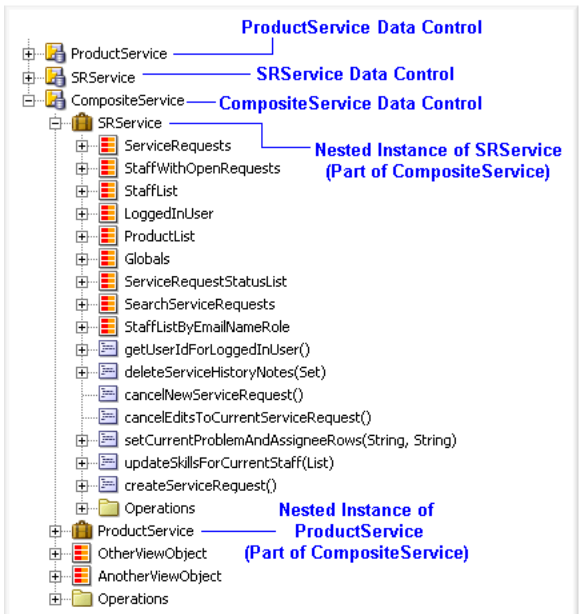
If you build composite application modules, by including nested instances of other application modules, the Data Control Palette reflects this component assembly in the tree hierarchy. For example, assume that in addition to the SRDemo application’s `oracle.srdemo.model.SRService` application module that you have also created the following application modules in the same package:

- An application module named `ProductService`, and renamed its data control to `ProductService`
- An application module named `CompositeService`, and renamed its data control to `CompositeService`

Then assume that you've added two view object instances named `OtherViewObject` and `AnotherViewObject` to the data model of `CompositeService` and that on **Application Modules** page of the Application Module Editor you have added an instance of the `SRService` application module and an instance of the `ProductMaintenance` application module to reuse them as part of `CompositeService`. Figure 10–8 illustrates how your `CompositeService` would appear in the Data Control Palette. The nested instances of SRService and ProductService appear in the palette tree display nested inside of the `CompositeService` data control. The entire data model and set of client methods that the nested application module instances expose to clients are automatically available as part of the `CompositeService` that reuses them.

One possibly confusing point is that even though you have reused nested instances of `SRService` and `ProductService` inside of `CompositeService`, the `SRService` and `ProductService` application modules also appear themselves as top-level data control nodes in the palette tree. JDeveloper assumes that you might want to sometimes use `SRService` or `ProductService` on their own as a separate data controls from `CompositeService`, so it displays all three of them. You need to be careful to perform your drag and drop data binding from the correct data control. If you want your page to use a view object instance from the nested `SRService` instance's data model that is an aggregated part of the `CompositeService` data control, then ensure you select the data collection that appears as part of the `CompositeService` data control node in the palette.

Figure 10–8 How Nested Application Modules Appear in the Data Control Palette



It is important to do the drag and drop operation that corresponds to your intended usage. When you drop a data collection from the *top-level* `SRService` data control node in the palette, at runtime your page will use an instance of the `SRService` application module acquired from a pool of `SRService` components. When you drop a data collection from the *nested* instance of `SRService` that is part of `CompositeService`, at runtime your page will use an instance of the `CompositeService` application module acquired from a pool of `CompositeService` components. Since different types of application module data controls will have distinct transactions and database connections, inadvertently mixing and matching data collections from both a nested application module and a top-level data control will lead to unexpected runtime behavior. Forewarned is forearmed.

10.4 How to Add a Create Button on a Page

To add a Create button, drag and drop a `Create` operation for a data collection from the Data Control Palette onto a JSP page or Swing panel.

Note: The built-in `Create` operation behaves differently in a web-based application than in a Swing-based desktop application. For web applications, depending on the situation, you may want to use `CreateInsert` instead of Create.

10.4.1 What Happens When You Drop a Create Button on a Web Page

If you drag a `Create` operation for a data collection from the Data Control Palette onto a JSP page — whether using JSF or not — you'll get a **Create** button on your page that is declaratively bound to the built-in `Create` operation. The `Create` operation creates a new row for the data collection, but *does not* insert it into the data collection's row set. By not inserting the new row into the row set at creation time, it helps avoid having an unwanted blank row appear in other pages should the user navigate away from your create page before actually entering any data for the new row. After you've

invoked a `Create` operation, the iterator binding temporarily points to this new row as if it were the current row. When the user successfully submits data for the attributes in the new row, the new row gets inserted into the row set at that time. This is the declarative row creation approach that works best for most web application use cases.

10.4.2 What Happens When You Drop a Create Operation Onto a Swing Panel

If you drag a `Create` operation for a data collection from the Data Control Palette onto a Swing panel, you'll get a **Create** button on your panel that is declaratively bound to a built-in operation called `CreateInsert` instead. The `CreateInsert` operation creates a new row in the data collection and inserts that new row into the collection just before the current row. CreateInsert is the best approach for Swing applications.

10.4.3 When to Use CreateInsert Instead of Create

There are some situations in a web application where you need to use `CreateInsert` instead of `Create`. Use CreateInsert when creating:

- An editable table control
- A table with a single, current editable row
- A Master/detail page and want the newly created master row to correctly show no existing detail rows

`CreateInsert` is used when a new row needs to be inserted into the row set. Since only the one `Create` operation shows in the Data Control Palette, to use a `CreateInsert` operation in a web application involves a couple of additional steps.

How to Change a Create Operation to CreateInsert:

1. Drop the `Create` operation from the Data Control Palette onto your page
2. Select the button in the visual editor and choose **Edit Binding...** from the context menu
3. In the **Action Binding Editor** use the **Select an Action** dropdown list to change the action binding from using the `Create` to using `CreateInsert` instead.

10.4.4 What You May Need to Know About Create and CreateInsert

When you use the `Create` or `CreateInsert` operations to declaratively create a new row, under the covers they end up performing the following lines of code:

```
// create a new row for the view object
Row newRow = yourViewObject.createRow();
// mark the row as being "initialized", but not yet new
newRow.setNewRowState(Row.STATUS_INITIALIZED);
```

In addition, if you are using the `CreateInsert` operation, it performs the additional line of code to insert the row into the row set:

```
// insert the new row into the view object's default rowset
yourViewObject.insertRow(newRow);
```

When you create a row in an entity-based view object, the `Transaction` object associated to the current application module immediately takes note of the fact. The new entity row that gets created behind the view row is already part of the `Transaction`'s list of pending changes. When a newly created row is marked as having the *initialized* state, it is removed from the `Transaction`'s pending changes

list and is considered a blank row in which the end user has not yet entered any data values. The term *initialized* is appropriate since the end user will see the new row initialized with any default values that the underlying entity object has defined. If the user never enters any data into any attribute of that initialized row, then it is as if the row never existed. At transaction commit time, since that row is not part of the `Transaction`'s pending changes list, no `INSERT` statement will be attempted for it.

As soon as at least one attribute in an initialized row is set, it automatically transitions from the initialized status to the new status (`Row.STATUS_NEW`). At that time, the underlying entity row gets enrolled in the `Transaction`'s list of pending changes, and the new row will be permanently saved the next time you commit the transaction.

Note: If the end user performs steps while using your application that result in creating many initialized rows but never populating them, it might seem like a recipe for a slow memory leak. Not to worry, however. The memory used by an initialized row that never transitions to the new state will eventually be reclaimed by the Java virtual machine's garbage collector.

10.5 Application Module Databinding Tips and Techniques

10.5.1 How to Create a Record Status Display

When building pages you'll often want to display some kind of record status indicator like: "Record 5 of 25". If you display multiple rows on a page, then you may also want to display a variant like "Record 5-10 of 25". You can build a record indicator like this using simple text components, each of which displays an appropriate value from an iterator binding or table binding using an EL expression. The iterator binding's `rangeSize` property defines how many rows per page it makes available to display in the user interface. If your page definition contains either an iterator binding named *`SomeViewIter`* or a table binding named *`SomeView`*, you can reference the following EL expressions:

- Number of Rows per Page

  ```
  #{bindings.SomeViewIter.rangeSize}
  ```

  ```
  #{bindings.SomeView.rangeSize}
  ```

- Total Rows

  ```
  #{bindings.SomeViewIter.estimatedRowCount}
  ```

  ```
  #{bindings.SomeView.estimatedRowCount}
  ```

- First Row on the Current Page

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

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

- Last Row on the Current Page

  ```
  #{bindings.SomeViewIter.rangeStart +
  bindings.SomeViewIter.rangeSize}
  ```

  ```
  #{bindings.SomeView.rangeStart + bindings.SomeView.rangeSize}
  ```

- Current Row Number

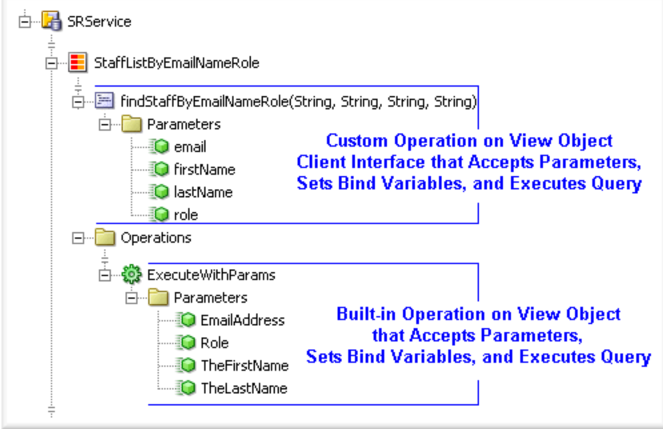
```
#{bindings.SomeViewIter.rangeStart +
bindings.SomeViewIter.currentRowIndexInRange + 1}
```

```
#{bindings.SomeView.currentRowIndex + 1}
```

10.5.2 How to Work with Named View Object Bind Variables

When a view object has named bind variables, an additional `ExecuteWithParams` operation appears in the corresponding data collection's **Operations** folder. As shown in Figure 10–9, this built-in operation accepts one parameter corresponding to each named bind variable. For example, the `StaffListByEmailNameRole` view object in the figure has four named bind variables — `EmailAddress`, `Role`, `TheFirstName`, and `TheLastName` — so the parameters appear for the `ExecuteWithParams` action having these same names. At runtime, when the `ExecuteWithParams` built-in operation is invoked by name by the data binding layer, each named bind variable value is set to the respective parameter value passed by the binding layer, and then the view object's query is executed to refresh its row set of results.

Figure 10–9 Work with Named Bind Variables Using Either a Built-in or Custom Operation



An alternative approach, also shown in Figure 10–9, involves creating a custom view object "finder" method that accepts the arguments you want to allow the user to set and then including this method on the client interface of the view object in the View Object Editor. Example 10–1 shows what the code for such a custom method would look like. It is taken from the `StaffListByEmailNameRole` view object in the SRDemo application. Notice that due to the application module's active data model, the method does not need to return the data to the client. The method has a `void` return type, sets the bind variable values to the parameter values passed into the method using the generated bind variable accessor methods `setEmailAddress()`, `setRole()`, `setTheFirstName()`, and `setTheLastName()`. Then, it calls `executeQuery()` to execute the query to refresh the view object's row set of results.

Example 10–1 Custom View Object Method Sets Named Bind Variables and Executes Query

```
// From SRDemo Sample's StaffListByEmailNameRoleImpl.java
public void findStaffByEmailNameRole(String email,
                                     String firstName,
                                     String lastName,
                                     String role)
{
  setEmailAddress(email);
  setRole(role);
  setTheFirstName(firstName);
  setTheLastName(lastName);
  executeQuery();
}
```

Both of these approaches accomplish the same end result. Both can be used with equal ease from the Data Control Palette to create a search form. The key differences between the approaches are the following:

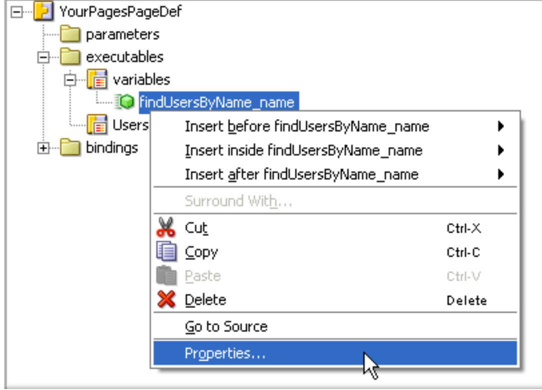
Using the built-in ExecuteWithParams operation

- You don't need to write code since it's a built-in feature.
- You can drop the operation from the Data Control Palette to create an **ADF Search Form** to allow users to search the view object on the named bind variable values.
- Your search fields corresponding to the named bind variables inherit bind variables UI control hints for their label text that you can define as part of the view object component.

Using the custom View Object "finder" operation

- You need to write a little custom code, but you see a top-level operation in the Data Control Palette whose name can help clarify the intent of the find operation.
- You can drop the custom operation from the Data Control Palette to create an **ADF Search Form** to allow users to search the view object on the parameter values.
- Your search fields corresponding to the method arguments do not inherit label text UI control hints you define on the view object's named bind variables themselves. Instead, you need to define UI control hints for their label text by using the **Properties...** choice from the context menu of the page definition variable corresponding to the method argument as shown in Figure 10–10.

Figure 10–10 Accessing Properties of Page Definition Variables to Set UI Control Hints



In short, it's good to be aware of both approaches and you can decide for yourself which approach you prefer. As described in Section 10.6.5, "The SRStaffSearch Page", while the `StaffListByEmailNameRole` view object contains the above `findStaffByEmailNameRole()` custom method for educational purposes, the demo's JSF page uses the declarative `ExecuteWithParams` built-in action.

10.5.3 How to Use Find Mode to Implement Query-by-Example

In the ADF Model layer, an iterator binding supports a feature called "find mode" when working with data collections that support query-by-example. As cited in Section 5.8, "Filtering Results Using Query-By-Example View Criteria", view objects support query-by-example when view criteria row set of view criteria rows has been applied. The view criteria rows have the same structure as the rows in the view object, but the datatype of each attribute is `String` to allow criteria like "`> 304`" or "`IN (314,326)`" to be entered as search criteria.

In support of the find mode feature, an iterator binding has a boolean `findMode` property that defaults to `false`. As shown in Figure 10–11, when `findMode` is `false` the iterator binding points at the row set containing the rows of data in the data collection. In contrast, when you set `findMode` to `true` the iterator binding switches to point at the row set of view criteria rows.

Figure 10–11 When Find Mode is True, Iterator Binding Points at ViewCriteria Row Set Instead

Data Collection

ViewCriteria

UserId	Email	LastName
> 304	d%	
IN (324,326)		Baer

RowSet

UserId	Email	LastName
305	daustin	Austin
307	dlorentz	Lorentz
324	hbaer	Baer

findMode == true / false — Iterator Binding

In the binding layer, action bindings that invoke built-in data control operations are associated to an iterator binding in the page definition metadata. This enables the binding layer to apply an operation like `Create` or `Delete`, for example, to the correct data collection at runtime. When an operation like `Create` or `Delete` is invoked for an iterator binding that has `findMode` set to `false`, these operations affect the row set containing the data. In contrast, if the operations are invoked for an iterator binding that has `findMode` set to `true`, then they affect the row set containing the view criteria row, effectively creating or deleting a row of query-by-example criteria.

The built-in `Find` operation allows you to toggle an iterator binding's `findMode` flag. If an iterator binding's `findMode` is `false`, invoking the `Find` operation for that iterator binding sets the flag to `true`. The UI components that are bound to attribute bindings related to this iterator switch accordingly to displaying the current view criteria row in the view criteria row set. If the user modifies the values of UI components while their related iterator is in find mode, the values are applied to the view criteria row in the view criteria row set. If you invoke the `Execute` or `ExecuteWithParams` built-in operation on an iterator that is in find mode, each will first toggle find mode to false, apply the find mode criteria, and refresh the data collection.

If an iterator binding's `findMode` is `true` invoking the `Find` operation sets it to `false` and removes the view criteria rows in the view criteria row set. This effectively cancels the query-by-example mode.

10.5.4 How to Customize the ADF Page Lifecycle to Work Programmatically with Bindings

The ADF Controller layer integrates the JSF page lifecycle with the ADF Model data binding layer. You can customize this integration either globally for your entire application, or on a per-page basis. This section highlights some tips related to how the SRDemo application illustrates using both of these techniques.

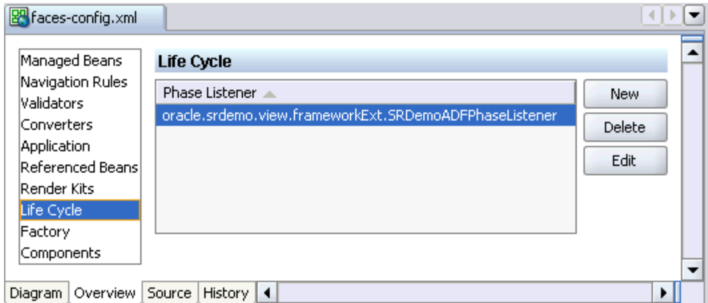
10.5.4.1 Globally Customizing the ADF Page Lifecycle

To globally customize the ADF Page Lifecycle, do the following:

- Create a class that extends `oracle.adf.controller.faces.lifecycle.FacesPageLifecycle`
- Create a class that extends `ADFPhaseListener` and overrides the `createPageLifecycle()` method to return an instance of your custom page lifecycle class.
- Change your `faces-config.xml` file to use your subclass of `ADFPhaseListener` *instead of* the default `ADFPhaseListener`. As shown in Figure 10–12, you can do this on the **Overview** tab of the JDeveloper `faces-config.xml` editor, in the **Life Cycle** category.

> **Note:** Make sure to *replace* the existing ADFPhaseListener with your custom subclass of ADFPhaseListener, or everything in the JSF / ADF lifecycle coordination will happen twice!

Figure 10–12 Setting Up a Custom ADFPhaseListener To Install a Custom Page Lifecycle Globally



The SRDemo application includes a `SRDemoPageLifecycle` class that globally overrides the `reportErrors()` method of the page lifecycle to change the default way that exceptions caught and cached by the ADF Model layer are reported to JSF. The changed implementation reduces the exceptions reported to the user to include only the exceptions that they can directly act upon, suppressing additional "wrapping" exceptions that will not make much sense to the end user.

10.5.4.2 Customizing the Page Lifecycle for a Single Page

You can customize the lifecycle of a single page setting the `ControllerClass` attribute of the <pageDefinition> to identify a class that either:

- Extends `oracle.adf.controller.v2.PageController` class
- Implements `oracle.adf.controller.v2.PagePhaseListener` interface

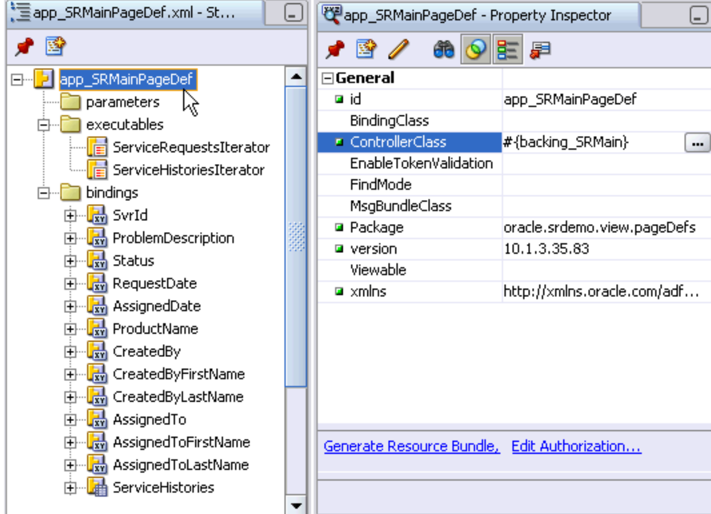
The value of the page definition's `ControllerClass` attribute can either be:

- A fully qualified class name
- An EL expression that resolves to a class that meets the requirements above

Using an EL expression for the value of the `ControllerClass`, it is possible to specify the name of a custom page controller class (or page phase listener implementation) that you've configured as a managed bean in the `faces-config.xml` file. This includes a backing bean for a JSF page, provided that it either extended `PageController` or implements `PagePhaseListener`.

Figure 10–13 illustrates how to select the root node of the page definition in the Structure window to set.

Figure 10–13 Setting the ControllerClass of a Page Definition



Note: When using an EL expression for the value of the `ControllerClass` attribute, the Structure window may show a warning, saying the "`#{`*`YourExpression`*`}`" is not a valid class. You can safely ignore this warning.

10.5.4.3 Using Custom ADF Page Lifecycle to Invoke an onPageLoad Backing Bean Method

The SRDemo application contains a `OnPageLoadBackingBeanBase` class in the `oracle.srdemo.view.util` package that implements the `PagePhaseListener` interface described above using code like what's shown in Example 10–2. The class implements the interface's `beforePhase()` and `afterPhase()` methods so that in invokes an `onPageLoad()` method before the normal ADF prepare model phase, and an `onPagePreRender()` method after the normal ADF prepare render phase.

Example 10–2 PagePhaseListener to Invoke an onPageLoad() and onPagePreRender() Method

```
// In class oracle.srdemo.view.util.OnPageLoadBackingBeanBase
/**
 * Before the ADF page lifecycle's prepareModel phase, invoke a
 * custom onPageLoad() method. Subclasses override the onPageLoad()
 * to do something interesting during this event.
 * @param event
 */
public void beforePhase(PagePhaseEvent event) {
  FacesPageLifecycleContext ctx =
    (FacesPageLifecycleContext)event.getLifecycleContext();
  if (event.getPhaseId() == Lifecycle.PREPARE_MODEL_ID) {
    bc = ctx.getBindingContainer();
    onPageLoad();
    bc = null;
  }
}
/**
 * After the ADF page lifecycle's prepareRender phase, invoke a
 * custom onPagePreRender() method. Subclasses override the onPagePreRender()
 * to do something interesting during this event.
 * @param event
 */
public void afterPhase(PagePhaseEvent event) {
  FacesPageLifecycleContext ctx =
   (FacesPageLifecycleContext)event.getLifecycleContext();
  if (event.getPhaseId() == Lifecycle.PREPARE_RENDER_ID) {
    bc = ctx.getBindingContainer();
    onPagePreRender();
    bc = null;
  }
}
public void onPageLoad() {
  // Subclasses can override this.
}
public void onPagePreRender() {
  // Subclasses can override this.
}
```

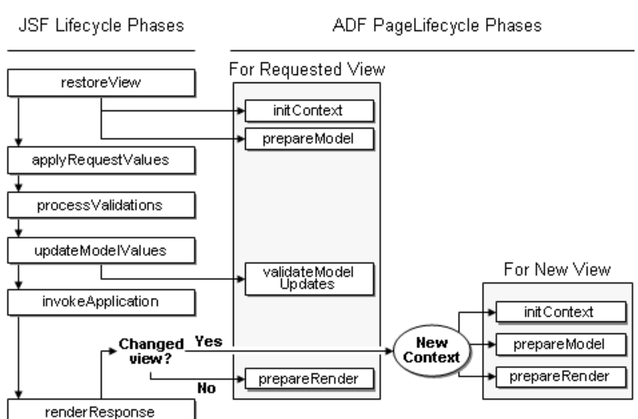
If a managed bean extends the `OnPageLoadBackingBeanBase` class, then it can be used as an ADF page phase listener because it inherits the implementation of this interface from the base class. If that backing bean then overrides either or both of the `onPageLoad()` or `onPagePreRender()` method, that method will be invoked by the ADF Page Lifecycle at the appropriate time during the page request lifecycle. The last step in getting such a backing bean to work, is to tell the ADF page definition to use the backing bean as its page controller for that page. As described above, this is done by setting the `ControllerClass` attribute on the page definition in question to an EL expression that evaluates to the backing bean.

The `SRMain` page in the SRDemo application uses the technique described in this section to illustrate writing programmatic code in the `onPageLoad()` method of the `SRMain` backing bean in the `oracle.srdemo.view.backing` package. Since that backing bean is named `backing_SRMain` in `faces-config.xml`, the `ControllerClass` property of the `SRMain` page's page definition is set to the EL expression "`#{backing_SRMain}`".

10.5.5 How to Use Refresh Correctly for InvokeAction and Iterator Bindings

Figure 10–14 illustrates how the JSF page lifecycle relates to the extended page processing phases that the ADF page lifecycle adds. You can use the `Refresh` property on iterator bindings and `invokeAction` executables in your page definition to control when each are evaluated during the ADF page lifecycle, either during the prepareModel phase, the prepareRender phase, or both. Since the term "refresh" isn't exactly second-nature, this section clarifies what it means for each kind of executable and how you should set their `Refresh` property correctly to achieve the behavior you need.

Figure 10–14 How JSF Page Lifecycle and ADF Page Lifecycle Phases Relate



10.5.5.1 Correctly Configuring the Refresh Property of Iterator Bindings

In practice, when working with iterator bindings for view object instances in an application module, you can simply leave the default setting of *Refresh*="`ifNeeded`". You may complement this with a boolean-valued EL expression in the `RefreshCondition` property to conditionally avoid refreshing the iterator if desired. However, you may still be asking yourself, "What does *refreshing* an iterator binding mean anyway?"

An iterator binding, as its name implies, is a binding that points to an iterator. For scalability reasons, at runtime the iterator bindings in the binding container release any reference they have to a row set iterator at the end of each request. During the next request, the iterator bindings are refreshed to again point at a "live" row set iterator that is tracking the current row of some data collection. The act of *refreshing* an ADF iterator binding during the ADF page lifecycle is precisely the operation of accessing the row set iterator to "reunite" the binding to the row set iterator to which it is bound.

If an iterator binding is *not* refreshed during the lifecycle, it is not pointing to any row set iterator for that request. This results in the value bindings related to that iterator binding not having any data to display. This can be a desirable result, for example, if you want a page like a search page to initially show no data. To achieve this, you can use the `RefreshCondition` of:

```
#{adfFacesContext.postback == true}
```

The `adfFacesContext.postback` boolean property evaluates to `false` when a page is first rendered, or rendered due to navigation from another page. It evaluates to `true` when the end user has interacted with some UI control on the page, causing a postback to that page to handle the event. By using this expression as the `RefreshCondition` for a particular iterator binding, it will refresh the iterator binding only when the user interacts with a control on the page.

The valid values for the `Refresh` property of an iterator binding are as follows. If you want to refresh the iterator during:

- Both the prepareModel and prepareRender phases, use **Refresh**="`ifNeeded`" (*default*)
- Just during the prepareModel phase, use **Refresh**="`prepareModel`"
- Just during the prepareRender phase, use **Refresh**="`renderModel`"

If you only want the iterator binding to refresh when your own code calls `getRowSetIterator()` on the iterator binding, set **Refresh**="`never`". Other values of `Refresh` are either not relevant to iterator bindings, or reserved for future use.

10.5.5.2 Refreshing an Iterator Binding Does Not Forcibly Re-Execute Query

It is important to understand that when working with iterator bindings related to a view object instance in an application module, *refreshing* an iterator binding does not forcibly re-execute its query each time. The first time the view object instance's row set iterator is accessed during a particular user's unit of work, this will implicitly execute the view object's query if it was not already executed. Subsequent refreshing of the iterator binding related to that view object instance on page requests that are part of the same logical unit of work will only access the row set iterator again, not forcibly re-execute the query. Should you desire re-executing the query to refresh its data, use the `Execute` or `ExecuteWithParams` built-in operation, or programmatically call the `executeQuery()` method on the iterator binding.

10.5.5.3 Correctly Configuring Refresh Property of InvokeAction Executables

Several page definitions in the SRDemo application use the declarative `invokeAction` binding to trigger either built-in operations or custom operations during this extended ADF page processing lifecycle. Each `invokeAction` has an `id` property that give the binding a name, and then three other properties of interest:

- The `Binds` property controls *what* the `invokeAction` will do if it fires

  Its value is the name of an action binding or method action binding in the same binding container.

- The `Refresh` property controls *when* the `invokeAction` will invoke the action binding

  To have it fire during the ADF page lifecycle's:

  - prepareModel phase, use **Refresh**=`prepareModel`
  - prepareRender phase, use **Refresh**=`renderModel`
  - prepareModel *and* prepareRender phases, use **Refresh**=`ifNeeded`

- The `RefreshCondition` property can be used to control *whether* it will fire at all

  Its value, if supplied, is a boolean-valued EL expression. If the expression evaluates to `true` when the `invokeAction` is considered during the page lifecycle, the related action binding is invoked. If it evaluates to `false`, then the action binding is not invoked.

Note: Other values of the `Refresh` property not described here are reserved for future use.

Notice in Figure 10–14 that the key distinction between the ADF prepareModel phase and the prepareRender phase is that one comes before JSF's invokeApplication phase, and one after. Since JSF's invokeApplication phase is when action listeners fire, if you need your `invokeAction` to trigger after these action listeners have performed their processing, you'll want to use the **Refresh**="`renderModel`" setting on it.

If the `invokeAction` binds to a *method* action binding that accepts parameters, then two additional values can be supplied for the `Refresh` property: `prepareModelIfNeeded` and `renderModelIfNeeded`. These have the same meaning as their companion settings without the `*IfNeeded` suffix, except that they perform an optimization to compare the current set of evaluated parameter values with the set that was used to invoke the method action binding the previous time. If the parameter values for the current invocation are exactly the same as the ones used previously, the `invokeAction` does not invoke its bound method action binding.

Note: The default value of the `Refresh` property is `ifNeeded`. This means that if you do not supply a `RefreshCondition` expression to further refine its firing, the related action binding will be invoked *twice* during each request. Oracle recommends either adding an appropriate `RefreshCondition` expression (if you want it evaluated during both phases) or changing the default Refresh setting for invokeAction bindings to either `prepareModel` or `renderModel`, depending on whether you want your `invokeAction` to occur before or after the JSF `invokeApplication` phase.

10.5.6 Understanding the Difference Between setCurrentRowWithKey and setCurrentRowWithKeyValue

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

Confusingly, as shown in Figure 10–15, the `setCurrentRowWithKey` and `setCurrentRowWithKeyValue` operations both expect a parameter named `rowKey`, but they differ precisely by what each expects that `rowKey` parameter value to be at runtime:

setCurrentRowWithKey

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

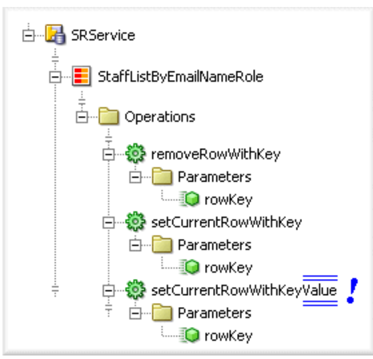
```
000200000002C20200000002C102000000010000010A5AB7DAD9
```

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

setCurrentRowWithKeyValue

`setCurrentRowWithKeyValue` expects the `rowKey` parameter value to be the literal value representing the key of the view row. For example, it's value would be simply "`201`" to find service request number `201`.

Figure 10–15 The setCurrentRowWithKeyValue Operation Expects a Literal Attribute Value as the Key



Note: If you write custom code in an application module class and need to find a Row based on a serialized string key passed from the client, you can use the `getRowFromKey()` method in the `JboUtil` class in the `oracle.jbo.client` package:

```
static public Row getRowFromKey(RowSetIterator rsi, String sKey)
```

Pass the view object instance in which you'd like to find the row as the first parameter, and the serialized string format of the Key as the second parameter.

10.5.7 Understanding Bundled Exception Mode

An application module provides a feature called bundled exception mode which allows web applications to easily present a maximal set of failed validation exceptions to the end user, instead of presenting only the *first* error that gets raised. By default, the ADF Business Components application module pool enables bundled exception mode for web applications.

You typically will not need to change this default setting. However it is important to understand that it is enabled by default since it effects how validation exceptions are thrown. Since the Java language and runtime only support throwing a single exception object, the way that bundled validation exceptions are implemented is by wrapping a set of exceptions as details of a new "parent" exception that contains them. For example, if multiple attributes in a single entity object fail attribute-level validation, then these multiple `ValidationException` objects will be wrapped in a `RowValException`. This wrapping exception contains the row key of the row that has failed validation. At transaction commit time, if multiple rows do not successfully pass the validation performed during commit, then all of the `RowValException` objects will get wrapped in an enclosing `TxnValException` object.

When writing custom error processing code, as illustrated by the overridden `reportErrors()` method in the `SRDemoPageLifecycle` class in the SRDemo application, you can use the `getDetails()` method of the `JboException` base exception class to recursively process the bundled exceptions contained inside it.

Note: All the exception classes mentioned here are in the `oracle.jbo` package.

10.6 Overview of How SRDemo Pages Use the SRService

This section provides a brief overview of how each page in the SRDemo application uses the `SRService` application module's view object instances and service methods.

10.6.1 The SRList Page

10.6.1.1 Overview of Data Binding in the SRList Page

Figure 10–16 illustrates the data binding for `SRList` page.

Iterator Bindings to View Object Instances

`ServiceRequestsByStatusIterator` for `ServiceRequestsByStatus` view object instance

Page Definition Variables

None

Action Bindings to Built-in Operations

`setCurrentRowWithKey`, `ExecuteWithParams` related to the `ServiceRequestsByStatusIterator` iterator binding

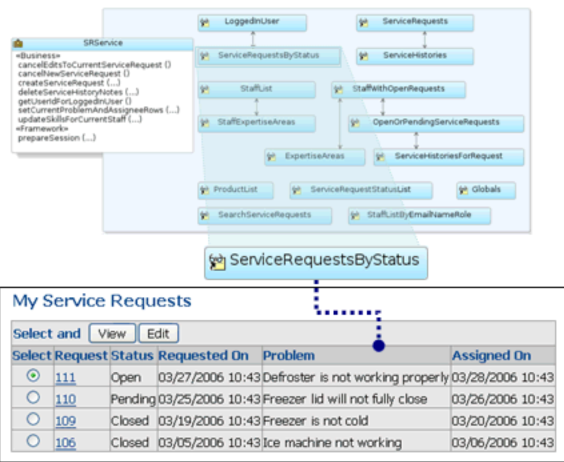
Method Action Bindings to Custom Operations

None

InvokeActions Customizing Page Lifecycle

None

Figure 10–16 View Object for SRList Page



10.6.1.2 Business Service Notes for the SRList Page

View Object Instances

ServiceRequestsByStatus is an instance of the entity-based view object ServiceRequestsByStatus, which extends the ServiceRequests view object and adds a named bind variable called StatusCode.

Application Module Custom Methods

None

10.6.2 The SRMain Page

10.6.2.1 Overview of Data Binding in the SRMain Page

Figure 10–17 illustrates the data binding for the SRMain page.

Iterator Bindings to View Object Instances

- ServiceRequestsIterator for ServiceRequests view object instance
- ServiceHistoriesIterator for ServiceHistories view object instance

Page Definition Variables

None

Action Bindings to Built-in Operations

- setCurrentRowWithKey, Delete related to the ServiceRequestsIterator iterator binding
- Create, DeleteNewHistory (for Delete built-in) related to the ServiceHistoriesIterator iterator binding
- Commit related to the SRService data control

Method Action Bindings to Custom Operations

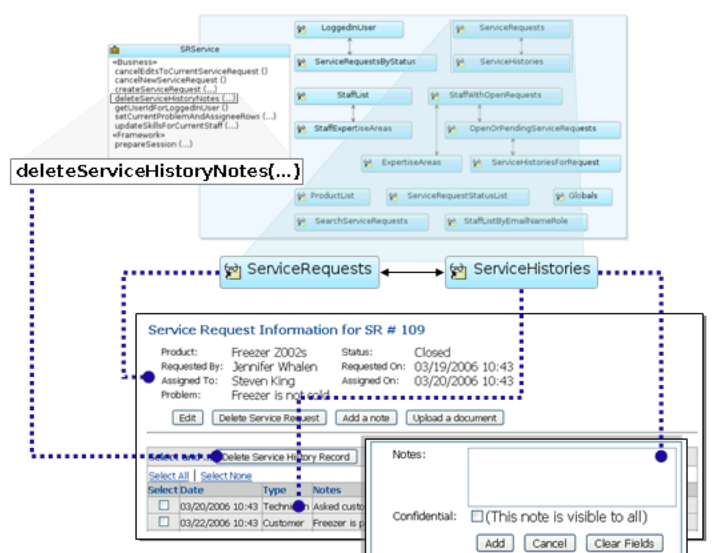
deleteServiceHistoryNotes invokes deleteServiceHistoryNotes() method on the SRService client interface

InvokeActions Customizing Page Lifecycle

None

Note: The `SRMain` backing bean for the `SRMain` page (in the `oracle.srdemo.view.backing` package) is using the technique described in Section 10.5.4.3, "Using Custom ADF Page Lifecycle to Invoke an onPageLoad Backing Bean Method" to programmatically accomplish exactly the same thing that the `SREdit` page is doing declaratively.

Figure 10–17 Service Method and View Objects for the SRManage Page



10.6.2.2 Business Service Notes for the SRMain Page

View Object Instances

- `ServiceRequests` is an instance of the entity-based view object `ServiceRequests`. It joins data from the main `ServiceRequest` entity usage and three additional reference entity usages: `CreatedByUser` (`User` entity object), `AssignedToUser` (`User` entity object), and `Product`. The `ServiceRequests` view object is linked master/detail to the `ServiceHistories` view object.

- `ServiceHistories` is an instance of the entity-based view object `ServiceHistories`. It joins data from the main `ServiceHistory` entity usage and an additional reference entity usage `SystemUser` (`User` entity object). It is an XML-only view object, with no custom Java class.

Application Module Custom Methods

As shown in Example 10–3, the `deleteServiceHistoryNotes()` method deletes service history note rows corresponding to the `Key` objects in the key set passed as an argument.

Example 10–3 DeleteServiceHistoryNotes Method in SRServiceImpl.java

```
public void deleteServiceHistoryNotes(Set keySet) {
  if (keySet != null) {
    ViewObject histories = getServiceHistories();
    for (Key k: (Set<Key>)keySet) {
      Row[] rowToDelete = histories.findByKey(k, 1);
      if (rowToDelete == null || rowToDelete.length == 0) {
        throw new JboException("Failed to find row with serialized key" +
                               k.toStringFormat(false));
      }
      rowToDelete[0].remove();
      getDBTransaction().commit();
    }
  }
}
```

10.6.3 The SREdit Page

10.6.3.1 Overview of Data Binding in the SREdit Page

Figure 10–18 illustrates the data binding in the `SREdit` page.

Iterator Bindings to View Object Instances

- `ServiceRequestsIterator` for `ServiceRequests` view object instance
- `ServiceRequestStatusListIterator` for `ServiceRequestStatusList` view object instance

Page Definition Variables

None

Action Bindings to Built-in Operations

- `setCurrentRowWithKey` related to the `ServiceRequestsIterator` iterator binding
- `Commit` related to the `SRService` data control

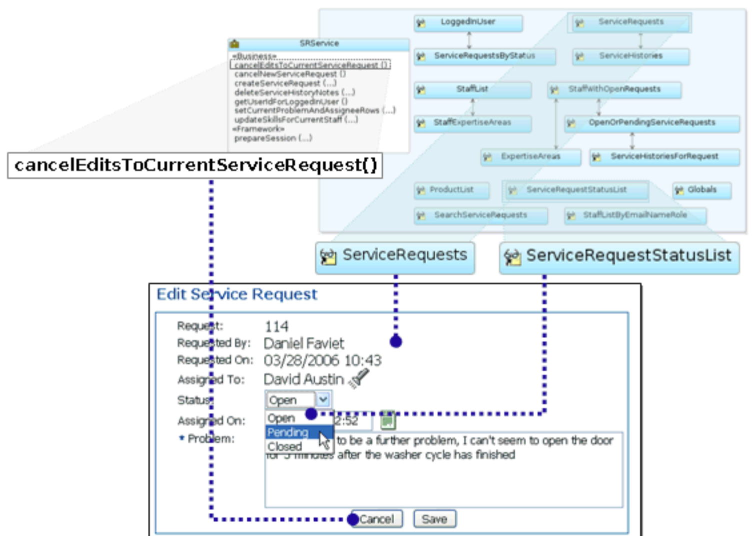
Method Action Bindings to Custom Operations

`cancelEditsToCurrentServiceRequest` invokes `cancelEditsToCurrentServiceRequest()` on the `SRService` client interface

InvokeActions Customizing Page Lifecycle

`setRowToEditFromRequestParameter` invokes the built-in `setCurrentRowWithKey` operation in prepare model phase (**Refresh**="`prepareModel`") when first navigating to the page (i.e. not processing a postback) and `processScope.rowKeyStr` attribute is not set (*RefreshCondition*="`${adfFacesContext.postback == false and not empty processScope.rowKeyStr}`").

Figure 10–18 Service Method and View Objects for the SREdit Page



10.6.3.2 Business Service Notes for the SREdit Page

View Object Instances

- `ServiceRequests` is an instance of the entity-based view object `ServiceRequests`. It joins data from the main `ServiceRequest` entity usage and three additional reference entity usages: `CreatedByUser` (`User` entity object), `AssignedToUser` (`User` entity object), and `Product`.
- `ServiceRequestStatusList` is an instance of the read-only view object `ServiceRequestStatusList`. Its data is provided by a static list supplied in the `ServiceRequestStatusListImpl.java` class. This class extends a `SRStaticDataViewObjectImpl` in the `FrameworkExtensions` project which provides the basic support for implementing a view object based on static data.

Application Module Custom Methods

As shown in Example 10–4, the `cancelEditsToCurrentServiceRequest()` method uses the `refresh()` method to cancel edits made in the current transaction to the current service request row.

Example 10–4 CancelEditsToCurrentServiceRequest Method in SRServiceImpl.java

```
public void cancelEditsToCurrentServiceRequest() {
  Row svReq = getServiceRequests().getCurrentRow();
  if (svReq != null) {
    svReq.refresh(Row.REFRESH_WITH_DB_FORGET_CHANGES);
  }
}
```

10.6.4 The SRSearch Page

10.6.4.1 Overview of Data Binding in the SRSearch Page

Figure 10–19 illustrates the data binding in the `SRSearch` page.

Iterator Bindings to View Object Instances

- `SearchServiceRequestsIterator` for `SearchServiceRequests` view object instance (Forced to stay in find mode by `AlwaysFind` invokeAction).
- `SearchServiceRequestsResultsIterator` for `SearchServiceRequests` view object instance
- `ServiceRequestStatusListIterator` for `ServiceRequestStatusList` view object instance

Page Definition Variables

None

Action Bindings to Built-in Operations

- `Execute`, `Delete`, `Create` related to the `SearchServiceRequestsIterator` iterator binding
- `Find`, `First`, `Next`, `Previous`, `Last`, `setCurrentRowWithKey` related to the `SearchServiceRequestsResultsIterator` iterator binding

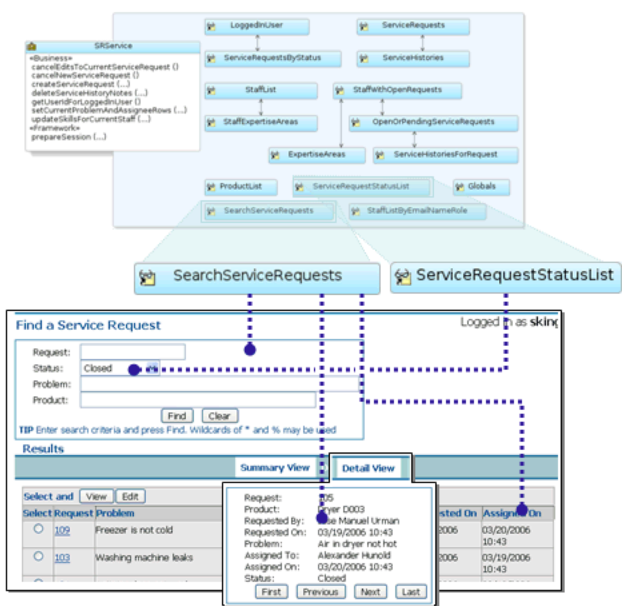
Method Action Bindings to Custom Operations

None

InvokeActions Customizing Page Lifecycle

- `AlwaysFind` invokes the built-in `Find` operation (for the `SearchServiceRequestsIterator` iterator binding) in either prepare model or render model phases (*Refresh*="`ifNeeded`") when the `SearchServiceRequestsIterator` is not in find mode (`${bindings.SearchServiceRequestsIterator.findMode == false}`)
- `insertBlankViewCriteriaRowIfThereAreNone` invokes the built-in `Create` operation in either prepare model or render model phases (*Refresh*="`ifNeeded`") when the `SearchServiceRequestsIterator` is not in find mode (`${bindings.SearchServiceRequestsIterator.findMode == false}`)

Figure 10–19 View Object for the SRSearch Page



10.6.4.2 Business Service Notes for the SRSearch Page

View Object Instances

- `SearchServiceRequests` is an instance of the entity-based view object `ServiceRequests`. It joins data from the main `ServiceRequest` entity usage and three additional reference entity usages: `CreatedByUser` (`User` entity object), `AssignedToUser` (`User` entity object), and `Product`.
- `ServiceRequestStatusList` is an instance of the read-only view object `ServiceRequestStatusList`. Its data is provided by a static list supplied in the `ServiceRequestStatusListImpl.java` class. This class extends a `SRStaticDataViewObjectImpl` in the `FrameworkExtensions` project which provides the basic support for implementing a view object based on static data.

Application Module Custom Methods

None

10.6.5 The SRStaffSearch Page

10.6.5.1 Overview of Data Binding in the SRStaffSearch Page

Figure 10–20 illustrates the data binding for the `SRStaffSearch` page.

Iterator Bindings to View Object Instances

`StaffListByEmailNameRoleIterator` for `StaffListByEmailNameRole` view object instance

Page Definition Variables

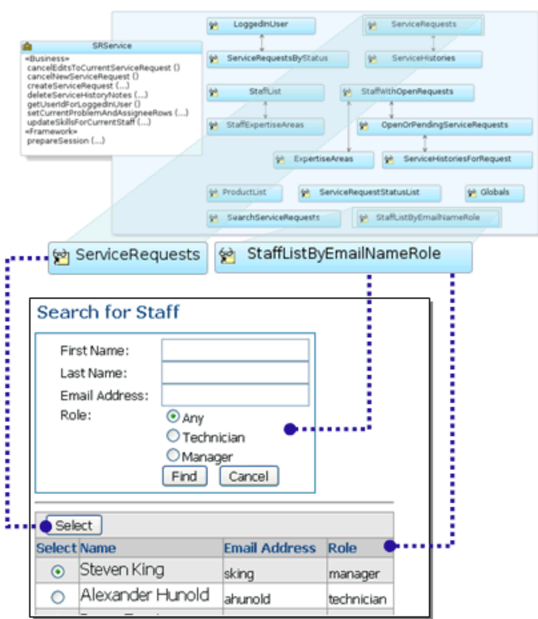
`StaffListByEmailNameRole_Role`, `StaffListByEmailNameRole_EmailAddress`, `StaffListByEmailNameRole_TheFirstName`, `StaffListByEmailNameRole_TheLastName`

Action Bindings to Built-in Operations
`ExecuteWithParams` related to `StaffListByEmailNameRoleIterator` iterator binding

Method Action Bindings to Custom Operations
None

InvokeActions Customizing Page Lifecycle
None

Figure 10–20 View Objects for the SRStaffSearch Page



10.6.5.2 Business Service Notes for the SRStaffSearch Page

View Object Instances
`StaffListByEmailNameRole` is an instance of the entity-based view object `StaffListByEmailNameRole`, which extends the `StaffList` view object and adds name bind variables `EmailAddress`, `Role`, `TheFirstName`, and `TheLastName`.

Application Module Custom Methods
None

10.6.6 The SRManage Page

10.6.6.1 Overview of Data Binding in the SRManage Page

Figure 10–21 illustrates the data binding in the `SRManage` page.

Iterator Bindings to View Object Instances

- `StaffWithOpenRequestsIterator` for `StaffWithOpenRequests` view object instance
- `ExpertiseAreasIterator` for `ExpertiseAreas` view object instance
- `OpenOrPendingServiceRequestsIterator` for `OpenOrPendingServiceRequests` view object instance
- `ServiceHistoriesForRequestIterator` for `ServiceHistoriesForRequest` view object instance

Page Definition Variables

None

Action Bindings to Built-in Operations

`setCurrentStaffRowWithKey` (for `setCurrentRowWithKey` built-in) related to `StaffWithOpenRequestsIterator` iterator binding

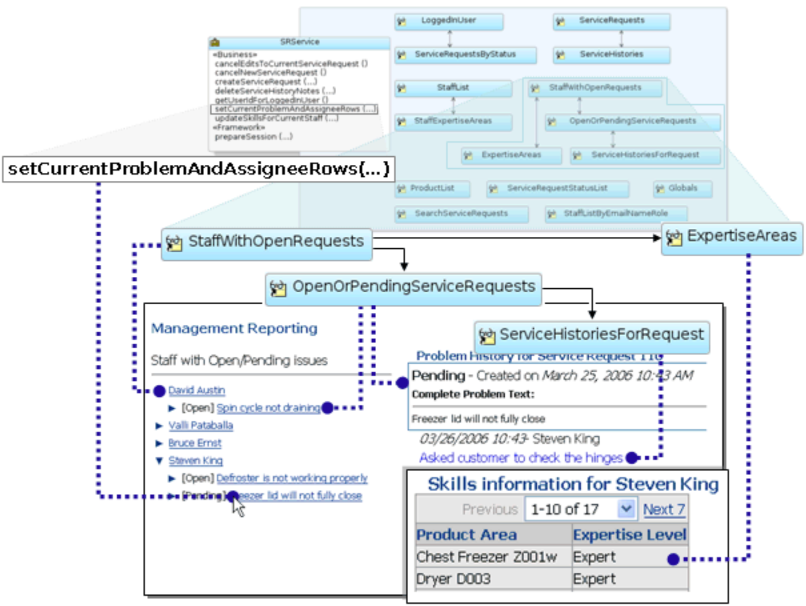
Method Action Bindings to Custom Operations

`setCurrentProblemAndAssigneeRows` invokes `setCurrentProblemAndAssigneeRows()` on the `SRService` client interface

InvokeActions Customizing Page Lifecycle

None

Figure 10–21 Service Method and View Objects for the SRManage Page



10.6.6.2 Business Service Notes for the SRManage Page

View Object Instances

- StaffWithOpenRequests is an instance of the read-only view object StaffWithOpenRequests. It queries information from the USERS and SERVICE_REQUESTS tables. It is an XML only view object, with no related Java class. This view object is linked master/detail with the ExpertiseAreas and OpenOrPendingServiceRequests view objects.
- ExpertiseAreas is an instance of the entity-based view object ExpertiseAreas. It joins information from the primary ExpertiseArea entity usage and a reference Product entity usage. It is an XML only view object, with no related Java class.
- OpenOrPendingServiceRequests is an instance of the read-only view object OpenOrPendingServiceRequests. It queries information from the SERVICE_REQUESTS table. It is an XML only view object, with no related Java class. This view object is linked master/detail with the ServiceHistories view object.
- ServiceHistoriesForRequest is an instance of the entity-based view object ServiceHistories. It joins data from the main ServiceHistory entity usage and an additional reference entity usage SystemUser (User entity object). It is an XML-only view object, with no custom Java class.

Application Module Custom Methods

As shown in Example 10–5, the setCurrentProblemAndAssigneeRows() method uses a helper method to set the current row in the StaffWithOpenRequests view object instance and the OpenOrPendingServiceRequests view object instance based on the serialized string keys passed in.

Example 10–5 SetCurrentProblemAndAssigneeRows Method in SRServiceImpl.java

```
public void setCurrentProblemAndAssigneeRows(String requestKeyStr,
                                             String staffKeyStr) {
  setRowWithKeyString(getStaffWithOpenRequests(), staffKeyStr);
  setRowWithKeyString(getOpenOrPendingServiceRequests(), requestKeyStr);
}
```

10.6.7 The SRSkills Page

10.6.7.1 Overview of Data Binding in the SRSkills Page

Figure 10–21 illustrates the data binding for the SRSkills page.

Iterator Bindings to View Object Instances

- StaffListIterator for StaffList view object instance
- StaffExpertiseAreasIterator for StaffExpertiseAreas view object instance
- ProductListIterator for ProductList view object instance

Page Definition Variables

None

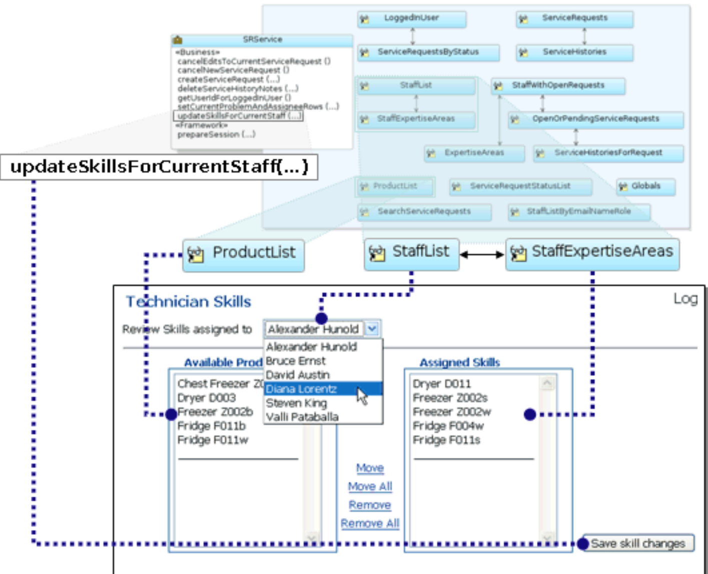
Action Bindings to Built-in Operations

None

Method Action Bindings to Custom Operations
`updateSkillsForCurrentStaff` invokes the `updateSkillsForCurrentStaff()` method on the `SRService` client interface

InvokeActions Customizing Page Lifecycle
None

Figure 10–22 Service Method and View Objects for the SRSkills Page



10.6.7.2 Business Service Notes for the SRSkills Page

View Object Instances

- `StaffList` is an instance of the entity-based view object `StaffList`. It queries information from the primary entity usage `SystemUser` (`User` entity object). This view object is linked master/detail with the `ExpertiseAreas` view object.
- `StaffExpertiseAreas` is an instance of the entity-based view object `ExpertiseAreas`. It joins information from the primary `ExpertiseArea` entity usage and a reference `Product` entity usage. It is an XML only view object, with no related Java class.
- `ProductList` is an instance of the entity-based view object `ProductList`. It queries data from the primary entity usage `Products` (`Product` entity object). It is an XML only view object, with no related Java class.

Application Module Custom Methods
As shown in Example 10–6, the `updateSkillsForCurrentStaff()` method performs the following steps:

1. Clones the list of product ids
2. Creates a secondary `RowSetIterator` to do programmatic iteration over the `ExpertiseAreas`
3. Removes rows for current user for products that are not in the list of products ids
4. Closes the secondary row set iterator when done iterating

5. Adds new rows for the keys that are left
6. Commits the transaction.

Example 10–6 UpdateSkillsForCurrentStaff Method in SRServiceImpl.java

```
public void updateSkillsForCurrentStaff(List productIds) {
  if (productIds != null && productIds.size() > 0) {
    // 1. Cone the list of product ids
    List<Number> copyOfProductIds = (List<Number>)Utils.cloneList(productIds);
    ViewObject skills = getStaffExpertiseAreas();
    // 2. Create a secondary rowset iterator for iteration
    RowSetIterator rsi = skills.createRowSetIterator(null);
    // 3. Remove rows for current user for products not in list of products
    while (rsi.hasNext()) {
      Row r = rsi.next();
      Number productId = (Number)r.getAttribute("ProdId");
      // if the existing row is in the list, we're ok, so remove from list.
      if (copyOfProductIds.contains(productId)) {
        copyOfProductIds.remove(productId);
      }
      // if the existing row is in not list, remove it.
      else {
        r.remove();
      }
    }
    // 4. Close the secondary row set iterator when we're done
    rsi.closeRowSetIterator();
    // 5. Add new rows for the keys that are left
    for (Number productIdToAdd: copyOfProductIds) {
      Row newRow = skills.createRow();
      skills.insertRow(newRow);
      newRow.setAttribute("ProdId", productIdToAdd);
    }
    // 6. Commit the transaction
    getDBTransaction().commit();
  }
}
```

Note: Since the iterator bindings in the ADF Model layer bind by default to the default row set iterator for the default rowset of the view object instance to which they are related, it's best practice to create a *secondary row set iterator* to perform programmatic iteration of a view object's row set in your application business logic. This way you do not affect the current row that the user sees in the user interface. The secondary iterator can have a developer-assigned name, but if you pass `null` the system assigns it a name. Since you'll typically always be closing it as soon as you're done iterating, using the system-assigned name is fine.

10.6.8 The SRCreate Page

10.6.8.1 Overview of Data Binding in the SRCreate Page

Figure 10–23 illustrates the data binding for the `SRCreate` page.

Iterator Bindings to View Object Instances

- `GlobalsIterator` for `Globals` view object instance
- `ProductListIterator` for `ProductList` view object instance

Page Definition Variables

None

Action Bindings to Built-in Operations

None

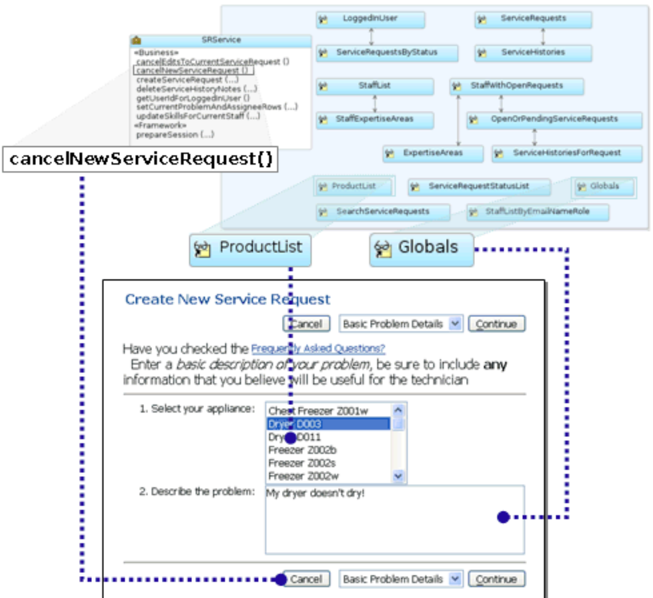
Method Action Bindings to Custom Operations

`cancelNewServiceRequest` invokes the `cancelNewServiceRequest` method on the `SRService` client interface

InvokeActions Customizing Page Lifecycle

`clearServiceRequestFieldsIfNotInTrain` invokes the `cancelNewServiceRequest` method action binding during prepare model phase (**Refresh**="`prepareModel`") when the page is not handling postback events and the `requestScope.processChoice` attribute is not set (**RefreshCondition**="`${adfFacesContext.postback == false and empty requestScope.processChoice}`").

Figure 10–23 Service Method and View Objects for the SRCreate Page



10.6.8.2 Business Service Notes for the SRCreate Page

View Object Instances

- ProductList is an instance of the entity-based view object ProductList. It queries data from the primary entity usage Products (Product entity object). It is an XML only view object, with no related Java class.
- Globals is an instance of the transient view object Globals. It contains transient attributes to temporarily store information about ProductId, ProductName, and ProductDescription across pages.

> **Note:** A *transient view object* is one that has no SQL query and all transient attributes. It can contain rows that are populated either programmatically by your application module business logic or declaratively using action bindings for built-in operations in the ADF Model layer. For users familiar with Oracle Forms, this is similar to what you know as a "non database block" in Forms.

Application Module Custom Methods

As shown in Example 10–7, the cancelNewServiceRequest() method ensures there is a single blank row in the Globals view object instance.

Example 10–7 CancelNewServiceRequest Method in SRServiceImpl.java

```
public void cancelNewServiceRequest() {
  ViewObject globals = getGlobals();
  globals.clearCache();
  globals.insertRow(globals.createRow());
}
```

10.6.9 The SRConfirmCreate Page

10.6.9.1 Overview of Data Binding in the SRConfirmCreate Page

Figure 10–24 illustrates the data binding for the SRConfirmCreate page.

Iterator Bindings to View Object Instances

- GlobalsIterator for Globals view object instance
- LoggedInUserIterator for LoggedInUser view object instance

Page Definition Variables

None

Action Bindings to Built-in Operations

None

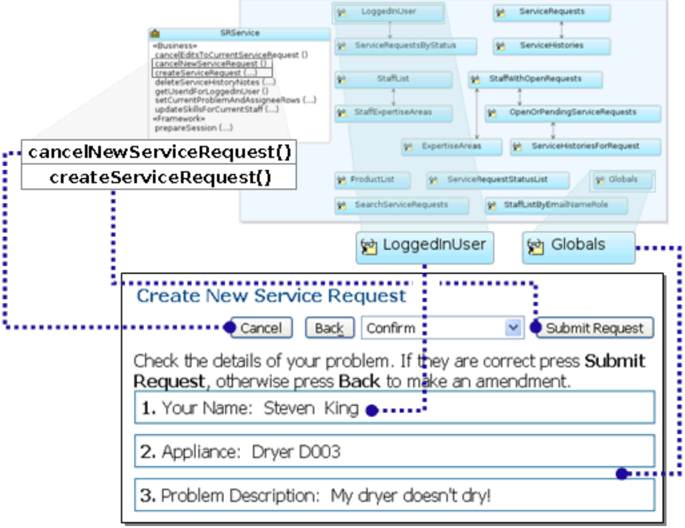
Method Action Bindings to Custom Operations

- cancelNewServiceRequest invokes the cancelNewServiceRequest() on the SRService client interface
- createServiceRequest invokes the createServiceRequest() on the SRService client interface

InvokeActions Customizing Page Lifecycle

None

Figure 10–24 Service Method and View Objects for the SRConfirmCreate Page



10.6.9.2 Business Service Notes for the SRCreate Page

View Object Instances

- `Globals` is an instance of the transient view object `Globals`. It contains transient attributes to temporarily store information about `ProductId`, `ProductName`, and `ProductDescription` across pages.
- `LoggedInUser` is an instance of the entity-based view object `LoggedInUser`. It queries data from the primary entity usage `Users` (`User` entity object). The `LoggedInUser` view object is linked master/detail with the `ServiceRequestsByStatus` view object.

Application Module Custom Methods

As shown in Example 10–8, the `createServiceRequest()` method performs the following basic steps:

1. Gets the entity definition for `ServiceRequest`
2. Creates a new `ServiceRequest` entity row (`ServiceRequestImpl` class)
3. Accesses the current row in `Globals` as a strongly-typed `GlobalsRowImpl`
4. Set the problem description and product ID for new service request
5. Commits the transaction
6. Returns an Integer representing the database-trigger assigned SR number for the new service request.

Example 10–8 CreateServiceRequest Method in SRServiceImpl.java

```
public Integer createServiceRequest() {
  // 1. Get the entity definition for ServiceRequest
  EntityDefImpl svcReqDef = ServiceRequestImpl.getDefinitionObject();
  // 2. Create a new ServiceRequest entity row
  ServiceRequestImpl newReq =
    (ServiceRequestImpl)svcReqDef.createInstance2(getDBTransaction(),null);
  // 3. Access the current row in Globals as a strongly-typed GlobalsRowImpl
  GlobalsRowImpl globalsRow = (GlobalsRowImpl)getGlobals().getCurrentRow();
  // 4. Set the problem description and product id for new service request
  newReq.setProblemDescription(globalsRow.getProblemDescription());
  newReq.setProdId(globalsRow.getProductId());
  // 5. Commit the transaction
  getDBTransaction().commit();
  // 6. Return an integer representing the database-assigned SR Number
  return newReq.getSvrId().getSequenceNumber().intValue();
}
```

Part III

Building Your Web Interface

Part III contains the following chapters:

- Chapter 11, "Getting Started with ADF Faces"
- Chapter 12, "Displaying Data on a Page"
- Chapter 13, "Creating a Basic Page"
- Chapter 14, "Adding Tables"
- Chapter 15, "Displaying Master-Detail Data"
- Chapter 16, "Adding Page Navigation"
- Chapter 17, "Creating More Complex Pages"
- Chapter 18, "Creating a Search Form"
- Chapter 19, "Using Complex UI Components"
- Chapter 20, "Using Validation and Conversion"
- Chapter 21, "Adding ADF Bindings to Existing Pages"
- Chapter 22, "Changing the Appearance of Your Application"
- Chapter 23, "Optimizing Application Performance with Caching"
- Chapter 24, "Testing and Debugging Web Applications"

11 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 11.1, "Introduction to ADF Faces"
- Section 11.2, "Setting Up a Workspace and Project"
- Section 11.3, "Creating a Web Page"
- Section 11.4, "Laying Out a Web Page"
- Section 11.5, "Creating and Using a Backing Bean for a Web Page"
- Section 11.6, "Best Practices for ADF Faces"

11.1 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 1.3, "Declarative Development with Oracle ADF and JavaServer Faces".

Table 11–1 shows the platforms currently supported for ADF Faces.

Table 11–1 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

11.2 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, ADF BC]** 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, ADF BC] 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.

11.2.1 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 ADF Business Components. 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 11–1 shows the Application Navigator view of the ViewController project after you create the workspace.

Figure 11–1 ViewController Project in the Navigator After You Create a Workspace

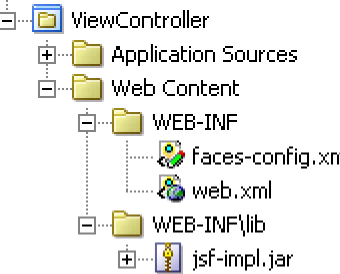
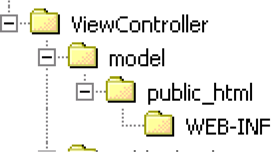


Figure 11–2 shows the actual folders JDeveloper creates in the `<JDEV_HOME>/jdev/mywork` folder in the file system.

Figure 11–2 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, ADF BC]** 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 11–2.
  - **Libraries**: JSF, Commons Beanutils, Commons Digester, Commons Logging, Commons Collections, JSTL.
  - **Technology Scope**: JSF, JSP and Servlets, Java, HTML, XML.

  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 11.2.1.1, "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 11.2.1.2, "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 11.3.2, "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 ADF Business Components technology. For information about creating a reusable layer of entity objects in the Model project, see Section 2.5, "Creating a Layer of Business Domain Objects for Tables". For information about building application modules and view objects, see Section 2.6, "Building the Business Service to Handle the Use Case".

11.2.1.1 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 11–1 shows the starter `web.xml` file JDeveloper first creates for you.

Example 11–1 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/*`. For example, if your web page is `index.jsp` or `index.jspx`, this means that when the URL `http://localhost:8080/SRDemoADFBC/faces/index.jsp` or `http://localhost:8080/SRDemoADFBC/faces/index.jspx` is issued, the URL activates the JSF servlet, which strips off the `faces` prefix and loads the file `/SRDemoADFBC/index.jsp` or `/SRDemoADFBC/index.jspx`.

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.7, "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 12.4, "Configuring the ADF Binding Filter".

11.2.1.2 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 11–2 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 11.2.3, "What You May Need to Know About Multiple JSF Configuration Files".

Example 11–2 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 16, "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.9, "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 12.2.3, "What Happens When You Use the Data Control Palette".

11.2.2 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**.

11.2.3 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 11–3 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 11–3 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.

11.3 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 11.3.1, "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.

11.3.1 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.

11.3.2 What Happens When You Create a JSF Page

Figure 11–3 shows the Application Navigator view of the ViewController project after you complete the wizard steps to add a JSF page.

Figure 11–3 ViewController Project in the Navigator After You Add a JSF Page

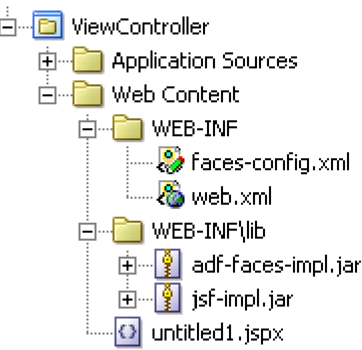
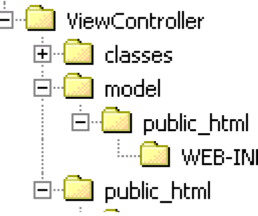


Figure 11–4 shows the actual folders JDeveloper creates in the `<JDEV_HOME>/jdev/mywork` folder in the file system.

Figure 11–4 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 11–2.
  - **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 11–4 shows a starter page for a JSF JSP document.

Example 11–4 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>
```

11.3.3 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 16, "Adding Page Navigation".

11.3.4 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 11–2 shows the URIs and default prefixes for the ADF Faces and JSF tag libraries used in JDeveloper.

Table 11–2 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 23, "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 34, "Deploying ADF Applications".

11.4 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 11–5 shows an example of a page created by using a `panelPage` component.

Figure 11–5 Page Layout Created with a PanelPage Component

ACME Corporation
Service Requests Portal

Logout Help

My Service Requests | Advanced Search | New Service Request | Management

Open Requests | Requests Awaiting Customer | Closed Requests | All Requests | Create New Service Request

Logged in as **sking**

My Service Requests

Select and [View] [Edit]

Select	Request Id	Status	Requested On	Problem	Assigned On
(•)	111	Open	Dec 17, 2005	Defroster is not working properly	Dec 18, 2005
( )	200	Open	Dec 19, 2005	Seal not working	Not assigned yet
( )	201	Open	Dec 20, 2005	Dryer is spitting out lots of lint	Dec 21, 2005
( )	202	Open	Dec 21, 2005	Leaking at the sides	Dec 21, 2005

© Oracle Corp, 2005
Contact Us About this sample

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

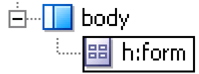
When you make changes to a page in one of the design tools, the other tools are automatically updated with the changes you made.

11.4.1 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.

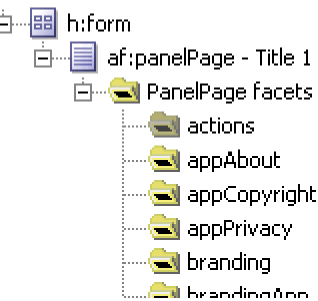
To insert UI components into a JSF page:

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 11.4.4, "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 11.4.3.1, "Editing in the Structure Window" for additional information about inserting components using the Structure window.

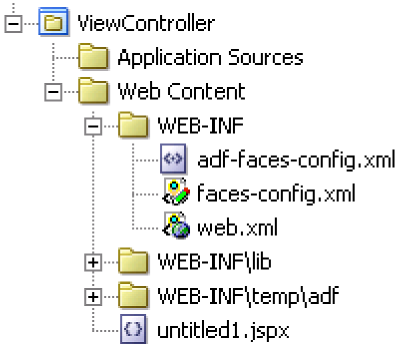
5. To edit the attributes for an inserted component, double-click the component in the Structure window to open a property editor, or select the component and then use the Property Inspector.

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 12, "Displaying Data on a Page".

11.4.2 What Happens When You First Insert an ADF Faces Component

Figure 11–6 shows the Application Navigator view of the ViewController project after adding your first ADF Faces component in a page.

Figure 11–6 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 11–4.
- Replaces the `html`, `head`, and `body` tags with `afh:html`, `afh:head`, and `afh:body`, respectively. See Example 11–5.
- Adds the ADF Faces filter and mapping configuration settings to `web.xml`. See Section 11.4.2.1, "More About the web.xml File".
- Adds the ADF Faces default render kit configuration setting to `faces-config.xml`. See Section 11.4.2.2, "More About the faces-config.xml File".
- Creates a starter `adf-faces-config.xml` in `/WEB-INF` of the ViewController project. See Section 11.4.2.3, "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 11–5 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>
```

11.4.2.1 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 11–6 shows the `web.xml` file after you add the first ADF Faces component.

Example 11–6 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.7.1, "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.

11.4.2.2 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 11–7.

Example 11–7 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>
```

11.4.2.3 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 11–8 shows the starter `adf-faces-config.xml` file.

Example 11–8 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 22.3.1, "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.10, "adf-faces-config.xml".

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

- Level of page accessibility support (See Section 11.6, "Best Practices for ADF Faces")
- Skin family (See Section 22.3, "Using Skins to Change the Look and Feel")
- Time zone (See Section 22.4.2, "How to Configure Optional Localization Properties for ADF Faces")
- Enhanced debugging output (See Section A.10.1.3, "Configuring For Enhanced Debugging Output")
- Oracle Help for the Web (OHW) URL (See Section A.10.1.11, "Configuring the Help Site URL")

You can also register a custom file upload processor for uploading files. For information, see Section 19.6.5, "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.10.1.12, "Retrieving Configuration Property Values From adf-faces-config.xml".

11.4.3 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 22.4, "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 19.4, "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 11.6, "Best Practices for ADF Faces".

11.4.3.1 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.

11.4.3.2 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 20.7, "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 11–9), or specify the error page using the JSP page directive (as shown in Example 11–10).

Example 11–9 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 11–10 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"/>
  ```

11.4.4 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 11–5.

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 19.2, "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 11–7 and Figure 11–8.

For instructions on how to insert child components into facets or into panelPage itself, see Section 11.4.1, "How to Add UI Components to a JSF Page".

11.4.4.1 PanelPage Facets

Figure 11–7 shows `panelPage` facets (numbered 1 to 12) for laying out branding images, global buttons, menu tabs, bars, and lists, and application-level text.

Figure 11–7 Basic Page Layout with Branding Images, Navigation Menus, and Application-Level Text

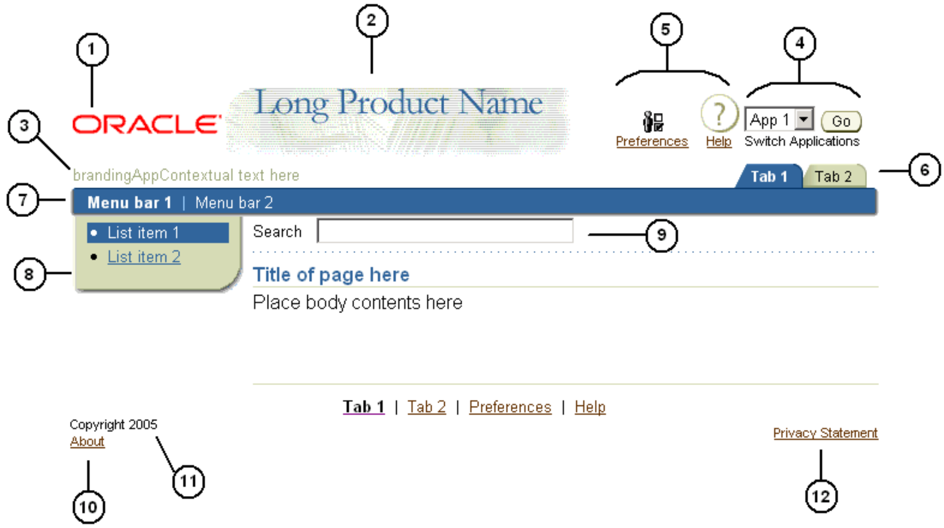


Table 11–3 shows the `panelPage` facets (as numbered in Figure 11–7), and the preferred children components that you could use in them. In JDeveloper, when you right-click a facet in the Structure window, the **Insert inside** context menu shows the preferred component to use, if any.

Table 11–3 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 11–3 (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 11–8 shows `panelPage` facets (numbered 1 to 7) for laying out page-level actions and text.

Figure 11–8 Basic Page Layout with Page-Level Actions and Informational Text

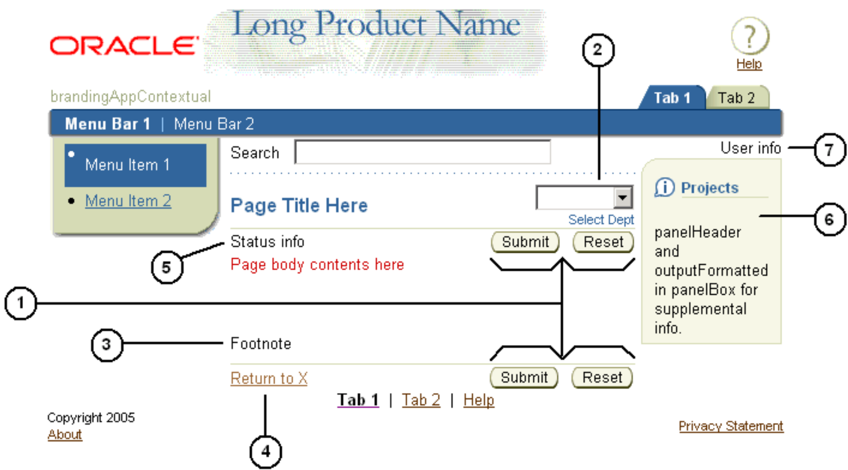


Table 11–4 shows the `panelPage` facets (as numbered in Figure 11–8), and the preferred children components that you could use in them.

Table 11–4 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 11–4 (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.

11.4.4.2 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 11–5 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 11–5 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 11–5 (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

11.5 Creating and Using a Backing Bean for a Web Page

In JSF, backing beans are JavaBeans used mainly to provide UI logic for handling events and page flow. Typically you have one backing bean per JSF page. The backing bean contains any 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.

If you don't perform any UI component manipulation or conditional page flow logic, then you might not even need to use a backing bean for a page. The rest of this section provides information about using backing beans, if you need to use them for your pages.

11.5.1 How to Create and Configure a Backing 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`.

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:

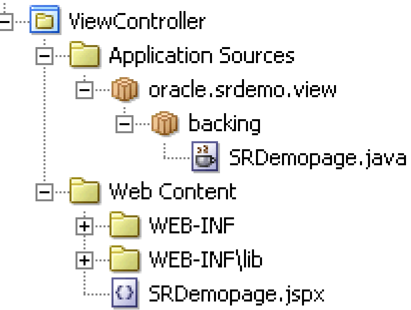
1. In the Application Navigator, double-click **faces-config.xml** to open it in the default mode of the JSF Configuration Editor.
2. At the bottom of the editor, select the **Overview** tab to switch to the Overview mode, if necessary.
3. In the element list on the left, select **Managed Beans**.
4. 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 11.5.3, "How to Use a Backing Bean in a JSF Page".

11.5.2 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 11–9.

Figure 11–9 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 11–11.

Example 11–11 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 11–12.

Example 11–12 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 11.5.7, "Using ADF Data Controls and Backing Beans".

11.5.3 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.

11.5.4 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.

11.5.5 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.

11.5.6 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 11–13.

Example 11–13 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 11–13, 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 11–14 shows an example of a managed bean that is a List.

Example 11–14 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.

11.5.7 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 12.2.3, "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 11–15 shows the `bindings` managed property in the `backing_SRMain` managed bean for the SRMain page.

Example 11–15 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 11–16 shows the part of `SRMain.java` that contains the relevant code for `bindings`.

Example 11–16 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 12.6, "Creating ADF Data Binding EL Expressions".

For an overview of how you might use JSF backing beans with ADF Model and ADF Business Components, see Chapter 1, "Introduction to Oracle ADF Applications".

11.6 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 19.4, "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 34, "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 23, "Optimizing Application Performance with Caching".

12 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 12.1, "Introduction to Displaying Data on a Page"
- Section 12.2, "Using the Data Control Palette"
- Section 12.3, "Working with the DataBindings.cpx File"
- Section 12.4, "Configuring the ADF Binding Filter"
- Section 12.5, "Working with Page Definition Files"
- Section 12.6, "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.

12.1 Introduction to Displaying Data on a Page

The ADF data controls provide an abstraction of an application's business services, giving the ADF binding layer access to the service data. An application module data control represents a specific application module and exposes instances of that application module's view objects. Each application module is automatically available as a data control. You can bind UI components to application module data controls to populate a page with data from your active data model at runtime. For more information about application module data controls, see Chapter 10, "Overview of Application Module Data Binding".

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.

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

12.2 Using the Data Control Palette

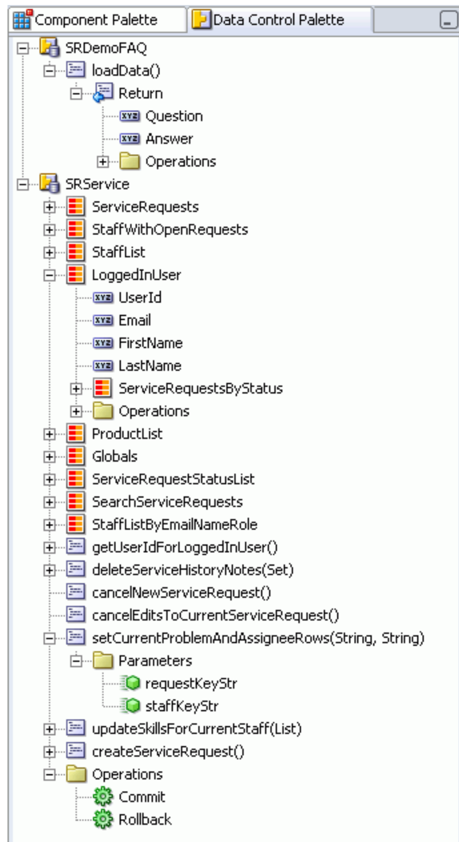
You can design a databound user interface by dragging an item from the Data Control Palette and dropping it on a page as a specific UI component. When you use the Data Control Palette to create a UI component, JDeveloper automatically creates the various code and objects needed to bind the component 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 12–1 shows the Data Control Palette for the SRDemo application, which uses ADF Business Components as the business service.

Note: If your Data Control Palette is empty, be sure that you have created view objects for the data you want to access and have added those view objects to an application module. For information about creating view objects and application modules, see Chapter 5, "Querying Data Using View Objects".

Figure 12–1 Data Control Palette



12.2.1 How to Understand the Items on the Data Control Palette

The Data Control Palette lists all the data controls that have been created for the application's business services and exposes all the 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. Throughout this guide, the term data collection and collection are used interchangeably.

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.

For example, in an application that uses ADF Business Components to define the business services, each data control on the Data Control Palette represents a specific application module, and exposes the view object instances in that application's data model. The hierarchy of objects in the data control is defined by the view links between view objects that have specifically been added to the application module data model. For information about creating view objects and view links, see Chapter 5, "Querying Data Using View Objects". For information about adding view links to the data model, see Section 5.10.4.3, "How to Enable Active Master/Detail Coordination in the Data Model".

In the Data Control Palette, each data control object is represented by a specific icon. Table 12–1 describes what each icon represents, where it appears in the Data Control Palette hierarchy, and what components it can be used to create.

Table 12–1 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. Usually, there is one data control for each application module. However, you may have additional data controls that were created for other types of business services (for example, for CSV files or web services). For information about creating data controls for CSV files, see Chapter 31, "Creating Data Control Adapters". For information about creating data controls for web services, see Chapter 33, "Working with Web Services".	Not used to create anything. Serves as a container for the other objects.
	Method	Represents a custom method in the data control that can accept parameters, perform some action or business logic, and return single values or data collections. In application module data controls, custom methods are defined in the application module itself and usually return either nothing or a single scalar value. For more information about creating custom methods, see Chapter 8, "Implementing Business Services with Application Modules". For more information about using methods that accept parameters, see Section 17.3.2.1, "Using Parameters in a Method".	Command components For methods that accept parameters: command components and parameterized forms
	Method Return	Represents an object that is returned by a custom method. The returned object can be a single value or a collection. If a custom method defined in the application module returns anything at all, it is usually a single scalar value. Application module methods do not need to return a set of data to the view layer, because displaying the latest changes to the data is handled by the view objects in the active data model (for more information, see Section 4.5, "Understanding the Active Data Model". However, custom methods in non-application module data controls (for example, a data control for a CSV file) can return collections to the view layer. A method return appears as a child under the method that returns it. The objects that appear as children under a method return can be attributes of the collection, other methods that perform actions related to the parent collection, and operations that can be performed on the parent collection.	For single values: text fields and selection lists For collections: forms, tables, trees, and range navigation components

Table 12–1 (Cont.) The Data Control Palette Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
	View Object Collection	Represents a named data collection, which is the default rowset contained in a view object instance. The name of the collection matches the view object instance name. 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. Throughout this guide, data collection and collection are used interchangeably. A view link creates a master-detail relationship between two view objects. If you explicitly add an instance of a detail view object (resulting from a view link) to the application module data model, the collection contained in that detail view object appears as a child of the collection contained in the master view object. For information about adding detail view objects to the data model, see Section 5.10.4.3, "How to Enable Active Master/Detail Coordination in the Data Model". For more information about using master-detail relationships to create UI components, see Chapter 15, "Displaying Master-Detail Data". The children under a collection may be attributes of the collection, other collections that are related by a view link, custom methods that return a value from the collection, and built-in operations that can be performed on the collection.	Forms, tables, trees, range navigation components, and master-detail components
	Attribute	Represents a discrete data element in an object (for example, an attribute in a view row). Attributes appear as children under the collections or method returns to which they belong. Only the attributes that were included in the view object are shown under a collection. If a view object joins one or more entity objects, that view object's collection will contain selected attributes from all of the underlying entity objects.	Label, text field, 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 collections or method returns and under the root data control node. The operations that are children of a particular collection or method return operate on those objects only, while operations under the data control node operate on all the objects in the data control. If an operation requires one or more parameters, they are listed in a Parameters 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.

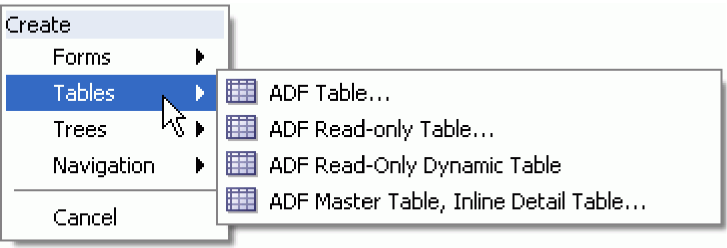
12.2.2 How to Use the Data Control Palette

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.

Figure 12–2 shows the context menu displayed when a collection from the Data Control Palette is dropped on a page.

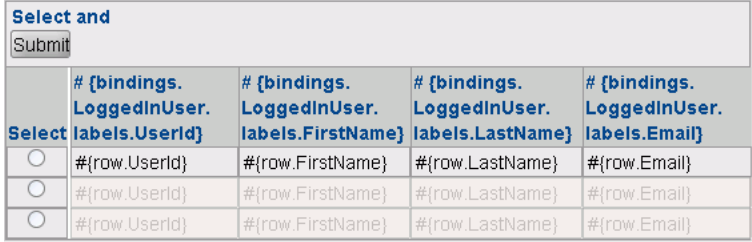
Figure 12–2 Data Control Palette Context Menu



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 14, "Adding Tables".)

The resulting UI component appears in the JDeveloper visual editor. For example, if you drag a collection 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 12–3.

Figure 12–3 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 12.6, "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 20, "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.

12.2.3 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 the binding definitions in the page definition file and registers the data controls used by those pages. For more information, see Section 12.3, "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 12.4, "Configuring the ADF Binding Filter".

- Register the ADF phase listener in the `faces-config.xml` file, as shown in Example 12–1.

Example 12–1 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 13.2.3, "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 12.5, "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 12.6, "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 11.4.2, "What Happens When You First Insert an ADF Faces Component".

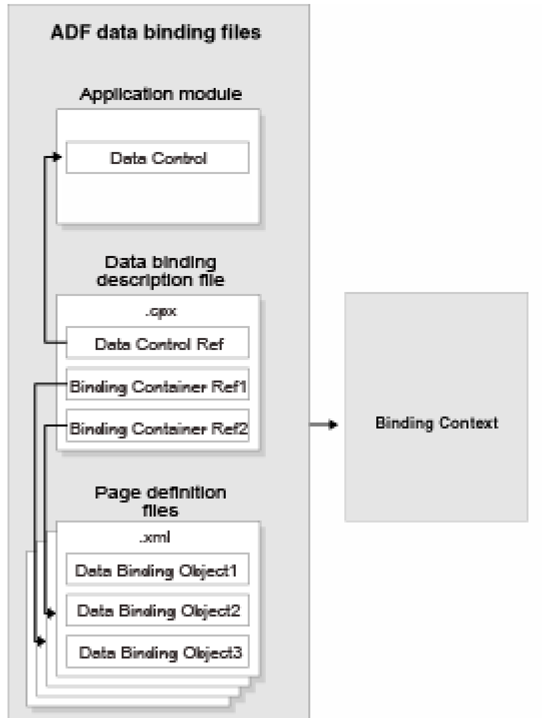
12.2.4 What Happens at Runtime

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 `DataBindings.cpx` file and page definition files, as shown in Figure 12–4. The application modules define the data controls available to the application at design time, but the `DataBindings.cpx` file defines what data controls are available to the application at runtime. The `DataBindings.cpx` file lists all the data controls that are being used by pages in the application and maps the binding containers, which contain the binding objects defined in the page definition files, to web page URLs.

The page definition files define the binding objects used by the application pages. There is one page definition file for each page. For information about the ADF lifecycle, see Section 13.2.3, "What Happens at Runtime: The JSF and ADF Lifecycles".

Figure 12–4 ADF Binding File Runtime Usage



12.3 Working with the DataBindings.cpx File

The `DataBindings.cpx` file defines the Oracle ADF binding context for the entire application and provides the metadata from which the Oracle ADF binding objects are created at runtime. It maps individual pages to page definition files and declares which data controls are being used by the application. At runtime, only the data controls listed in the `DataBindings.cpx` file are available to the current application.

12.3.1 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.

12.3.2 What Happens When You Create a DataBindings.cpx File

Example 12–2 shows a sample 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. The BC4JDataControl elements within the dataControlUsages element identify the data controls (application modules) being used by the application. The dc elements identify any data control adaptors (for example, data controls for CSV files) being used by the application. For more information about the elements and attributes in the DataBindings.cpx file, see Appendix A, "Reference ADF XML Files".

Example 12–2 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="app_SRListPageDef"/>
    <page path="/app/SRCreate.jspx" usageId="app_SRCreatePageDef"/>
    <page path="/app/SRCreateConfirm.jspx" usageId="app_SRCreateConfirmPageDef"/>
    ...
  </pageMap>
  <pageDefinitionUsages>
    <page id="app_SRListPageDef"
          path="oracle.srdemo.view.pageDefs.app_SRListPageDef"/>
    <page id="app_SRCreatePageDef"
          path="oracle.srdemo.view.pageDefs.app_SRCreatePageDef"/>
    <page id="app_SRCreateConfirmPageDef"
          path="oracle.srdemo.view.pageDefs.app_SRCreateConfirmPageDef"/>
    ...
  </pageDefinitionUsages>
  <dataControlUsages>
    <dc id="SRDemoFAQ" path="oracle.srdemo.faq.SRDemoFAQ"/>
    <BC4JDataControl id="SRService" Package="oracle.srdemo.model"
                     FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
                     SupportsTransactions="true" SupportsFindMode="true"
                     SupportsRangesize="true" SupportsResetState="true"
                     SupportsSortCollection="true"
                     Configuration="SRServiceLocal" syncMode="Immediate"
                     xmlns="http://xmlns.oracle.com/adfm/datacontrol"/>
  </dataControlUsages>
</Application>
```

12.4 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.

12.4.1 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.

12.4.2 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 12–3. 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 12–3 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 12–4. 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 12–4 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 12–5. 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 12–5 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.

12.4.3 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.

12.5 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.

12.5.1 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**.

12.5.2 What Happens When You Create a Page Definition File

Example 12–6 shows a sample page definition file that was created for the SRCreate 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 12.5.2.1, "Binding Objects Defined in the parameters Element")
- `executables` (for more information, see Section 12.5.2.2, "Binding Objects Defined in the executables Element")
- `bindings` (for more information, see Section 12.5.2.3, "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 12.6, "Creating ADF Data Binding EL Expressions", you will see how the ADF data binding EL expressions reference the binding object names.

Example 12–6 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="app_SRCreatePageDef"
                Package="oracle.srdemo.view.pageDefs">
  <parameters/>
  <executables>
    <invokeAction Binds="cancelNewServiceRequest"
                  id="clearServiceRequestFieldsIfNotInTrain"
                  Refresh="prepareModel"
                  RefreshCondition="${adfFacesContext.postback == false and empty
                                    requestScope.processChoice}"/>
    <iterator id="ProductListIterator" Binds="ProductList" RangeSize="-1"
              DataControl="SRService"/>
    <iterator id="GlobalsIterator" RangeSize="10" Binds="Globals"
              DataControl="SRService"/>
  </executables>
  <bindings>
    <attributeValues IterBinding="GlobalsIterator" id="ProblemDescription">
      <AttrNames>
        <Item Value="ProblemDescription"/>
      </AttrNames>
    </attributeValues>
    <list StaticList="false" ListOperMode="0" IterBinding="GlobalsIterator"
```

```
          ListIter="ProductListIterator" id="ProductList">
      <AttrNames>
        <Item Value="ProductId"/>
        <Item Value="ProductName"/>
      </AttrNames>
      <ListAttrNames>
        <Item Value="ProdId"/>
        <Item Value="Name"/>
      </ListAttrNames>
      <ListDisplayAttrNames>
        <Item Value="Name"/>
      </ListDisplayAttrNames>
    </list>
    <attributeValues IterBinding="GlobalsIterator" id="ProductId">
      <AttrNames>
        <Item Value="ProductId"/>
      </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="GlobalsIterator" id="ProductName">
      <AttrNames>
        <Item Value="ProductName"/>
      </AttrNames>
    <methodAction id="cancelNewServiceRequest"
    </attributeValues>
                  InstanceName="SRService.dataProvider"
                  DataControl="SRService"
                  MethodName="cancelNewServiceRequest"
                  RequiresUpdateModel="true" Action="999"/>
  </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.6, "<pageName>PageDef.xml".

12.5.2.1 Binding Objects Defined in the parameters Element

The `parameters` element of the page definition file defines the parameters for the page.

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 13.2.3, "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 12–7 shows how parameter binding objects can be defined in a page definition file.

Example 12–7 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 17, "Creating More Complex Pages".

12.5.2.2 Binding Objects Defined in the executables Element

The `executables` element of the page definition file defines the following types of executable binding objects:

- `iterator`: Binds to an iterator that iterates over view object collections.

  When you drop a collection or an attribute of a collection on the page, an iterator binding is automatically added to the `executables` element.

- `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 12.5.2.3, "Binding Objects Defined in the bindings Element".

  You will see `methodIterator` bindings in the `executables` element only if you drop a method return collection or an attribute of a method return collection from a non-application module data control on the page. If you are using only application module data controls, you will see only `iterator` bindings.

- `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.

  The `variableIterator` element can contain one of two types of variable definitions: `variable` and `variableUsage`. A `variable` type variable is a simple value holder, while a `variableUsage` type variable is a value holder that is related to a view object's named bind parameter. Defining a variable as a `variableUsage` type allows it to inherit the default value and UI control hints from the view object named bind variable to which it is bound.

- `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 12.5.2.3, "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 12.5.2.3, "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`.

For more information about how bindings are refreshed and how to set the `Refresh` and `RefreshCondition` attributes, see Section 10.5.5, "How to Use Refresh Correctly for InvokeAction and Iterator Bindings".

> **Tip:** Use the Structure window to re-order bindings in the `executables` element using drag and drop.

Example 12–8 shows an example of an `executables` element, which defines an `invokeAction` binding object and two iterator binding objects.

Example 12–8 The executables Element in a Page Definition File

```
<executables>
  <invokeAction Binds="cancelNewServiceRequest"
                id="clearServiceRequestFieldsIfNotInTrain"
                Refresh="prepareModel"
                RefreshCondition="${adfFacesContext.postback == false and empty
                                   requestScope.processChoice}"/>
  <iterator id="ProductListIterator" Binds="ProductList" RangeSize="-1"
            DataControl="SRService"/>
  <iterator id="GlobalsIterator" RangeSize="10" Binds="Globals"
            DataControl="SRService"/>
</executables>
<bindings>
  <attributeValues IterBinding="GlobalsIterator" id="ProblemDescription">
    <AttrNames>
      <Item Value="ProblemDescription"/>
    </AttrNames>
  </attributeValues>
  ...
  <methodAction id="cancelNewServiceRequest"
                InstanceName="SRService.dataProvider"
                DataControl="SRService"
                MethodName="cancelNewServiceRequest"
                RequiresUpdateModel="true" Action="999"/>
</bindings>
```

The `invokeAction` object invokes the `cancelNewServiceRequest` method, which is named in the `Binds` attribute. In the `bindings` wrapper element, the `methodAction` binding object encapsulates the details about how to invoke the method. (For more information about `methodAction` objects, see Section 12.5.2.3, "Binding Objects Defined in the bindings Element".) The `Refresh` attribute specifies 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.6, "<pageName>PageDef.xml".

The iterator binding named `ProductListIterator` was created by dropping the `ProductId` attribute of the `Globals` collection on the page as a list box. In the ensuing List Binding Editor, the **Add** button was used to create a new iterator binding for the items to display as choices in the list from the `ProductList` view object instance in the data model. (For more information, see Section 19.7, "Creating Selection Lists".)

The `Binds` attribute of the `iterator` element defines the collection the iterator will iterate over, which in this case is the `ProductList` collection. The `RangeSize` attribute defines the number of objects the iterator is to display on the page at one time. A `RangeSize` value of `-1` causes the iterator to display *all* the objects from the collection.

> **Tip:** Normally, an iterator binding's default range size is 10. However, when an iterator binding is created from the List Binding Editor, the range size defaults to `-1` so that all choices display in the list, not just the first 10.

The iterator binding named `GlobalsIterator` was created the first time one of the attributes from the `Globals` collection was dropped on the page. In this case, the `RangeSize` attribute is set to `10`, which means the iterator binding will display a maximum of 10 objects at a time from the collection. In the `bindings` wrapper element, notice that the `IterBinding` attribute of the `attributeValues` element references the `GlobalsIterator` iterator binding, which populates the `ProblemDescription` text field with data.

12.5.2.3 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.
- `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 12–9 shows a sample `bindings` element, which defines one attribute binding for a text field called `ProblemDescription` and one `methodAction` binding called `cancelNewServiceRequest`.

Example 12–9 The bindings Element of a Page Definition File

```
<bindings>
  <attributeValues IterBinding="GlobalsIterator" id="ProblemDescription">
    <AttrNames>
      <Item Value="ProblemDescription"/>
    </AttrNames>
  </attributeValues>
  ...
  <methodAction id="cancelNewServiceRequest"
                InstanceName="SRService.dataProvider"
                DataControl="SRService"
                MethodName="cancelNewServiceRequest"
                RequiresUpdateModel="true" Action="999"/>
</bindings>
```

The binding object defined in the `methodAction` element encapsulates the information needed to invoke the `cancelNewServiceRequest` method, which is identified in the `MethodName` attribute. The value of `true` in the `RequiresUpdateModel` attribute specifies that the model layer needs to be updated before the method is executed.

The `cancelnewServiceRequest` method does not accept parameters. If it did, the `methodAction` binding object definition would include one or more `NamedData` elements that would define the parameters expected by the method. For more information about passing parameters to methods, see Chapter 17, "Creating More Complex Pages".

The `attributeValues` element defines the value bindings for the text fields on the page. In the example, only one attribute is displayed on the page, `ProblemDescription`, which is defined in the `AttrNames` element. The `IterBinding` attribute references the iterator binding that displays the data in the text field.

12.5.3 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.

12.5.4 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. For more information about refreshing iterator bindings, see Section 10.5.5, "How to Use Refresh Correctly for InvokeAction and Iterator Bindings". Use the ADF Business Components State Management facility, described in Chapter 28, "Application Module State Management", to ensure that the rowset iterator state and any pending changes in the transaction are managed between requests.

12.6 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.

12.6.1 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:

```
#{bindings.BindingObject.propertyName}
```

where:

- `bindings` is a variable that identifies that the binding object being referenced by the expression is located in the binding container of the current page. All ADF data binding EL expressions must start with the `bindings` variable.
- *`BindingObject`* is the 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 12.6.4, "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 12.6.3, "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 12.6.2, "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.

12.6.2 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 12.6.4, "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.

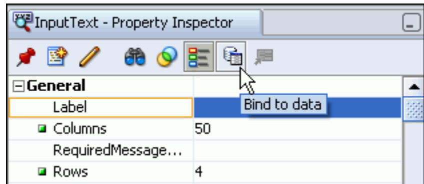
To open the Expression Builder from the Property Inspector:

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 12–5.

     (JDeveloper activates the Bind to data button only if it is valid to add a binding expression to the selected property.)

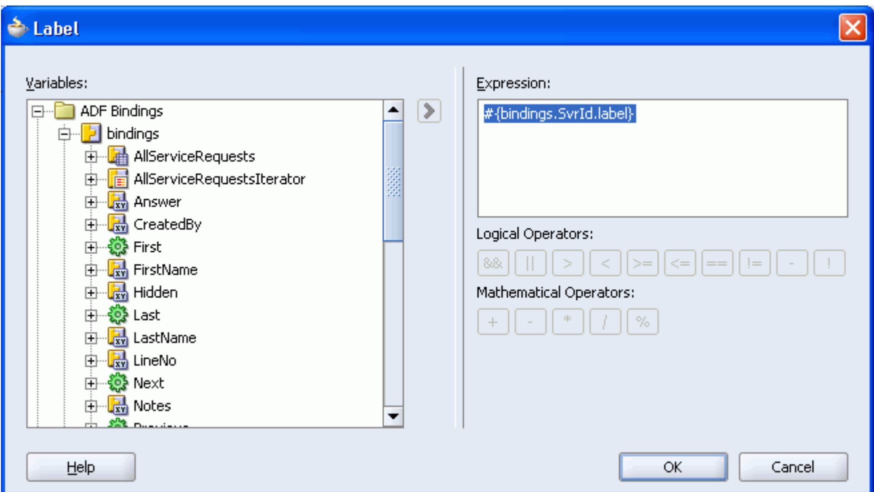
Figure 12–5 Bind to data Button in the Property Inspector



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 12–6.

Figure 12–6 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 12–2 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 12–2 Icons Under the ADF Bindings Node of the Expression Builder

Icon	Description
bindings	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.
data	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. Use this node only to view the binding objects defined for other pages; do not use it to create expressions in the current page. If you want to include an object in the current page that is defined in another page, add a binding to that object in the current page using the Data Control Palette.
	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. Use this node only to view the binding objects defined for other pages; do not use it to create expressions in the current page. If you want to include an object in the current page that is defined in another page, add a binding to that object in the current page using the Data Control Palette.
	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.

Table 12–2 (Cont.) Icons Under the ADF Bindings Node of the Expression Builder

Icon	Description
	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 12.6.4, "What You May Need to Know About ADF Binding Properties".
	Represents a parameter binding object.

12.6.3 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.

12.6.3.1 EL Expressions That Reference Attribute Binding Objects

Example 12–10 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 12–10 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 attribute binding object, `SvrId`, referenced by the EL expressions is defined in the page definition file, as shown in Example 12–11. 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 12–11 Attribute Binding Object Defined in the Page Definition File

```
<attributeValues id="SvrId" IterBinding="ServiceRequestsIterator">
  <AttrNames>
     <Item Value="SvrId"/>
  </AttrNames>
</attributeValues>
```

12.6.3.2 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 12–12.

Example 12–12 EL Expression That References a Table Binding Object

```
<af:table value="#{bindings.ServiceRequests.collectionModel}" var="row"
                  rows="#{bindings.ServiceRequests.rangeSize}"
                  first="#{bindings.ServiceRequests.rangeStart}"
                  emptyText="#{bindings.ServiceRequests.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 `ServiceRequests` table binding object, which is defined in the page definition file as shown in Example 12–13.

Example 12–13 Table Binding Object Defined in the Page Definition File

```
<table id="ServiceRequests" IterBinding="ServiceRequestsIterator">
   <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.

12.6.3.3 EL Expressions That Reference Action Binding Objects

Example 12–14 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 12–14 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 12–15 shows the action binding object defined in the page definition for the command button.

Example 12–15 Action Binding Object Defined in the Page Definition File for an Operation

```
<bindings>
  <action id="First" IterBinding="ServiceRequestsIterator"
          InstanceName="SRService.ServiceRequests" DataControl="SRService"
          RequiresUpdateModel="true" Action="12"/>
</bindings>
```

The `action` element, `First`, defines the action binding object that is directly referenced by the EL expression in the command button. The `IterBinding` attribute of the `action` binding references the iterator binding for the data collection being operated on by the action.

> **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 12–16 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 `deleteServiceHistoryNotes` method. The `execute` binding property in the EL expression in the `actionListener` attribute invokes the method when the user clicks the button

Example 12–16 EL Expression That References an Action Binding Object for a Method

```
<af:commandButton actionListener="#{bindings.deleteServiceHistoryNotes.execute}"
                  text="deleteServiceHistoryNotes"
                  disabled="#{!bindings.deleteServiceHistoryNotes.enabled}"/
```

Example 12–17 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 12–17 Method Action Binding Defined in the Page Definition File

```
<bindings>
  <methodAction id="deleteServiceHistoryNotes"
                InstanceName="SRService.dataProvider" DataControl="SRService"
                MethodName="deleteServiceHistoryNotes"
                RequiresUpdateModel="true" Action="999">
    <NamedData NDName="keySet" NDType="java.util.Set"/>
  </methodAction>
</bindings>
```

12.6.4 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, iterator bindings have a property called `RangeSize`, which specifies the number of rows the iterator should display at one time. The value assigned to `RangeSize` is specified in the page definition file, as shown in Example 12–18.

Example 12–18 Iterator Binding Object with the RangeSize Property

```
<iterator id="ServiceRequestsIterator" RangeSize="10"
          Binds="ServiceRequests" DataControl="SRService"/>
```

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 12.6.2, "How to Use the Expression Builder".

13 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 13.1, "Introduction to Creating a Basic Page"
- Section 13.2, "Using Attributes to Create Text Fields"
- Section 13.3, "Creating a Basic Form"
- Section 13.4, "Incorporating Range Navigation into Forms"
- Section 13.5, "Creating a Form to Edit an Existing Record"
- Section 13.6, "Creating an Input Form"
- Section 13.7, "Modifying the UI Components and Bindings on a Form"

13.1 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. You can create forms that display values, forms that allow users to edit values, forms that collect values (input forms), and search forms.

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 create different types of forms, such as forms that allow you to edit existing objects or create new ones.

- How to add operations that allow you to navigate between the data objects displayed in a form, or that allow you to manipulate data.
- How to modify the form once it has been created

13.2 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.

13.2.1 How to Use the Data Control Palette to Create a Text Field

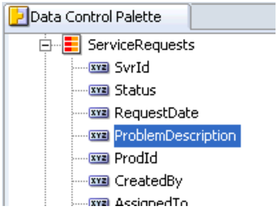
To create a text field that can display or update an attribute, you must bind the UI component to the attribute using a data control. JDeveloper allows you to do this declaratively by dragging and dropping an attribute of a collection from the Data Control Palette.

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 12.2.1, "How to Understand the Items on the Data Control Palette").

   For example, Figure 13–1 shows the `ProblemDescription` attribute under the `ServiceRequests` collection of the `SRService` data control in the SRDemo application. This is the attribute to drop to display or enter the problem description for a service request.

   Figure 13–1 Attributes Associated with a Collection in the Data Control Palette

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 20, "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 19.7, "Creating Selection Lists".

13.2.2 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 directory name, and appending PageDef as the name of the page definition file. For example, the page definition file for the SREdit page in the ./app/staff subdirectory of the web root is app_staff_SREditPageDef.xml. For a complete account of what happens when you drag an attribute onto a page, see Section 12.2.3, "What Happens When You Use the Data Control Palette". Bindings for the iterator and attributes are created and added to the page definition file. Additionally, the necessary JSPX page code for the UI component is added to the JSF page.

13.2.2.1 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 an iterator binding if it does not already exist. An iterator binding references an iterator for the data collection, which facilities 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 specify the current data object in the collection. Other bindings then refer to the iterator binding in order to return data for the current object or to perform an action on the object's data. Note that the iterator binding is not an iterator. it is a binding to an iterator. In the case of ADF Business Components, the actual iterator is the default row set iterator for the default row set of the view object instance in the application module's data model.

> **Tip:** There is one iterator binding created for each collection. This means that when you drop two attributes from the same collection (or drop the collection twice), they use the same binding. This is fine, unless you need the binding to behave differently for the different components. In that case, you will need to manually create separate iterator bindings. For procedures and an example, see Section 18.5, "Conditionally Displaying the Results Table on a Search Page".

For example, if you drop the `ProblemDescription` attribute under the `ServiceRequests` collection, JDeveloper creates an iterator binding for the `ServiceRequests` 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 13.4.2.2, "Iterator RangeSize Attribute". Example 13–1 shows the iterator binding created when you drop an attribute from the `ServiceRequests` collection.

Example 13–1 Page Definition Code for an Iterator Binding When You Drop an Attribute from a Collection

```
<executables>
  <iterator id="ServiceRequestsIterator" RangeSize="10"
            Binds="ServiceRequests" DataControl="SRService"/>
</executables>
```

For information regarding the iterator binding element attributes, see Section A.2.2, "Oracle ADF Data Binding Files".

This metadata allows the ADF binding container to access the attribute. Because the iterator binding is an executable, it is invoked when the page is loaded, thereby allowing the iterator to access and iterate over the `ServiceRequests` collection. This means that the iterator will manage all the service requests in the collection, including determining the current service request or range of service requests.

13.2.2.2 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 `ServiceRequests` collection as an ADF Output Text w/Label widget onto a page, JDeveloper creates an attribute binding for the `ProblemDescription` attribute. This allows the binding to access the attribute value of the current record. Example 13–2 shows the attribute binding for `ProblemDescription` created when you drop the attribute from the `ServiceRequests` collection. Note that the attribute value references the iterator named `ServiceRequestsIterator`.

Example 13–2 Page Definition Code for an Attribute Binding

```
<bindings>
   ...
   <attributeValues id="ServiceRequestsProblemDescription"
                    IterBinding="ServiceRequestsIterator">
     <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".

13.2.2.3 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 13–3 shows the code generated on the JSF page when you drop the `ProblemDescription` attribute as an Output Text w/Label widget.

Example 13–3 JSF Page Code for an Attribute Dropped as an Output Text w/Label

```
<af:panelLabelAndMessage
         label="#{bindings.ServiceRequestsProblemDescription.label}">
    <af:outputText
         value="#{bindings.ServiceRequestsProblemDescription.inputValue}"/>
</af:panelLabelAndMessage>
```

If instead you drop the `ProblemDescription` attribute as an Input Text w/Label widget, JDeveloper creates an `inputText` component. As Example 13–4 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 20.3, "Adding Validation" for more information about this property.
- `columns`: bound to the `displayWidth` property. This determines how wide the text box will be.

Example 13–4 JSF Page Code for an Attribute Dropped as an Input Text w/Label

```
<af:inputText value="#{bindings.ServiceRequestsProblemDescription.inputValue}"
            label="#{bindings.ServiceRequestsProblemDescription.label}"
            required="#{bindings.ServiceRequestsProblemDescription.mandatory}"
          columns="#{bindings.ServiceRequestsProblemDescription.displayWidth}">
  <af:validator
binding="#{bindings.ServiceRequestsProblemDescription.validator}"/>
</af:inputText>
```

For more information about the properties, see Appendix B, "Reference ADF Binding Properties".

13.2.3 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.

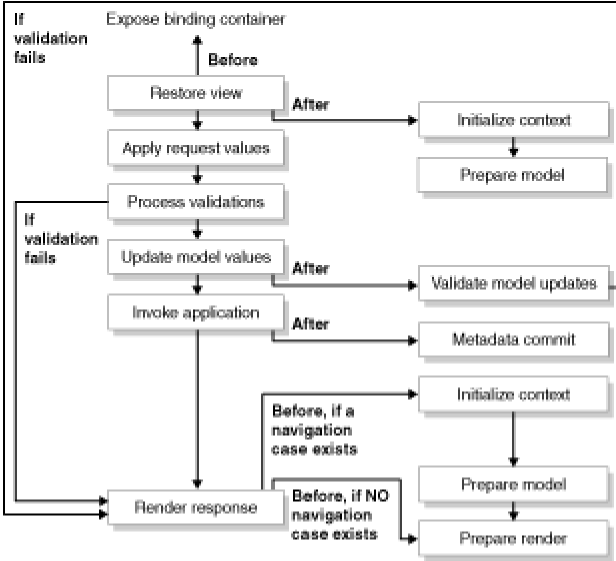
When an ADF Faces component bound to an ADF data control is inserted into a JSF page for the first time, JDeveloper adds the ADF PhaseListener to `faces-config.xml`. Example 13–5 shows the ADF PhaseListener configuration in `faces-config.xml`.

Example 13–5 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 13–2 shows how the JSF and ADF phases integrate in the lifecycle.

Figure 13–2 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.6.1, "PageDef.xml Syntax".

  In the problem description example, the `bindingContainer` invokes the `ServiceRequestsIterator` iterator, which returns the `ServiceRequests` 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 13–2), and since the UI component is bound to the `ProblemDescription` binding using an EL expression (`#{bindings.problemDescription.inputValue}`), that data can be 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 20, "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 20, "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 16.4, "Using Dynamic Navigation". For a description of action bindings used to invoke business logic, see Section 13.4, "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.

13.3 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 `ServiceRequests` collection, which contains all the attributes necessary to edit a given service request.

You can also create forms that provide more functionality than displaying data from a collection. For information about creating a form that allows a user to update data, see Section 13.5, "Creating a Form to Edit an Existing Record". For information about creating forms that allow users to create a new object for the collection, see Section 13.6, "Creating an Input Form". You can also create search forms. For more information, see Chapter 18, "Creating a Search Form".

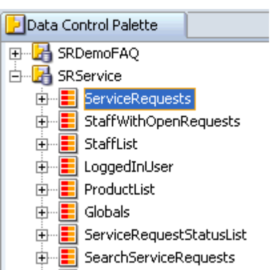
13.3.1 How to Use the Data Control Palette to Create a Form

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.

To create a basic form:

1. From the Data Control Palette, select the collection that represents the data you wish to display. Figure 13–3 shows the `ServiceRequests` collection for the `SRService` data control.

Figure 13–3 Collections Can Be Used to Create Forms that Display Data



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 20.3, "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 13.4, "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 Search Form**: Creates a form that can be used to execute a Query-By-Example (QBE) search. For more information, see Chapter 18, "Creating a Search Form".

   – **ADF Creation Form**: Creates an input form that allows users to create a new instance for the collection. For more information, see Section 13.6, "Creating an Input Form".

3. If you are building a form that allows users to update data, you now need to drag and drop an operation that will perform the update. For more information, see Section 13.5, "Creating a Form to Edit an Existing Record".

13.3.2 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 13–6 shows the code generated on the JSF page when you drop the `ServiceRequests` collection as a default ADF Form.

Example 13–6 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}"/>
  </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:panelForm>
```

Note: For information regarding the validator and converter tag, see Chapter 20, "Using Validation and Conversion".

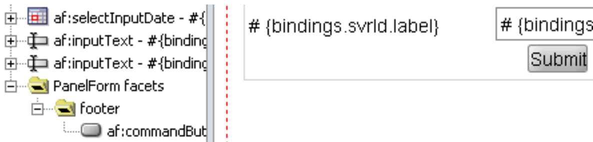
13.3.2.1 Using Facets

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 13–4 shows the command button in the `panelForm's footer` facet.

Figure 13–4 Footer Facet for the Panel Form



Example 13–7 shows the corresponding code in the JSF page.

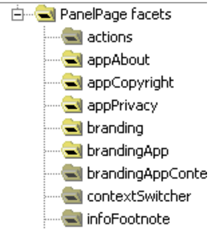
Example 13–7 Facet in a JSF Page

```
<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 13.4.2.3, "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 13–5 shows both full and empty facets for a `panelPage` component

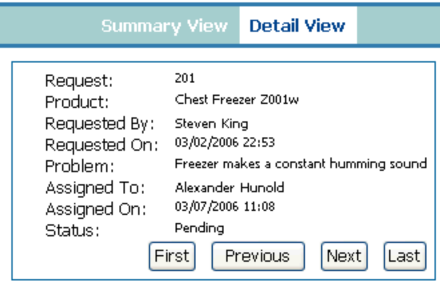
Figure 13–5 Empty and Full Facet Folders in the Structure Window



13.4 Incorporating Range Navigation into Forms

When you create an ADF Form, you are given the option of adding navigation. If you choose to do so, JDeveloper includes ADF Faces command components bound to existing navigational logic on the data control. This built-in logic allows the user to navigate through all the data objects in the collection. Figure 13–6 shows a form that contains navigation buttons.

Figure 13–6 Navigation in a Form



13.4.1 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 `ServiceRequests` collection. Figure 13–7 shows the operations associated with a collection.

Figure 13–7 Navigation Operations Associated With a Collection

Data Control Palette
ServiceRequests
SvrId
Status
RequestDate
ProblemDescription
ProdId
CreatedBy
AssignedTo
AssignedDate
CreatedByFirstName
CreatedByLastName
AssignedToFirstName
AssignedToLastName
ProductName
ProductId
ServiceHistories
Operations
Create
Delete
Find
Execute
First
Last
Previous
Next

2. Choose either **Command Button** or **Command Link** from the context menu.

13.4.2 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

13.4.2.1 Using Action Bindings for Built-in Navigation Operations

Action bindings execute business logic. For example, they can invoke built-in methods on the action binding object. These built-in methods operate on the iterator or on the data control itself, and are represented as operations in the Data 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.

Like value bindings, action bindings for operations will contain a reference to the iterator binding when the action binding is bound to one of the iterator-level actions, such as `Next` or `Previous`, 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 (action bindings to other than iterator-level actions, for example for a custom method on an AM, or for the commit or rollback operations, will not contain this reference). Example 13–8 shows the action bindings for the navigation operations.

> **Tip:** The numerical values of the `Action` attribute in the `<action>` tags (as shown in Figure 13–8) 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 13–8 Page Definition Code for an Operation Action Binding

```
<action id="First" RequiresUpdateModel="true" Action="12"
           IterBinding="ServiceRequestsIterator"/>
<action id="Previous" RequiresUpdateModel="true" Action="11"
           IterBinding="ServiceRequestsIterator"/>
<action id="Next" RequiresUpdateModel="true" Action="10"
           IterBinding="ServiceRequestsIterator"/>
<action id="Last" RequiresUpdateModel="true" Action="13"
           IterBinding="ServiceRequestsIterator"/>
```

13.4.2.2 Iterator RangeSize Attribute

Iterator bindings have a `rangeSize` attribute used to determine the number of data objects to make available for the page for each iteration. This attribute helps in situations when the number of objects in the data source is quite large. Instead of returning all objects, only a set number are returned and accessible by the other bindings. Once the iterator reaches the end of the range, it accesses the next set. Example 13–9 shows the default range size for the `ServiceRequestsIterator` iterator.

> **Note:** This `rangeSize` attribute is not the same as the `row` attribute on a table component. For more information, see Table 14–1, " ADF Faces Table Attributes and Populated Values".

Example 13–9 RangeSize Attribute for an Iterator

```
<executables>
  <iterator id="ServiceRequestsIterator" RangeSize="10"
            Binds="ServiceRequests" DataControl="SRService"/>
</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.

Table 13–1 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 13.4.3, "What Happens at Runtime: About Action Events and Action Listeners".

Table 13–1 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.

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".

13.4.2.3 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 13.4.3, "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 13–10 shows the code generated on the JSF page for navigation operation buttons.

Example 13–10 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>
```

13.4.3 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.

13.4.4 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 13–11 shows a page definition file after token validation was set to true.

Example 13–11 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">
```

13.5 Creating a Form to Edit an Existing Record

You can create a form that allows a user to edit the current data, and then commit those changes to the data source. You then use a operations associated with a collection or the data control itself to create command buttons that can be used to modify data records. For example, you use the `Delete` operation to create a button that allows a user to delete a record from the current range. Or you can use the built-in **Submit** button to submit changes.

> **Tip:** You can also use the Create operation on a form to create a new object, however using the ADF Creation Form provides additional built-in functionality. See Section 13.6, "Creating an Input Form" for more information.

It is important to note that these operations are executed only against objects in the ADF cache. You need to use the `Commit` operation on the root data control to actually commit any changes to the data source. You use the data control's `Rollback` operation to rollback any changes made to the cached object.

13.5.1 How to Use the Data Control Palette to Create Edit Forms

To use the operations on a form, you follow the same procedures as the navigation operations (see Section 13.4.1, "How to Insert Navigation Controls into a Form" for procedures), however you must also create the buttons for the commit and rollback operations in order for changes to be committed to the data store or to restore the cache.

To create an edit form:

1. From the Data Control Palette, drag the collection for which you wish to create the form, and select **ADF Form** from the context menu.
2. In the Edit Form Fields dialog, if you want the user to be able to change data, select **Include Submit Button**.
3. 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.

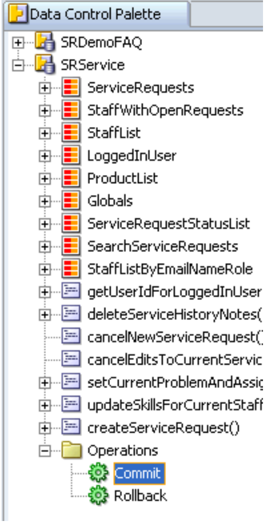
   For example, if you want to be able delete service requests, you would drag the `Delete` operation associated with the `ServiceRequests` collection. Figure 13–8 shows the operations associated with a collection.

Figure 13–8 Operations Associated With a Collection

4. Choose either **Command Button** or **Command Link** from the context menu.

5. From the Data Control Palette, drag the `Commit` and `Rollback` operations associated with the root level data control, and drop them as either a command button or command link. This will allow the changes to be committed to or rolled back. Figure 13–9 shows the commit and rollback operations for the `SRService` data control.

Figure 13–9 Commit and Rollback Operations for a Data Control



13.5.2 What Happens When You Use Built-in Operations to Change Data

Dropping any data control operation as a command button causes the same events as dropping navigation operations. See Section 13.4.2, "What Happens When Command Buttons Are Created Using the Data Control Palette" for more information.

The only difference is that the action bindings for the `Commit` and `Rollback` operations do not require a reference to the iterator, as they execute a method on the application module. Example 13–12 shows the action bindings generated in the page definition file for these operations.

Example 13–12 Action Bindings for Commit and Rollback Operations

```
<action id="Commit" InstanceName="SRService"
            DataControl="SRService" RequiresUpdateModel="true"
            Action="100"/>
<action id="Rollback" InstanceName="SRService"
            DataControl="SRService" RequiresUpdateModel="false"
            Action="101"/>
```

The following table shows the built-in non-navigation operations provided on data controls, along with the result of invoking the operation or executing an event bound to the operation (see Section 13.4.3, "What Happens at Runtime: About Action Events and Action Listeners" for more information about action events).

Table 13–2 More Built-in Operations

Operation	Action Attribute Value	When invoked, the associated iterator binding will...
Create	41	Creates a row directly before the current row, then moves the current row pointer to the new row. Note that the range does not move, meaning that the last row in the range may now be excluded from the range. Also note that this performs a `Create` operation and not a `CreateInsert` operation. In other words, the record will not be inserted into the rowset, avoiding a blank row should the user navigate away without actually creating data. The new row will be created when the user submits the data.
Delete	30	Deletes the current row from the cache and moves the current row pointer to the next row in the result set. Note that the range does not move, meaning that a row may be added to the end of the range. If the last row is deleted, the current row pointer moves to the preceding row. If there are no more rows in the collection, the enabled attribute is set to "disabled."
SetCurrentRowWith Key	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 14.7.1, "How to Manually Set the Current Row".
SetCurrentRowWith KeyValue	98	Set the current object on the iterator, given a key's value.
RemoveRowWithKey	99	Uses the row key as a `String` converted from the value specified by the input field to remove the data object in the bound data collection.
Commit	100	Causes all items currently in the cache to be committed to the database.
Rollback	101	Clears the cache and returns the transaction and iterator to the initial state.
Execute and Find		These operations are used only in search forms. See Chapter 18, "Creating a Search Form" for more information.

13.6 Creating an Input Form

You can create a form that allows a user to enter information for a new record and then commit that record into the data source. While you can choose to use the default ADF Form and then drop the Create operation as a command button, when this type of form is first rendered, it displays the data for the first instance in the collection. The ADF Creation form allows users to create new instances in the collection without first displaying existing instances.

There may be times, however, when you need more control over how a new object is created. For example, you may want certain attributes to be populated programmatically. In this case, you might create a custom method on your application module to handle the creation of objects. To use a custom method to create an input form, instead of using an ADF Creation form, you use a standard form and then drop a custom method as a command button. The custom method will execute when the user clicks the button.

Note: When you create a row programmatically, you should call the `NewRow.setNewRowState(RowSet.STATUS_INITIALIZED` method to get the same behavior as the built-in `Create` action. For more information, see Section 10.4.4, "What You May Need to Know About Create and CreateInsert".

13.6.1 How to Create an Input Form

You create an input form similar to the way you create any other form. However, by selecting an ADF Creation Form when dropping a collection, JDeveloper provides additional functionality automatically.

To create an input form:

1. From the Data Control Palette, drag the collection for which you wish to create the form, and select **ADF Creation Form** from the context menu.
2. In the Edit Form Fields dialog, do not include the Submit button.
3. From the Data Control Palette, drag the Commit and Rollback operations associated with the root data control, and drop them as command buttons or links.
4. In the Structure window, select the command button for the commit operation.
5. In the Property Inspector, set the command button's **Disabled** property to `False`.

   By default, JDeveloper binds the `Disabled` attribute of the button to the `Enabled` property of the binding, causing the button to be disabled when the `Enabled` property is set to `False`. For this binding, the `Enabled` property is `false` until an update has been posted. For the purposes of an input form, the button should always be enabled, since there will be no changes posted before the user needs to create the new object.

13.6.2 What Happens When You Create an Input Form

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

- Creates an iterator binding for the collection, an action binding for the Create operation, and attribute bindings for each of the attributes of the object in the collection. It also creates an invoke action in the executables section of the page definition that causes the `Create` operation to execute during the Render Model phase. If you created command buttons or links using the `Commit` and `Rollback` operations, JDeveloper also creates an action bindings for those operations.
- Inserts code in the JSF page for the form using ADF Faces `inputText` components, and in the case of the operations, `commandButton` components.

For example, to create a simple input form for products in the SRDemo application, you might drop the `ProductsList` collection from the Data Control Palette as an ADF Creation Form, and then drop the Commit operation as a button below the form, as shown in Figure 13–10.

Note: This page is an example only and does not exist in the SRDemo application.

Figure 13–10 A Simple Create Product Form

* Product Id
* Product
* Description
Commit

Example 13–13 shows the page definition file for this input form. When the invoke action executes during the prepare render lifecycle phase, the `Create` operation is invoked, and a new instance for the collection is created. For more information about the lifecycle, see Chapter 13.2.3, "What Happens at Runtime: The JSF and ADF Lifecycles"

Example 13–13 Page Definition Code for a Creation Form

```
<executables>
  <iterator id="ProductListIterator" RangeSize="10" Binds="ProductList"
            DataControl="SRService"/>
  <invokeAction Binds="Create" id="invokeCreate" Refresh="renderModel"
                RefreshCondition="${!adfFacesContext.postback and empty
                                             bindings.exceptionsList}"/>
</executables>
<bindings>
  <action id="Create" RequiresUpdateModel="true" Action="41"
          IterBinding="ProductListIterator"
          InstanceName="SRServiceDataControl.ProductList"
          DataControl="SRServiceDataControl"/>
  <attributeValues id="ProdId" IterBinding="ProductListIterator">
    <AttrNames>
      <Item Value="ProdId"/>
    </AttrNames>
  </attributeValues>
  <attributeValues id="Name" IterBinding="ProductListIterator">
```

```
      <AttrNames>
        <Item Value="Name"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="Description" IterBinding="ProductListIterator">
      <AttrNames>
        <Item Value="Description"/>
      </AttrNames>
    </attributeValues>
    <action id="Commit" InstanceName="SRServiceDataControl"
            DataControl="SRServiceDataControl" RequiresUpdateModel="true"
            Action="100"/>
</bindings>
```

Anytime the `Create` operation is invoked (for example, when the page is first rendered), an data object is created. However, since data is cached (which allows the `Create` operation to do a postback to the server), and since the `Create` operation is invoked whenever the page enters the `renderModel` phase, the `Create` operation 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 Create operation will only be invoked whenever there has been no postback to the server and as long as there are no error messages (see Example 13–13 for the EL expression). When the user clicks the **Commit** button (which is bound to the Commit action through an action binding), the new object with the data is submitted to the data source. If you do want the user to be able to stay on the same page to create multiple objects, see Chapter 13.6.4, "What You May Need to Know About Create Forms and the RefreshCondition".

13.6.3 What You May Need to Know About Displaying Sequence Numbers

Because the `invokeCreate` action is executed before the page is displayed, if you are populating the primary key (in this case the Product ID) using sequences, that number will appear in the input text field. For example, instead of being blank, the Product ID in Figure 13–10 displays the product ID number. This is because the product entity class contains a method that uses an eager fetch to generate a sequence of numbers for the `prodId` attribute. This populates the value as the row is created.

However, if instead you've configured the attribute's type to DBSequence (which uses a database trigger to generate the sequence), the number would not be populated until the object is committed to the database. In this case, the user would see a negative number as a placeholder. To avoid this, you can use the following EL expression for the `Rendered` attribute of the input text field:

```
#{bindings.EmployeeId.inputValue.value > 0}
```

This expression will display the component only when the value is greater than zero, which will not be the case before it is committed. Similarly, you can simply set the `Rendered` attribute to `false`. However, then the page will never display the input text field component for the primary key.

13.6.4 What You May Need to Know About Create Forms and the RefreshCondition

If the **Commit** button for the input form does not cause the user to navigate off the page, the page re-renders displaying the data from the last entry. This is because the `refreshCondition` on the `invokeCreate` invoke action (`${!adfFacesContext.postback and empty bindings.exceptionsList}`) causes the action to not be invoked when there is a postback, which is the case when committing the data.

If you want users to be able to stay on the page (for example to create more than one product at a time), you need to change the `refreshCondition` so that it will allow the `invokeCreate` action to execute after committing a new object. To do this, you use the `setActionListener` component within the **Commit** command component. The `setActionListener` component allows the command component to set a value before navigating. In this case, the command component will set a flag (named `createAnother`) on the request scope to `true`. The `refreshCondition` can then evaluate this flag to determine whether or not to execute the `invokeCreate` action, as shown in the following procedure. For more information about the `setActionListener` component, see Section 17.4.2, "What Happens When You Set Parameters".

To create an input form that can create multiple objects:

1. Follow the procedure to create an input form.
2. On the JSF page, drag a `setActionListener` component from the Component Palette onto the **Commit** command button.

   > **Note:** This procedure assumes that the user will click this **Commit** button and return to the same page to create another entry. You may wish to change the name of the button to reflect that, for example, **Commit and Create Another**. You will need to create a different button to navigate off the page.

3. In the Insert setActionListener dialog, set the following:
   - **From**: `#{true}`
   - **To**: `#{requestScope.createAnother}`

   This creates a flag named `createAnother` on the request scope, and sets it to `true`.
4. Open the page definition for the JSF page.
5. Change the `refreshCondition` for the `invokeCreate` invoke action from

   ```
   ${!adfFacesContext.postback and empty bindings.exceptionsList}
   ```

   to:

   ```
   #{empty bindings.expressionList and (!adfFacesContext.postback or
    requestScope.createAnother)}
   ```

   This sets the condition to execute the `invokeCreate` action when the expression list that contains error messages is empty, and there was not or postback or the `createAnother` flag on the request scope is set to true.

Note: The `setActionListener` component executes during the Invoke Application phase of the lifecycle. Therefore, the `refresh` attribute for the `invokeCreate` action must be set to a subsequent phase. In this case, it is set to `renderModel` by default, and should not be changed.

13.7 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.

13.7.1 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 12.6, "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 13.7.1.1, "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". This property allows your page to use the UI control hints for labels that you have defined for your entity object attributes or view object attributes, where you can change the value once and have it appear the same on all pages that display the label.

  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 22.4, "Internationalizing Your Application".

13.7.1.1 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*>**.

13.7.1.2 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.

13.7.2 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.

14 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 14.1, "Introduction to Adding Tables"
- Section 14.2, "Creating a Basic Table"
- Section 14.3, "Incorporating Range Navigation into Tables"
- Section 14.4, "Modifying the Attributes Displayed in the Table"
- Section 14.5, "Adding Hidden Capabilities to a Table"
- Section 14.6, "Enabling Row Selection in a Table"
- Section 14.7, "Setting the Current Object Using a Command Component"

14.1 Introduction to Adding Tables

Unlike forms, tables allow you to display more than one data object from a collection at a time. Figure 14–1 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 14–1 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 row selection components that allow users to select a specific row. When you add command buttons bound to actions, users can then click those buttons to execute some logic on the selected row. For more information, see Section 17.3, "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

14.2 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.

14.2.1 How to Create a Basic Table

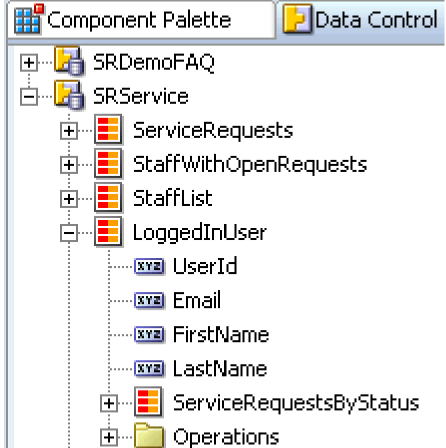
To create a table using a data control, you bind the `table` component to a view object collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Control Palette.

To create a databound table:

1. From the Data Control Palette, select a collection.

   For example, to create the SRList table in the SRDemo application, you select the `ServiceRequestsByStatus` collection that is under the `LoggedInUser` collection. Figure 14–2 shows the `ServiceRequestsByStatus` collection in the Data Control Palette.

Figure 14–2 ServiceRequestsByStatus Collection in the Data Control Palette



The ServiceRequestsByStatus collection, which extends ServiceRequests, is a child of the LoggedInUser collection because of the view link ServiceRequestsForUser. The ServiceRequestsByStatus collection also has a named bind variable StatusCode that represents the service request status type (for example, open or pending requests). In the SRList page, when the logged in user selects a command link in the menu bar to view open, pending, closed, or all service requests, the requests created by or assigned to the currently logged in user for the selected status type are returned.

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

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

   - **ADF Table**: Allows you to select the specific attributes you 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 polymorphic collection (for example, a view object collection that can be a collection of mammals or a collection of birds), the dynamic table can display the different attributes accordingly.
   - **ADF Master Table, Inline Detail Table**: For more information, see Section 15.6, "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 labels property for the attribute on the table binding. For more information about the labels property, see Appendix B, "Reference ADF Binding Properties". The bindings to the labels property allow the labels to be inherited from the UI control hints that you have defined in your business domain layer, thus enabling you to change the value of a label text once in a central place, and have the change appear the same on all pages that display the label. In the Edit Table Columns 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 heading for the Status column in the table on the SRList page is bound to the labels property that uses the Status key to get the attribute:

     ```
     #{bindings.LoggedInUserServiceRequests.labels.Status}
     ```

     However, you could change the heading to instead be bound to a key in a properties resource file, for example:

     ```
     #{srlist['sr.status']}
     ```

     In the 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 22.4, "Internationalizing Your Application".

Note that the SRDemo pages mainly use the inherited UI control hints for all attributes, and JSF resource strings for other kinds of labels that are not directly related to view object attributes.

- Change the attribute binding for a column.

  For example, you can change the status column to instead be bound to the `requestDate` attribute. If you simply want to rearrange the columns, you should use the order buttons, as described later in the section. If you do change the attribute binding for a column, 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.

- 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, replace the component with the desired UI component (such as a command link).

- 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. There's no limit to the number of columns you can add. When you first click **New**, JDeveloper adds a new column line at the bottom of the dialog and populates it with default values from the first attribute in the bound collection; subsequent new columns are populated with values from the next attribute in the sequence, and so on.

- Delete a column using the **Delete** 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 14.6, "Enabling Row Selection in a Table".

- Allow sorting for all columns by selecting **Enable sorting**.

14.2.2 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 13.2.2, "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.

14.2.2.1 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; the iterator binding references an iterator for the data collection, which facilities iterating over the data objects in the collection. Instead of creating a separate binding for each attribute, only the table binding is created. In the table binding, the `AttrNames` element contains a a child element for each attribute that you want to be available for display or reference in each row of the table. Example 14–1 shows the value binding for the table created when you drop the `ServiceRequestsByStatus` collection.

Example 14–1 Value Binding Entries for a Table in the Page Definition File

```
<table id="LoggedInUserServiceRequests"
       IterBinding="ServiceRequestsByStatusIterator">
  <AttrNames>
    <Item Value="SvrId"/>
    <Item Value="Status"/>
    <Item Value="RequestDate"/>
    <Item Value="ProblemDescription"/>
    <Item Value="ProdId"/>
    <Item Value="CreatedBy"/>
    <Item Value="AssignedTo"/>
    <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.

14.2.2.2 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, which 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 is bound to the `labels` property for the attribute on the table binding. Example 14–2 shows a simplified code excerpt from the table on the SRList page.

Example 14–2 Simplified JSF Code for an ADF Faces Table

```
<af:table value="#{bindings.LoggedInUserServiceRequests.collectionModel}"
          var="row" ..>
  ...
  <af:column headerText="#{bindings.LoggedInUserServiceRequests.labels.Status}"
             sortProperty="Status" sortable="false">
    <af:outputText value="#{row.Status}"/>
  </af:column>
  <af:column
           headerText="#{bindings.LoggedInUserServiceRequests.labels.RequestDate}"
           sortProperty="RequestDate" sortable="false">
    <af:outputText value="#{row.RequestDate}">
      <f:convertDateTime
           pattern="#{bindings.LoggedInUserServiceRequests.formats.RequestDate}"/>
    </af:outputText>
  </af:column>
  ...
</af:table>
```

The table binding iterates over the data exposed by the iterator binding. The `FacesCtrlRangeBinding` class extends the base `JUCtrlRangeBinding` class to add specific methods to the base table binding object; one of the methods is the `getCollectionModel` method, which the EL accesses using the `collectionModel` property of the table binding. The table wraps the result set from the iterator binding in an `oracle.adf.view.faces.model.CollectionModel` object. As the table binding iterates, it makes each item in the collection available within the `table` component using the `var` attribute.

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

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

Table 14–1 shows the other attributes defined by default for ADF Faces tables created using the Data Control Palette.

Table 14–1 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 14.3, "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 14.3, "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**.

Table 14–1 (Cont.) ADF Faces Table Attributes and Populated Values

Attribute	Description	Default Value
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 14.6, "Enabling Row Selection in a Table".

14.3 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 14–3 shows an example of how the `selectRangeChoiceBar` component might look like in a table.

Figure 14–3 SelectRangeChoiceBar in a Table

My Service Requests

Select and View Edit — Previous 1-5 of 7 Next 2

Select	Request Id	Status	Requested On	Problem	Assigned On
(•)	106	Closed	Nov 25, 2005	Ice machine not working	Nov 26, 2005
( )	109	Closed	Dec 9, 2005	Freezer is not cold	Dec 10, 2005
( )	110	Pending	Dec 15, 2005	Freezer lid will not fully close	Dec 16, 2005
( )	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

14.3.1 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:

```
<af:table rows="#{bindings.LoggedInUserServiceRequests.rangeSize}".../>
```

The EL expression in the rows attribute evaluates to the iterator's range size, which is defined in the page definition file:

```
<executables>
  <iterator id="ServiceRequestsByStatusIterator" RangeSize="10"
            Binds="ServiceRequestsByStatus" DataControl="SRService"/>
  </executables>
```

By default, the RangeSize value is 10, which means that 10 records are returned at a time for display in the current page.

To change the number of records to return for display in the current page, edit the RangeSize value in the page definition file (instead of editing directly the rows attribute on the table component).

If you change the rows attribute on the table component directly instead of changing the RangeSize value on the iterator, make sure the value of rows is equal to or less than the iterator's RangeSize value. For additional information about the RangeSize attribute, see Section 13.4.2.2, "Iterator RangeSize Attribute".

14.3.2 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 14–3). When you change the RangeSize attribute on the iterator, the selectRangeChoiceBar component automatically changes to show the new range sets.

The rows attribute on a table component is used in conjunction with the first attribute to set the ranges. The first attribute, which is a zero-based index of the rows in a range, determines the first row in the current range. 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 also uses an EL expression, but the expression binds 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.LoggedInUserServiceRequests.rangeSize}"
          first="#{bindings.LoggedInUserServiceRequests.rangeStart}"
```

Each current range starts with the row identified by first, and contains only as many rows as indicated by the rows attribute.

14.3.3 What Happens at Runtime

To determine the range sets for the `selectRangeChoiceBar` to use, the view object retrieves the first "RangeSize" number of rows and then stops, and the table makes a separate `SELECT COUNT(*) FROM (...)` query by calling the `getEstimatedRowCount()` method, which estimates the total number of rows the query would retrieve without actually retrieving them all. For more information about the `getEstimatedRowCount()` method, see Section 5.6.2, "Counting the Number of Rows in a RowSet".

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 17.5, "Overriding Declarative Methods".

14.3.4 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 13. For more information, see Section 13.4.4, "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 14–1, 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 13.4.4, "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.

14.4 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.

14.4.1 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 Table Columns dialog, you can do the following:
   - Change the display label for the column. By default, the label is bound to the `labels` property for the attribute on the table binding. For more information about the `labels` property, see Appendix B, "Reference ADF Binding Properties". The bindings to the `labels` property allow the labels to be inherited from the UI control hints that you have defined in your business domain layer, thus enabling you to change the value of a label text once in a central place, and have the change 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 heading for the Status column in the table on the SRList page is bound to the `labels` property that uses the `Status` key to get the attribute:

     ```
     #{bindings.LoggedInUserServiceRequests.labels.Status}
     ```

     However, you could change it to instead be bound to a key in a properties resource 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 22.4, "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, 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 composite component you selected when you dropped the collection onto the page. You can change to the other UI component using the dropdown menu. If you want to use an entirely 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, replace the `outputText` component with the desired UI component (such as a command link).

  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.Boolean` types.
  - The `selectOneListbox/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 19.7, "Creating Selection 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 14.6, "Enabling Row Selection in a Table".
- Add sorting capabilities by selecting **Enable sorting**.

14.4.2 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 14.4.1, "How to Modify the Displayed Attributes" to modify those bindings.

14.4.3 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.

14.5 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 13.3.2.1, "Using Facets". Figure 14–4 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 14–4 Table with an Output UI Component in the DetailStamp Facet

My Service Requests

Select and View Edit

Show All Details | Hide All Details

Select	Details	Request Id	Status	Requested On
◉	Show	200	Open	Dec 19, 2005
○	Hide	201	Open	Dec 20, 2005
Dryer is spitting out lots of lint.				
○	Hide	202	Open	Dec 20, 2005
Leaking at the sides				

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 15.6, "Using an Inline Table to Display Detail Data in a Master Table".

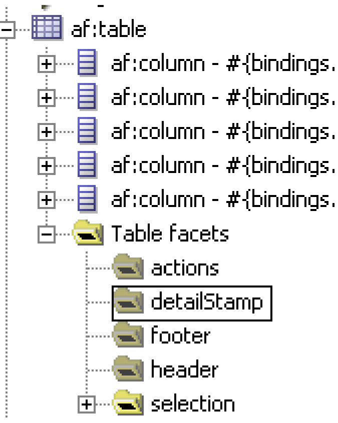
14.5.1 How to Use the DetailStamp Facet

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. From the Data Control Palette, drag the attribute to be displayed in the facet onto the **detailStamp** facet folder. Figure 14–5 shows how the **detailStamp** facet folder appears in the Structure window.

Figure 14–5 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>
```

14.5.2 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. Since the table was created using the `ServiceRequestsByStatus` collection, you can drag the `ProblemDescription` attribute and drop it inside the **detailStamp** facet folder in the Structure window.

Example 14–3 shows the code JDeveloper then adds to the JSF page.

Example 14–3 JSF Code for a detailStamp Facet

```
<f:facet name="detailStamp">
  <af:outputText
value="#{bindings.ServiceRequestsByStatusProblemDescription.inputValue}"/>
</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 14–4 shows how the code should appear after using the `row` variable.

Example 14–4 Modified JSF Code for a detailStamp Facet

```
<f:facet name="detailStamp">
  <af:outputText value="#{row.ServiceRequestsByStatusProblemDescription}"/>
</f:facet>
```

14.5.3 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.

14.6 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 14–6. 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 14–6 The SRList Table Uses the TableSelectOne Component

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

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 14–7. 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 14–7 The Service History Table Uses the TableSelectMany Component

Select and ... [Delete Service History Record]

Select All | Select None

Select	Date	Type	Note
☐	Dec 10, 2005	Technician	Asked customer to check if freezer is plugged in
☐	Dec 12, 2005	Customer	Freezer is plugged in, please suggest something else
☐	Dec 12, 2005	Technician	Asked customer to set freezer temperature to lowest setting and check after 24 hours
☐	Dec 13, 2005	Customer	Freezer is now cold

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 20.3.1.1.1, "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 11.6, "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.

14.6.1 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 14.6.5, "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 21.6, "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 21.6, "Adding ADF Bindings to Actions". For more information about using methods to create command buttons, see Section 17.3, "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.

14.6.2 What Happens When You Use the TableSelectOne Component

As Example 14–5 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 14–5 Selection Facet Code

```
<f:facet name="selection">
  <af:tableSelectOne text="Select and">
    <af:commandButton text="Submit"/>
</af:tableSelectOne>
```

As Example 14–6 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 selection facet to set the selected row as the current object on the iterator.

Example 14–6 Selection Attributes on a Table

```
<af:table var="row"
   rows="#{bindings.ServiceRequests.rangeSize}"
   first="#{bindings.ServiceRequests.rangeStart}"
   selectionState="#{bindings.ServiceRequests.collectionModel.selectedRow}"
   selectionListener="#{bindings.ServiceRequests.collectionModel.makeCurrent}"...>
```

14.6.3 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 17.5, "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.

14.6.4 What You May Need to Know About Using Links Instead of the Selection Facet

As described in Section 14.6.1, "How to Use the TableSelectOne Component in the Selection Facet", a `commandButton` component is automatically added as a child of the `tableSelectOne` component when you drop a collection as a table and select **Enable selection** at the same time, or when you use the Edit Table Columns dialog to enable selection later. The `tableSelectOne` component is inserted into the `selection` facet on the `table` component.

When the user makes a selection in the table, the `tableSelectOne` component sets the selected row as the current object on the iterator. Then the user can perform some action on the selected row via the command button.

Instead of using the `selection` facet components to set the current object and providing a `commandButton` to navigate to the next page, you can use a `commandLink` component that lets the user click a link to both perform an operation on a selection and navigate to another page. As shown in Figure 14–6, the second column of links in the table saves the user the step of having to first select a row and then click a command button to perform an action and navigate. If you add column links, you must manually set the current object on the iterator binding. For more information about manually setting the current object, see Section 14.7, "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 manually set the parameter that represents the selected row for the subsequent page. For example, from the SRList page, when the user selects a service request and then clicks the **Edit** command button, the application navigates to the SREdit page, which displays the correct data for the selected service request. The **Edit** command button, which uses the `setCurrentRowWithKey` action binding, includes the `af:setActionListener` tag to set the appropriate value into `processScope` before navigation. The SREdit page has an `invokeAction` object that invokes the `setCurrentRowWithKey` operation; the value bound to the `rowKey NamedData` element is passed in as the parameter, which determines the current row to display. For more information, see Section 17.4, "Setting Parameter Values Using a Command Component".

14.6.5 How to Use the TableSelectMany Component in the Selection Facet

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.

Unlike the `tableSelectOne` component, with the `tableSelectMany` component, there are multiple selected objects.

When you add the `tableSelectMany` component to a table that uses an ADF table binding, you need to pass the set of selected Keys to a method that processes each one in turn.

To use the tableSelectMany component in an ADF application:

1. Create the table as shown in Section 14.2.1, "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 objects on top of the **af:tableSelectMany** node. From the ensuing context menu, choose **Create > ADF Command Button**. Doing so drops the method as a `commandButton` component. You need to set the parameter value for the method, if the method accepts parameters.

   For example, to create the SRMain page where a user can delete multiple service history records associated with a service request, you would drag the custom `deleteServiceHistoryNotes(Set)` operation onto the **af:tableSelectMany** node.

5. In the Action Binding Editor, enter the value for the 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, the table on the SRMain page is bound to the `historyTable` property on the managed bean named `backing_SRMain`. To access the set of selected rows in the table, you would use the following as the value for the parameter:

   ```
   #{backing_SRMain.historyTable.selectionState.keySet}
   ```

   For more information, see Section 17.3.1, "How to Create a Command Component Bound to a Service Method".

6. Add logic to allow the declarative method to operate against the set of selected rows. To add the logic, you override the declarative method created when the command button was dropped. For instructions, see Section 17.5, "Overriding Declarative Methods". Briefly, you override the declarative method by binding the command button's `action` attribute to a backing bean method that has the added logic. Example 14–11 shows the backing bean method `onDeleteHistoryRows()` created for the SRMain page.

14.6.6 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

Example 14–7 shows the code for the `table`, `tableSelectMany`, and `commandButton` components.

Example 14–7 JSF Code for a Table That Uses the TableSelectMany Component

```
<af:table value="#{bindings.ServiceHistories.collectionModel}" var="row"
          rows="#{bindings.ServiceHistories.rangeSize}"
          first="#{bindings.ServiceHistories.rangeStart}"...
          binding="#{backing_SRMain.historyTable}">
  <af:column headerText="#{bindings.ServiceHistories.labels.SvhDate}"
             sortable="false">
    <af:outputText value="#{row.SvhDate}">
      <f:convertDateTime pattern="#{bindings.ServiceHistories.formats.SvhDate}"/>
    </af:outputText>
  </af:column>
  <af:column headerText="#{bindings.ServiceHistories.labels.SvhType}"
             sortable="false">
    <af:outputText value="#{row.SvhType}"/>
  </af:column>
  <af:column headerText="#{bindings.ServiceHistories.labels.Notes}"
             sortable="false">
    <af:outputText value="#{row.Notes}"/>
  </af:column>
  <f:facet name="selection">
    <af:tableSelectMany text="Select items and ...">
      <af:commandButton text="..."
                  actionListener="#{bindings.deleteServiceHistoryNotes.execute}"
                       .../>
    </af:tableSelectMany>
  </f:facet>
</af:table>
```

When you create a command button using a custom service 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. Example 14–8 shows the page definition code that contains the method action binding for the bound method on the command button.

Example 14–8 DeleteServiceHistoryNotes Method Action Binding

```
<methodAction id="deleteServiceHistoryNotes"
              InstanceName="SRService.dataProvider" DataControl="SRService"
              MethodName="deleteServiceHistoryNotes"
              RequiresUpdateModel="true" Action="999">
  <NamedData NDName="keySet"
             NDValue="${backing_SRMain.historyTable.selectionState.keySet}"
             NDType="java.util.Set"/>
</methodAction>
```

The method action binding causes the associated method to be invoked on the business service, passing in the value bound to the `NamedData` element as the parameter.

Instead of binding the command button to the `execute` method on the action binding, you can bind the button to a method in a backing bean that overrides the declarative method. Doing so allows you to add logic before or after the original method runs. When you override a declarative method, JDeveloper automatically rebinds the command component to the new method using the `action` attribute (instead of the `actionListener` attribute), as shown in Example 14–9.

Example 14–9 Command Button Code After You Override the Declarative Method

```
<af:commandButton text="..."
                  action="#{backing_SRMain.onDeleteHistoryRows}"/>
```

Example 14–10 shows the binding container code that JDeveloper inserts into the backing bean when you override the declarative method. The code accesses the binding container and finds the binding for the associated method.

Example 14–10 Binding Container Code Added by JDeveloper in the Backing Bean

```
private BindingContainer bindings;

public BindingContainer getBindings() {
       return this.bindings;
}

public void setBindings(BindingContainer bindings) {
       this.bindings = bindings;
}

public String onDeleteHistoryRows() {
    BindingContainer bindings = getBindings();
    OperationBinding operationBinding =
        bindings.getOperationBinding("deleteServiceHistoryNotes");
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
        return null;
    }
    return null;
}
```

By adding logic before or after the `execute()` call on the method binding, you can perform needed logic before or after the declarative method.

14.6.7 What Happens at Runtime

Like the `tableSelectOne` component, when the user makes multiple selections with the `tableSelectMany` component and then clicks the associated command button, the `tableSelectMany` component updates the selection state of the table by placing all the rows selected by the user in a `RowKeySet`.

Example 14–11 shows the SRMain page backing bean method that overrides the declarative method created when the `deleteServiceHistoryNotes(Set)` operation is dropped as a command button.

Example 14–11 Backing Bean Method for Deleting Service History Records

```
public String onDeleteHistoryRows() {
  BindingContainer bindings = getBindings();
  Set keySet = getHistoryTable().getSelectionState().getKeySet();
  if (!keySet.isEmpty()) {
    getBindings().getOperationBinding("deleteServiceHistoryNotes").execute();
    keySet.clear();
  }
  return null;
}
```

Example 14–12 shows the code in the backing bean that you need to add for working with the service history table.

Example 14–12 Backing Bean Code for Working with the Service History Table

```
private CoreTable historyTable;

public void setHistoryTable(CoreTable historyTable) {
  this.historyTable = historyTable;
}

public CoreTable getHistoryTable() {
  return historyTable;
}
```

When the user selects multiple rows in the service history table, and then clicks the **Delete Service History Record** button, the `tableSelectMany` component updates the selection state of the table by placing all the selected rows in a `RowKeySet`. The backing bean method `onDeleteHistoryRows()` accesses the binding container, and retrieves the selected keys by calling `getSelectionState.getKeySet()` on the history table. The backing bean method then executes the `deleteServiceHistoryNotes` method action binding (which is bound to the `deleteServiceHistoryNotes()` method in the application module's client interface) by calling the `execute()` method on the binding. The service method loops through the `keySet`, deleting the row found by each key. Finally, the backing bean method calls the `clear()` method on the `keySet` to remove the keys, after the service method has deleted all the selected rows.

14.7 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 14–8, which the user can click to directly edit a service request, without needing to first select the row.

Figure 14–8 Command Links Used in a Table on the SRList Page

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

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.

14.7.1 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**.

   JDeveloper creates the command link component on the page, and adds the action binding to the page definition file. You need to change the value for the `rowKey` parameter in the Action Binding Editor.
3. Select the command link component in the Structure window, and choose **Edit Binding** from the context menu.
4. In the Action Binding Editor, by default, the value for the `rowKey` parameter 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 14–8 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 14.2.2.2, "Code on the JSF Page for an ADF Faces Table".

14.7.2 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 17.3, "Creating Command Components to Execute Methods").

Example 14–13 shows the code created in the page definition file when you drop the `setCurrentRowWithKey` operation and set `#{row.rowKeyStr}` as the value for the `rowKey` parameter.

Example 14–13 Page Definition Code for the SetCurrentRowWithKey Operation

```
<action id="setCurrentRowWithKey" IterBinding="ServiceRequestsByStatusIterator"
            InstanceName="SRService.ServiceRequestsByStatus"
            DataControl="SRService" RequiresUpdateModel="false"
            Action="96">
  <NamedData NDName="rowKey" NDValue="${row.rowKeyStr}"
             NDType="java.lang.String"/>
</action>
```

14.7.3 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 17.4, "Setting Parameter Values Using a Command Component".

15

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 15.1, "Introduction to Displaying Master-Detail Data"
- Section 15.2, "Identifying Master-Detail Objects on the Data Control Palette"
- Section 15.3, "Using Tables and Forms to Display Master-Detail Objects"
- Section 15.4, "Using Trees to Display Master-Detail Objects"
- Section 15.5, "Using Tree Tables to Display Master-Detail Objects"
- Section 15.6, "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 19.7, "Creating Selection Lists".

15.1 Introduction to Displaying Master-Detail Data

In ADF Business Components, a master-detail relationship refers to two view object instances that are related by a view link. As you may recall from Chapter 5, "Querying Data Using View Objects", a view link represents the relationship between two view objects, which is usually, but not necessarily, based on a foreign-key relationship between the underlying data tables. ADF uses the view link to associate a row of one view object instance (the master object) with one or more rows of another view object instance (the detail object).

View links support two different types of master-detail coordination: view link accessor attributes and data model view link instances (for more information, see Section 27.1.3, "Understanding View Link Accessors Versus Data Model View Link Instances"). However, when displaying master-detail data on a page using ADF data binding, you exclusively use data model view link instances, which support active data model master-detail coordination. To enable active data model master-detail coordination, you must add both the master view object and the detail view object instances to the application module data model (for more information, see Section 5.10.4.3, "How to Enable Active Master/Detail Coordination in the Data Model".) For example, in the SRDemo application, there is a view link from the `ServiceRequests` view object to the `ServiceHistories` view object based on the `Svr_Id` attribute (service request ID). Both the master and detail view objects have been added to the application module data model. So, a change in the current row of the master view object instance causes the rowset of the detail view object instance to refresh to include the details for the current master.

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

15.2 Identifying Master-Detail Objects on the Data Control Palette

JDeveloper enables you to declaratively create pages that display master-detail data using the Data Control Palette. The Data Control Palette displays master-detail related objects in a hierarchy that mirrors the one you defined in the application module data model, where the detail objects are children of the master objects. For information about adding master-detail objects to the data model, see Section 5.10.4.3, "How to Enable Active Master/Detail Coordination in the Data Model".

Figure 15–1 shows two master-detail related collections in the Data Control Palette of the SRDemo application. The `ServiceRequests` collection is an instance of the `ServiceRequests` view object, and the `ServiceHistories` collection, which appears as a child of the `ServiceRequests` collection, is an instance of the `ServiceHistories` view object. The master-detail hierarchy on the Data Control Palette reflects the hierarchy defined in the `SRService` application module data model, as shown in Figure 15–2. The hierarchy was established by creating a view link from the `ServiceRequests` view object to the `ServiceHistories` view object. Next, an instance of the resulting detail view object, `ServiceHistories via ServicesHistoriesForServiceRequest1`, has been added to the application module data model, and named `ServiceHistories`, as shown in Figure 15–2.

> **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).

Figure 15–1 Master-Detail Objects on the Data Control Palette

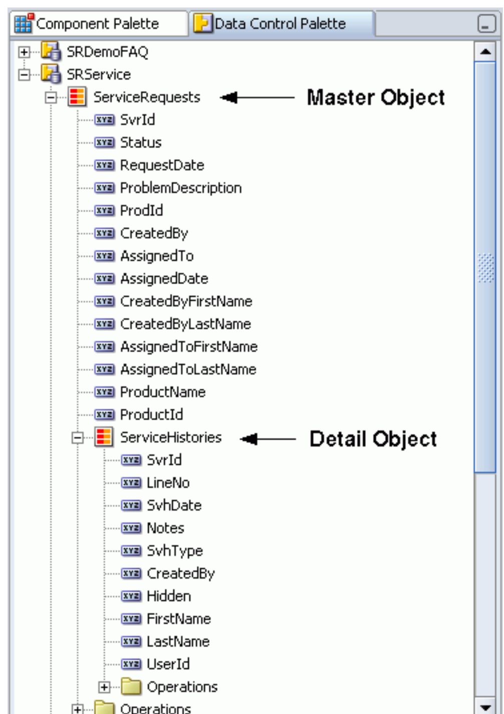
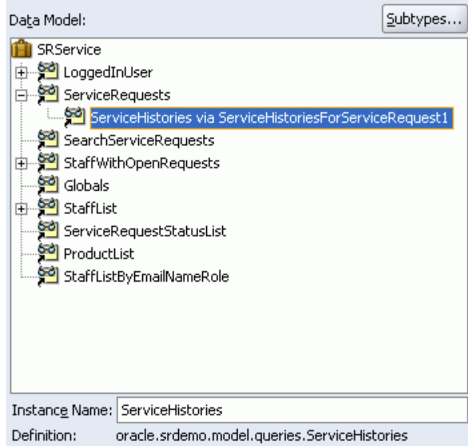


Figure 15–2 Master-Detail Hierarchy Defined in the Application Module Data Model



In the SRDemo application, the view link between the `ServiceRequests` view object and `ServiceHistories` view object is a one-way relationship. If the view link was bi-directional and both sets of master and detail view objects were added to the application module data model, then the Data Control Palette would also display the `ServiceHistories` collection at the same node level as the `ServiceRequests` collection, and the detail instance of the `ServiceRequests` collection as child of the `ServiceHistories` collection.

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:** By default, when you define a view link using the Create View Link wizard, the source view object is the master and the destination view object is the detail. However, if you choose to generate accessors in both the source and the destination view objects, then the master-detail relationship is bi-directional. If both sets of master-detail view objects resulting from a bi-directional view link are added to the application module data model, then instances of both sets of view objects will appear independently on the Data Control Palette.

For more information about the icons displayed on the Data Control Palette, see Section 12.2.1, "How to Understand the Items on the Data Control Palette".

15.3 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.

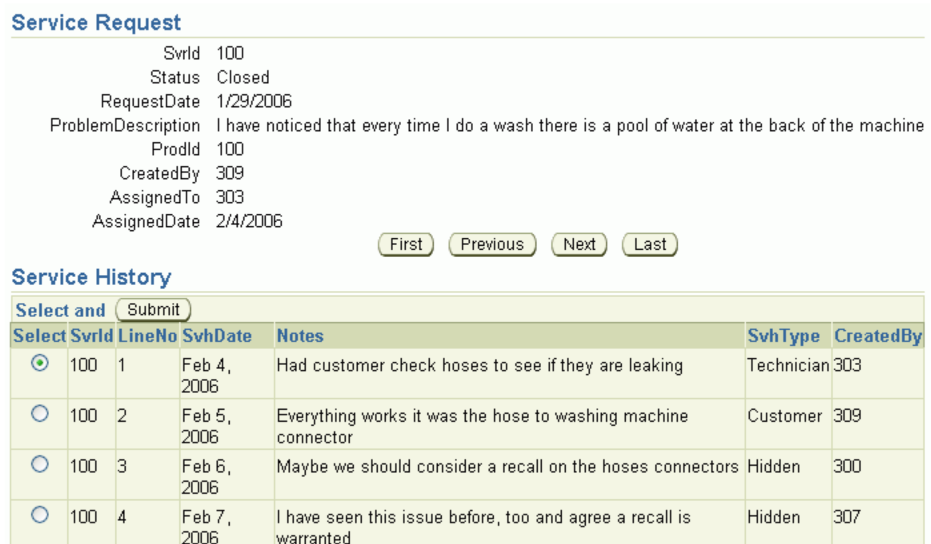
The pre-built master-detail widgets available from the Data Control Palette include range navigation that enables the user to scroll through the data objects in collections. The the table provided by the pre-built master-detail widgets includes a selection facet and **Submit** command button. By default, 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 13, "Creating a Basic Page" or Chapter 14, "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. At runtime, the active data model and the rowset iterator for the detail view object instance keep the rowset of the detail view object refreshed to the correct set of rows for the current master row as that current row changes.

Figure 15–3 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 15–3 Pre-Built Data Control Palette Master-Detail Widget



15.3.1 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 15.3.4, "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 15.2, "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.
   - **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.

If you want to modify the default forms or tables, see Chapter 13, "Creating a Basic Page" or Chapter 14, "Adding Tables".

15.3.2 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 12.2.3, "What Happens When You Use the Data Control Palette".

15.3.2.1 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 13, "Creating a Basic Page" and Chapter 14, "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 13.4, "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 17.3, "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 19.4, "Enabling Partial Page Rendering".

15.3.2.2 Binding Objects Defined in the Page Definition File

Example 15–1 shows the page definition file created for a master-detail page that was created by dropping the `ServiceHistories` collection, which is a detail object under the `ServiceRequests` object, on the page as an **ADF Master Form, Detail Table**.

The `executables` element defines two iterators: one for the service requests (which is the master object) and one for the service history (which is the detail object). The underlying view link from the master view object to the detail view object establishes the relationship between the two iterators. At runtime, the active data model and the rowset iterator for the detail view object instance keep the rowset of the detail view object refreshed to the correct set of rows for the current master row as that current row changes.(for more information, see Section 15.3.3, "What Happens at Runtime").

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

The 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.6, "<pageName>PageDef.xml".

Example 15–1 Binding Objects Defined in the Page Definition for a Master-Detail Page

```
<executables>
    <iterator id="ServiceRequestsIterator" RangeSize="10"
              Binds="ServiceRequests" DataControl="SRService"/>
    <iterator id="ServiceHistoriesIterator" RangeSize="10"
              Binds="ServiceHistories" DataControl="SRService"/>
  </executables>
  <bindings>
    <attributeValues id="SvrId" IterBinding="ServiceRequestsIterator">
      <AttrNames>
        <Item Value="SvrId"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="Status" IterBinding="ServiceRequestsIterator">
      <AttrNames>
        <Item Value="Status"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="RequestDate" IterBinding="ServiceRequestsIterator">
      <AttrNames>
        <Item Value="RequestDate"/>
      </AttrNames>
    </attributeValues>
    ...
    <action id="First" RequiresUpdateModel="true" Action="12"
            IterBinding="ServiceRequestsIterator"/>
    <action id="Previous" RequiresUpdateModel="true" Action="11"
            IterBinding="ServiceRequestsIterator"/>
    <action id="Next" RequiresUpdateModel="true" Action="10"
            IterBinding="ServiceRequestsIterator"/>
    <action id="Last" RequiresUpdateModel="true" Action="13"
            IterBinding="ServiceRequestsIterator"/>
    <table id="ServiceRequestsServiceHistories"
           IterBinding="ServiceHistoriesIterator">
      <AttrNames>
        <Item Value="SvrId"/>
        <Item Value="LineNo"/>
        ...
      </AttrNames>
    </table>
  </bindings>
```

15.3.3 What Happens at Runtime

As was previously mentioned in Section 12.5.2.2, "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. Because of the underlying view link from the master view object to the detail view object, the detail rowset iterator listens for row navigation events in both the master and detail collections. 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.

15.3.4 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 13, "Creating a Basic Page" or Chapter 14, "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 15.2, "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 16, "Adding Page Navigation".

15.4 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 15.5, "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 15–4 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 15–4 Databound ADF Faces Tree

15.4.1 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
- A view link accessor attribute that returns a detail object to be displayed as a branch of the current node level (for information about view link accessors, see Chapter 5.10.2, "How to Create Master/Detail Hierarchies Using View Links")

To create the tree on the SRMange page, a view object, `StaffWithOpenRequests`, was created to return just the users that have open or pending service requests. Another view object, `OpenOrPendingServiceRequests`, was created to return all the open or pending service requests. A view link was created from the `StaffWithOpenRequests` view object to the `OpenOrPendingServiceRequests` view object, thus establishing the master-detail relationship. To add a third-level node, for example, service history, a view link would need to exist from the service request view object to the service history view object. For more information about creating view links, see Section 5.10, "Working with Master/Detail Data".

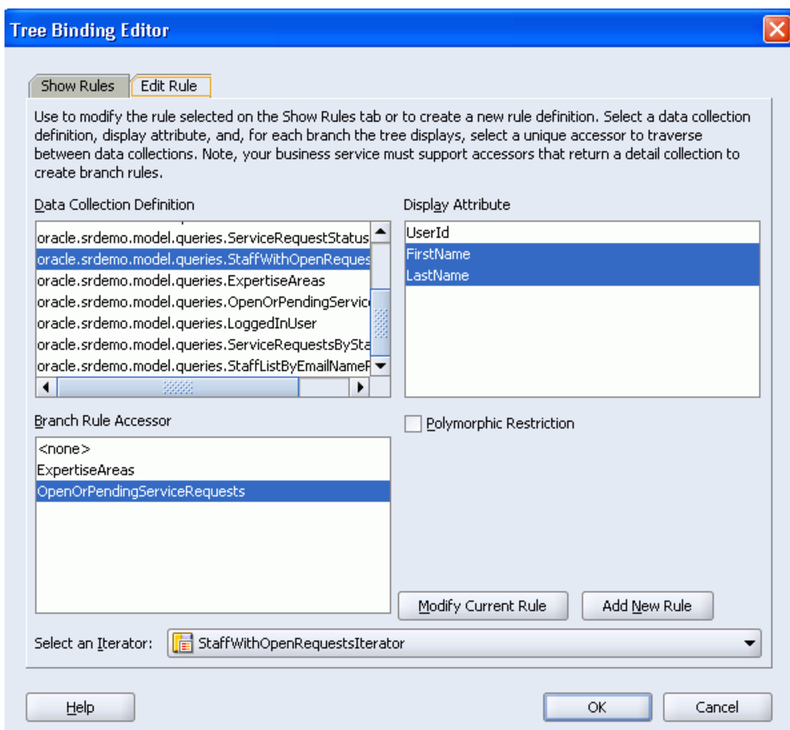
To display master-detail objects in a tree:

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.

2. In the context menu, choose **Trees** > **ADF Tree**.

   JDeveloper displays the Tree Binding Editor, as shown in Figure 15–5.

Figure 15–5 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 view link accessor attribute 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 attributes that return the detail collections for the master collection you selected for the rule. If you select **<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 attribute that returns the service request collection. Then, you must define a new rule for the `ServiceRequest` node level.

  View link accessor attributes, which return data collections, are generated when you create a view link. The **Branch Rule Accessor** field displays all accessor attributes that return detail collections for the master collection selected in the **Data Collection Definition** field. For more information about view objects, view links, and view link accessors, see Chapter 5, "Querying Data Using View Objects".

- **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 15–6, to:

   - Change the order of the rules

     The order of the rules should reflect the hierarchy that you want the tree to display.

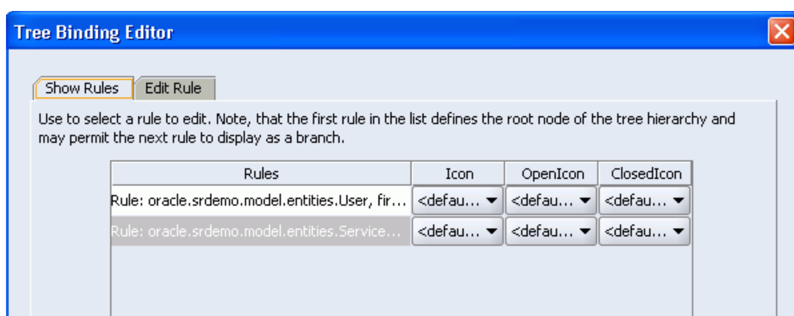
   - Delete rules

   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 15–4, 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 15–6 Tree Binding Editor, Show Rule Tab



15.4.2 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 12.2.3, "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.

15.4.2.1 Code Generated in the JSF Page

Example 15–2 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 `LoggedInUser` collection was used to populate the root node, which displays the users.

Example 15–2 Code Generated in the JSF Page for a Databound Tree

```
<h:form>
   <af:tree value="#{bindings.StaffWithOpenRequests.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 `LoggedInUser` tree binding object in the page definition file. The `treeModel` property in the binding expression refers to an ADF class that defines how the tree hierarchy is displayed, based on the underlying data model. The `var` attribute provides access to the current node.

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

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

the hierarchy. Any element contained by the `PathSet` instance is deemed expanded. All other elements are collapsed.

15.4.2.2 Binding Objects Defined in the Page Definition File

Example 15–3 shows the binding objects defined in the page definition file for the ADF databound tree.

Example 15–3 Binding Objects Defined in the Page Definition File for a Databound Tree

```
<executables>
    <iterator id="StaffWithOpenRequestsIterator" RangeSize="10"
              Binds="StaffWithOpenRequests" DataControl="SRService"/>
</executables>
<bindings>
  <tree id="StaffWithOpenRequests"
        IterBinding="StaffWithOpenRequestsIterator">
    <AttrNames>
      <Item Value="UserId"/>
      <Item Value="FirstName"/>
      <Item Value="LastName"/>
    </AttrNames>
    <nodeDefinition DefName="oracle.srdemo.model.queries.StaffWithOpenRequests"
                    id="StaffWithOpenRequestsNode">
      <AttrNames>
        <Item Value="FirstName"/>
        <Item Value="LastName"/>
      </AttrNames>
      <Accessors>
        <Item Value="OpenOrPendingServiceRequests"/>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition id="OpenOrPendingServiceRequestsNode"
               DefName="oracle.srdemo.model.queries.OpenOrPendingServiceRequests">
      <AttrNames>
        <Item Value="Status"/>
        <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.

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

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

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

The order of the nodeDefintion elements within the page definition file defines the order or level of the nodes in the tree, 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.6, "<pageName>PageDef.xml".

15.4.3 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.

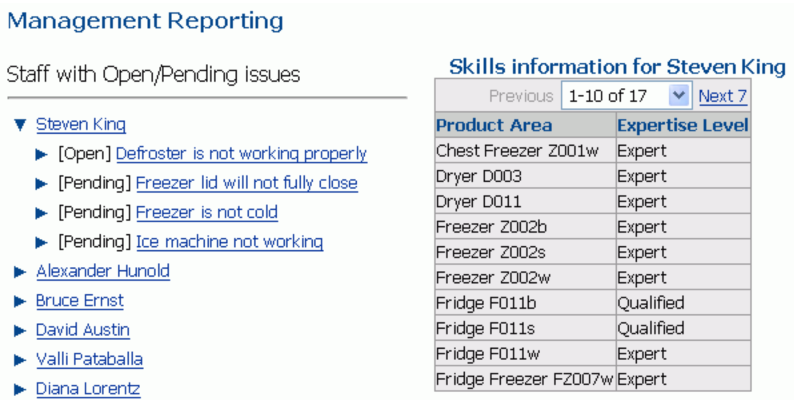
When a page with a tree is displayed, the iterator binding on the tree populates the root nodes. When a user collapses or expands a node to display or hide its branches, a DisclosureEvent event is sent. The isExpanded method on this event determines whether the user is expanding or collapsing the node.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

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

15.4.4 What You May Need to Know About Adding Command Links to Tree Nodes

The tree component on the SRManage page of the SRDemo application, which displays staff members in the root nodes and service requests in the branch nodes, was created using the default tree component available from the Data Control Palette. However, each node of the tree contains a command link that dynamically displays additional detail data related to that node in a separate UI component. For example, if the user clicks a staff member, the related expertise areas (detail objects) are displayed in a table to the right of the tree, as shown in Figure 15–7.

Figure 15–7 SRManage Page Displaying a Tree and Dynamic Detail Table



To achieve this functionality, an ADF Faces `switcher` component was manually added to the `tree` tag's `nodestamp` facet to dynamically display the data from detail objects when the user clicks a command link.

In the `switcher` component, which is shown in Example 15–4, the `facetName` attribute contains a JSF binding on the `SRManage` backing bean, which is configured as a managed bean in the `faces-config.xml` file. The backing bean has a `treeLevel` property of type `Map` that is declaratively defined as a managed property in the `faces-config.xml` file.

Example 15–4 ADF Faces switcher Component

```
<af:switcher facetName="#{backingSRManage.treeLevel[node.hierType.viewDefName]}">
  <f:facet name="StaffNode">
    <af:commandLink text="#{node}"
                  action="#{backing_SRManage.staffLinkDrilldown}"
                  actionListener="#{bindings.setCurrentStaffRowWithKey.execute}"/>
  </f:facet>
  <f:facet name="ServiceRequestsNode">
    <af:panelGroup>
      <af:outputText  value="[#{node.Status}] "/>
        <af:commandLink text="#{node.ProblemDescription}"
           action="#{backing_SRManage.problemLinkDrilldown}"
           actionListener="#{bindings.setCurrentProblemAndAssigneeRows.execute}"/>
    </af:panelGroup>
  </f:facet>
</af:switcher>
```

The managed property definition in the `faces-config.xml` contains information that injects the key-value pairs (`viewdefinitionname`, `facetname`) into the `treeLevel Map`, as shown in Example 15–5. The `Map` is used to display a different presentation in the tree for each different view object type.

Example 15–5 The treeLevel Managed Property Definition in the faces-config.xml File

```
<managed-bean>
    <managed-bean-name>backing_SRManage</managed-bean-name>
    ...
    <managed-property>
      <property-name>treeLevel</property-name>
      <map-entries>
        <key-class>java.lang.String</key-class>
        <value-class>java.lang.String</value-class>
        <map-entry>
          <key>oracle.srdemo.model.queries.StaffWithOpenRequests</key>
          <value>StaffNode</value>
        </map-entry>
        <map-entry>
          <key>oracle.srdemo.model.queries.OpenOrPendingServiceRequests</key>
          <value>ServiceRequestsNode</value>
        </map-entry>
      </map-entries>
    </managed-property>
  </managed-bean>
```

When you add command links to nodes in a tree, at runtime a user could click a link that is not the current object in the iterator that is populating the tree. (The current object in the iterator is the one displayed the last time the user opened a tree node.) Therefore, to display the correct detail object, the command links must be bound to operations or methods that programmatically set the current object in the iterator. For example, in the SRManage page, the command link on the service request node is bound to the `setCurrentProblemAndAssigneeRows` method, which programmatically sets the current object. For more information about manually setting the current object in a command component, see Section 14.7, "Setting the Current Object Using a Command Component". For more information about the ADF Faces `switcher` component, refer to the ADF Faces Javadoc.

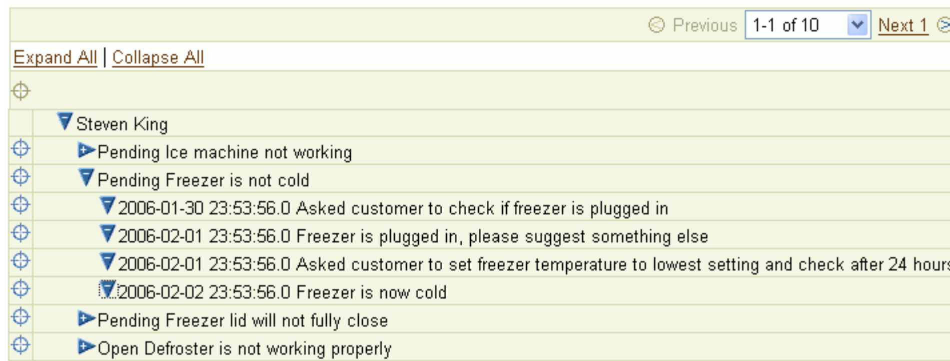
15.5 Using Tree Tables to Display Master-Detail Objects

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 15–8 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.

As with trees, to create a tree table with multiple nodes, it is necessary create view links between the view objects. The view links establish the master-detail relationships For example, to create the tree table shown in Figure 15–8, it was necessary to create view links from the user view object to the service request view object, and another view link from the service requests view object to the service history view object. For more information about creating view links, see Section 5.10, "Working with Master/Detail Data".

Figure 15–8 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.

15.5.1 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 15.4.1, "How to Display Master-Detail Objects in Trees".

15.5.2 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 12.2.3, "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.

15.5.2.1 Code Generated in the JSF Page

Example 15–6 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 `LoggedInUser` 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 15–6 Code Generated in the JSF Page for a Databound ADF Faces Tree Table

```
<h:form>
  <af:treeTable value="#{bindings.LoggedInUser.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.

15.5.2.2 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 15.4.2.2, "Binding Objects Defined in the Page Definition File".

15.5.3 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 attribute specified in the node rule defined in the page definition file. This accessor attribute is invoked in response to a disclosure event (for example, the user expands a node) and returns the collection that populates that node.

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

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

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

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

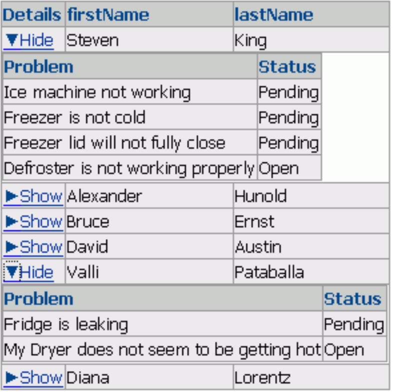
15.6 Using an Inline Table to Display Detail Data in a Master Table

As you may recall from Section 14.5, "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 14.5, "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 15–9 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 15–9 Inline Table Displaying Information from a Detail Collection



15.6.1 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 a view link accessor attribute that returns the detail collection that will populate the inline table. For information about view links accessors, see Section 5.10.2, "How to Create Master/Detail Hierarchies Using View Links".

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.
2. In the context menu, choose **Tables** > **ADF Master Table, Inline Detail Table**.

   JDeveloper displays the Tree Binding Editor (previously shown in Figure 15–5).
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 attribute that returns the detail collection that you want to appear in the inline detail table. The list displays only the accessor attributes 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.

  View link accessor attributes, which return data collections, are generated when you create view links. The **Branch Rule Accessor** field displays all view link accessors that return detail collections for the master collection selected in the **Data Collection Definition** field. For more information about view objects and view links, see Chapter 5, "Querying Data Using View Objects".

  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 15–6, 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

15.6.2 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 12.2.3, "What Happens When You Use the Data Control Palette".

15.6.2.1 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 15–7 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 main table is defined the same as any other ADF databound table. It is bound to the LoggedInUser 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 LoggedInUser 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 attribute that returns the detail collection.

Example 15–7 JSF Code Created for the Master Table with an Inline Detail Table

```
<af:table rows="#{bindings.LoggedInUser.rangeSize}"
          emptyText="#{bindings.LoggedInUser.viewable ? \'No rows yet.\' :
                       \'Access Denied.\'}"
          var="row" value="#{bindings.LoggedInUser.treeModel}">
  <af:column headerText="#{bindings.LoggedInUser.labels.FirstName}"
             sortable="false" sortProperty="FirstName">
    <af:outputText value="#{row.FirstName}"/>
  </af:column>
  <af:column headerText="#{bindings.LoggedInUser.labels.LastName}"
             sortable="false" sortProperty="LastName">
    <af:outputText value="#{row.LastName}"/>
  </af:column>
  <f:facet name="detailStamp">
    <af:table rows="#{bindings.LoggedInUser.rangeSize}"
              emptyText="No rows yet." var="detailRow"
              value="#{row.children}">
      <af:column headerText="#{row.children[0].labels.Status}"
                 sortable="false" sortProperty="Status">
        <af:outputText value="#{detailRow.Status}"/>
      </af:column>
      <af:column headerText="#{row.children[0].labels.ProblemDescription}"
                 sortable="false" sortProperty="ProblemDescription">
        <af:outputText value="#{detailRow.ProblemDescription}"/>
      </af:column>
    </af:table>
  </f:facet>
</af:table>
```

15.6.2.2 Binding Objects Defined in the Page Definition File

Example 15–8 shows the binding objects added to the page definition file for a master table with an inline detail table. The executables element defines an iterator binding named LoggedInUserIterator, which displays data from the LoggedInUser collection in the main table. No iterator binding is needed for the detail collection, because the accessor attribute referenced in the tree binding object returns the detail data that is related to the currently selected master data.

In the bindings element, 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 15.4.2, "What Happens When You Create ADF Databound Trees".

Example 15–8 Binding Objects Added to the Page Definition File for a Master Table with an Inline Detail Table

```
<executables>
  <iterator id="LoggedInUserIterator" RangeSize="10" Binds="LoggedInUser"
            DataControl="SRService"/>
</executables>
<bindings>
  <tree id="LoggedInUser" IterBinding="LoggedInUserIterator">
    <AttrNames>
      <Item Value="UserId"/>
      <Item Value="Email"/>
      <Item Value="FirstName"/>
      <Item Value="LastName"/>
    </AttrNames>
    <nodeDefinition DefName="oracle.srdemo.model.queries.LoggedInUser"
                    id="LoggedInUserNode">
      <AttrNames>
        <Item Value="FirstName"/>
        <Item Value="LastName"/>
      </AttrNames>
      <Accessors>
        <Item Value="ServiceRequestsByStatus"/>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition DefName="oracle.srdemo.model.queries.ServiceRequestsByStatus"
                    id="ServiceRequestsByStatusNode">
      <AttrNames>
        <Item Value="Status"/>
        <Item Value="ProblemDescription"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

15.6.3 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 attribute specified in the node rule defined in the page definition file. This accessor attribute is invoked in response to a `DisclosureEvent` event. For example, if the user clicks on the **Show** link, the accessor attribute is invoked to populate the data in the inline table.

16

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 16.1, "Introduction to Page Navigation"
- Section 16.2, "Creating Navigation Rules"
- Section 16.3, "Using Static Navigation"
- Section 16.4, "Using Dynamic Navigation"

For information about how to create dynamic navigation menus, see Chapter 19, "Using Complex UI Components".

16.1 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

16.2 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

16.2.1 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.

16.2.1.1 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 16–1.

Example 16–1 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 16.2.2, "What Happens When You Create a Navigation Rule".)

16.2.1.2 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.

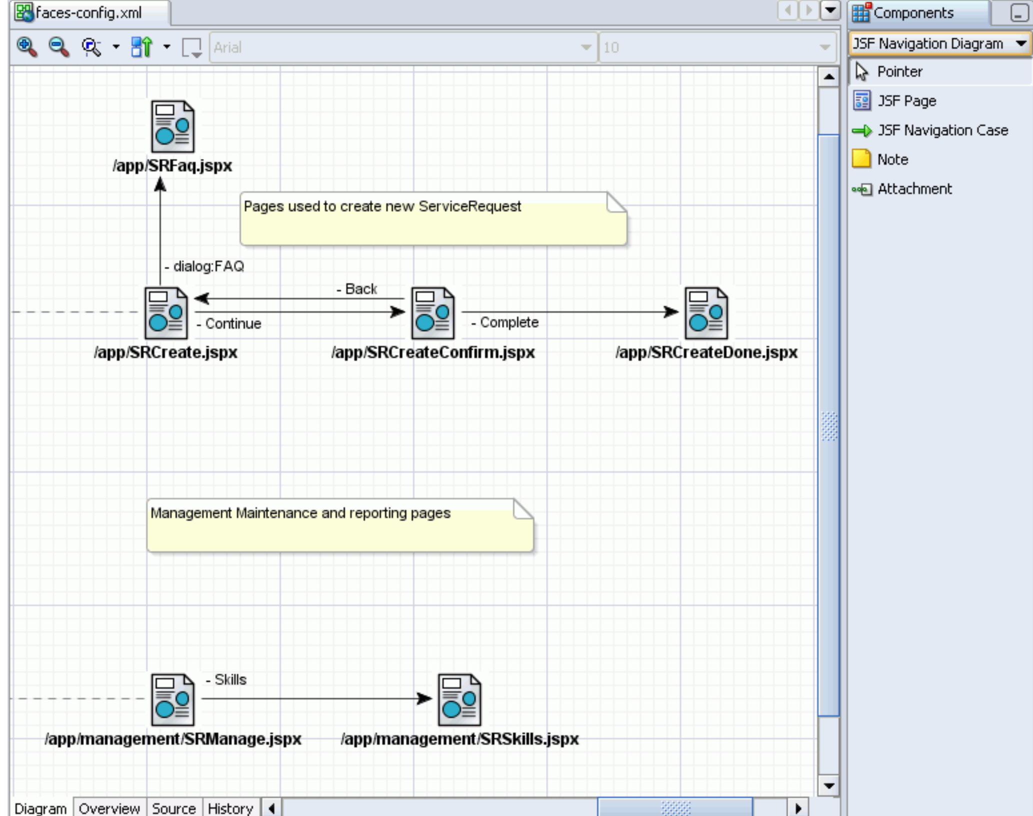
To define a navigation rule using the JSF Navigation Modeler:

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 16–1.

   Notice that the Component Palette automatically displays the JSF Navigation Modeler components.

Figure 16–1 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.

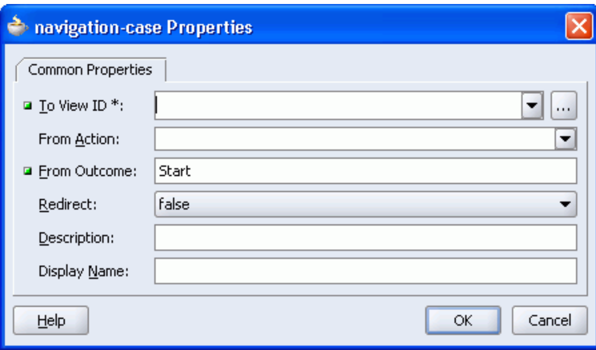
   JDeveloper draws the navigation case on the diagram as a solid line ending with an arrow between the two pages, as shown in Figure 16–2.

Figure 16–2 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 16–3.

Figure 16–3 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 16.2.1.1, "About Navigation Rule Elements".

16.2.1.3 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 16–2 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 16–2 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 16–3 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 16–3 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 16–4 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 16–4 Default Navigation Rule

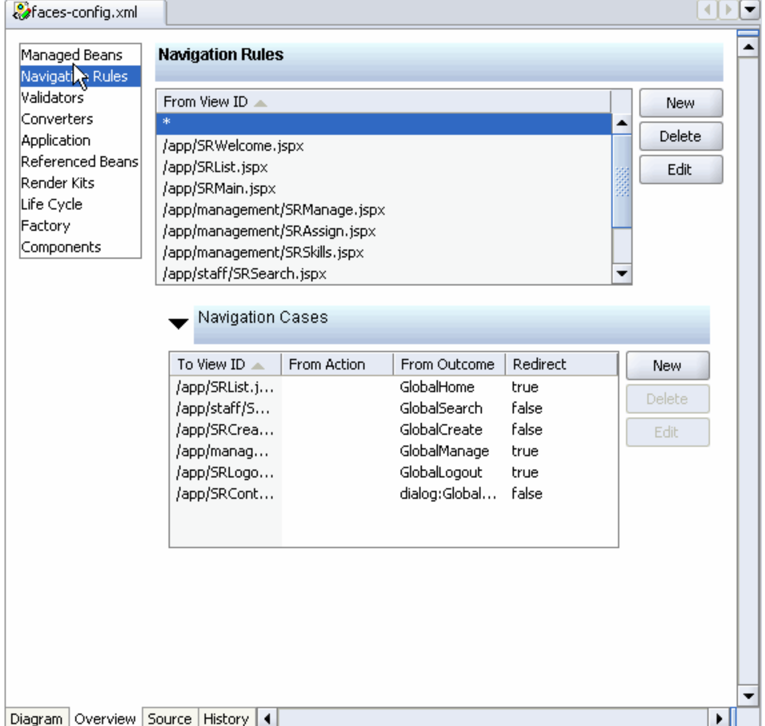
```
<navigation-rule>
  <navigation-case>
    <to-view-id>home.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

- 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 16–4.

Figure 16–4 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 16.2.1.1, "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.

16.2.2 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 16–5 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 19.3, "Using Popup Dialogs".

Example 16–5 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 16–6 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 16–6 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>
```

16.2.3 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 13.4.4, "What You May Need to Know About the Browser Back Button".

16.2.4 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.

16.2.4.1 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 11, "Getting Started with ADF Faces".)

16.2.4.2 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.

16.2.4.3 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 16–7 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 16–7 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>
```

16.2.4.4 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 16–8. When these rules are evaluated, the more specific navigation cases are used first, then the more general case.

Example 16–8 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>
```

16.2.5 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.

16.3 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 16–9, 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 16–10. In this case, when the user clicks the button, the navigation case causes the ConfirmAction page to be displayed.

Example 16–9 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 16–10 Static Navigation Button Defined in a JSF Page

```
<af:commandButton text="Continue" action="Confirm"/>
```

16.3.1 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 17.3, "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 17.4, "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.

16.3.2 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 16–11 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 16–12. The navigation case belongs to a global navigation rule that applies to all pages in the application.

Example 16–11 Navigation Component That Specifies a Static Outcome Value

```
<af:commandLink text="#{res['srdemo.about']}" action="GlobalAbout"
                              immediate="true"/>
```

Example 16–12 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.

16.4 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 17.5, "Overriding Declarative Methods".

16.4.1 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 11.5, "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.

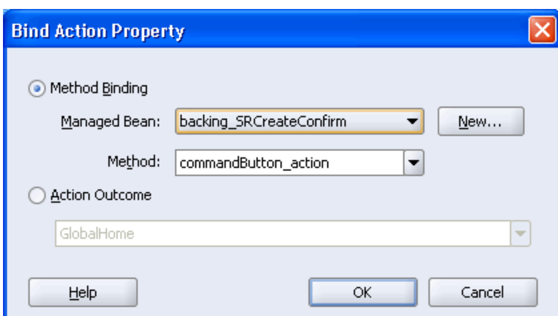
To create a navigation component that binds to a backing bean:

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 16–5.

Figure 16–5 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 16–13. If it is an existing method, the source editor displays that method, instead of the stub method.

Example 16–13 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.

16.4.2 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 16–14 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 16–14 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 16–15.

Example 16–15 Backing Bean Method That Returns a Dynamic Outcome

```
public String createSRButton_action() {
  BindingContainer bindings = getBindings();
  OperationBinding operBinding =
            bindings.getOperationBinding("createServiceRequest");
  Integer newServiceRequestId = (Integer)operBinding.execute();
  //Put the number of the created service ID onto the request as an
  // example of passing data in that way
  JSFUtils.setRequestAttribute("SRDEMO_CREATED_SVRID",newServiceRequestId);
  return "Complete";
}
```

The backing bean 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 17.5, "Overriding Declarative Methods".

Example 16–16 shows the navigation rule that handles the outcome returned by the backing bean.

Example 16–16 Navigation Rule Referenced by a Dynamic Outcome

```
<navigation-rule>
    <from-view-id>/SRCreateConfirm.jspx</from-view-id>
    ...
    <navigation-case>
      <from-outcome>Complete</from-outcome>
      <to-view-id>/SRCreateDone.jspx</to-view-id>
    </navigation-case>
</navigation-rule>
```

16.4.3 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.

16.4.4 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 16–17 shows the navigation cases for this scenario.

Example 16–17 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.

16.4.5 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 16–18.

Example 16–18 Navigation Button with Action Listener and Action Methods

```
<h:commandButton image="californiastate.jpg"
                 actionListener="#{someBean.someListenmethod}"
                 action="#{someBean.someActionmethod}"/>
```

16.4.6 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 16–19.

Example 16–19 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.

17

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 17.1, "Introduction to More Complex Pages"
- Section 17.2, "Using a Managed Bean to Store Information"
- Section 17.3, "Creating Command Components to Execute Methods"
- Section 17.4, "Setting Parameter Values Using a Command Component"
- Section 17.5, "Overriding Declarative Methods"

17.1 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 store information on a managed bean, or the ability to override declarative actions. ADF provides many features that allow you to add this complex functionality using very little actual code.

For example, the **Delete** button on the SRMain page was created by dragging the `Delete` operation for the `ServiceRequest` collection and dropping it as a command button. Then that `Delete` operation was overridden, so that when the button is clicked, it not only deletes the service request from the cache, but also commits the transaction. Additionally, the SRMain page can be accessed from a number of pages in the SRDemo application. The `userState` managed bean (`UserSystemState.java`) holds the value of the originating page. The overridden `Delete` operation also contains logic to access the value of the originating page so that the user navigates successfully off the SRMain page.

Read this chapter to understand:

- How to create an use a managed bean to store parameter values or flags
- How to create command components that will invoke custom methods on your application module
- How to set parameter values
- How to add logic to an operation or custom method bound to a command component

17.2 Using a Managed Bean to Store Information

Often, pages require information from other pages. 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.

> **Tip:** Use managed beans to store only "bookeeping" information. All application data and processing should be handled by logic in the business layer of the application.

For example, the SRMain page needs to know the page from which the user navigated from in order to return the user to the correct page. The SRDemo application has a managed bean that holds this information, allowing the sending page to set the information, and the SRMain page to use the information in order to determine where to navigate.

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.

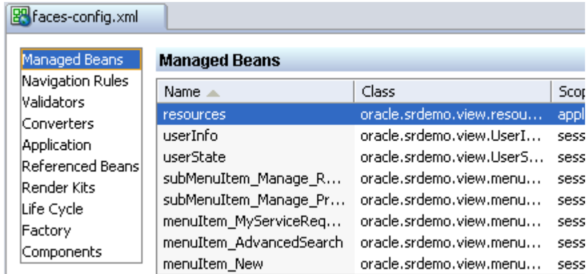
17.2.1 How to Use a Managed Bean to Store Information

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:

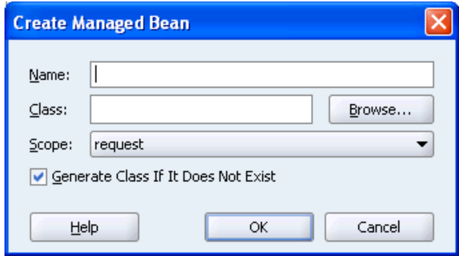
1. Open the faces-config.xml file. This file is stored in the <project_ name>/WEB-INF directory.
2. At the bottom of the window, select the **Overview** tab.
3. In the element list on the left, select **Managed Beans**. Figure 17–1 shows the JSF Configuration Editor for the faces-config.xml file.

Figure 17–1 The JSF Configuration Editor



4. Click the **New** button to open the Create Managed Bean dialog, as shown in Figure 17–2. Enter the name and fully qualified class path for the bean. Select a scope, select the **Generate Java File** checkbox, and click **OK**.

Figure 17–2 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 17.5, "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.

17.2.2 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 17–1 shows the code added to the `MyBean` class stored in the view package.

Example 17–1 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 17–2 shows the `managed-bean` element created for the `MyBean` class.

Example 17–2 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.

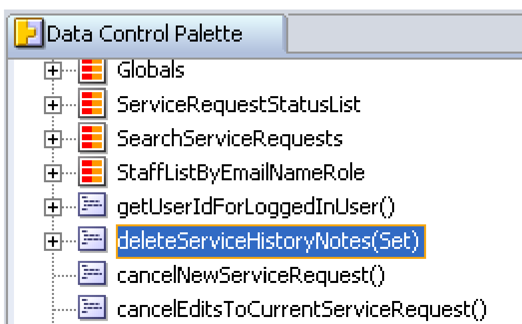
- Section 17.4, "Setting Parameter Values Using a Command Component"
- Section 17.5, "Overriding Declarative Methods"
- Section 18.5, "Conditionally Displaying the Results Table on a Search Page"

17.3 Creating Command Components to Execute Methods

When your application contains custom methods, these methods appear in the Data Control Palette. You can then drag these methods and drop them as command buttons. When a user clicks the button, the method is executed. For more information about custom methods, see Chapter 8, "Implementing Business Services with Application Modules".

For example, the SRService application module in the SRDemo application contains the `deleteServiceHistoryNotes(Set keySet)` method. This method ensures that one or more rows are selected, then deletes those rows and commits the transaction. To allow the user to execute this method, you drag the `deleteServicehistoryNotes(Set)` method from the Data Control Palette, as shown in Figure 17–3.

Figure 17–3 Methods in the Data Control Palette



Note: In the SRDemo application, additional view-layer logic is added to the `deleteServiceHistoryNotes(Set)` method by overriding this method in a managed bean. You can also override a custom method if you need to provide conditional navigational logic. For more information, see Section 17.5, "Overriding Declarative Methods".

17.3.1 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 `deleteServicehistoryNotes(Set)` method, you need to specify the set of rows to be deleted.

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.

17.3.2 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 13.4.2.3, "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.

17.3.2.1 Using Parameters in a Method

When you drop a method that takes parameters onto a JSF page, JDeveloper creates a method action binding. This binding is what causes the method to be executed when a user clicks the command component. When the method requires parameters to run, JDeveloper also creates `NamedData` elements for each parameter. These elements represent the parameters of the method.

For example, the deleteServiceHistoryNotes method action binding contains a NamedData element for the Set parameter. This element is bound to the value specified when you created the action binding. Example 17–3 shows the method action binding created when you drop the deleteServiceHistoryNotes(Set) method, and bind the Set parameter (named keySet) to the keySet property of the selectionState property of the history table UI component in the page's backing bean (for more information about using this method to delete multiple rows (the key set), see Section 14.6.5, "How to Use the TableSelectMany Component in the Selection Facet").

Example 17–3 Method Action Binding for a Parameter Method

```
<methodAction id="deleteServiceHistoryNotes"
              InstanceName="SRService.dataProvider"
              DataControl="SRService"
              MethodName="deleteServiceHistoryNotes"
              RequiresUpdateModel="true" Action="999">
  <NamedData NDName="keySet"
             NDValue="${backing_SRMain.historyTable.selectionState.keySet}"
             NDType="java.util.Set"/>
</methodAction>
```

17.3.2.2 Using EL Expressions to Bind to Methods

Like creating command buttons using operations, when you create a command button using a method, JDeveloper binds the button to the method using the actionListener attribute. The button is bound to the execute property of the action binding for the given method. This binding causes the binding's method to be invoked on the application module. For more information about the command button's actionListener attribute, see Section 13.4.3, "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 17.5, "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 17–4 shows the EL expression used to bind the command button to the deleteServiceHistoryNotes(Set) method.

Example 17–4 JSF Code to Bind a Command Button to a Method

```
<af:commandButton actionListener="#{bindings.deleteServiceHistoryNotes.execute}"
                  text="deleteServiceHistoryNotes"
                  disabled="#{!bindings.deleteServiceHistoryNotes.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.

17.3.3 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 `deleteServiceHistoryNotes(Set)` method, the method takes the value of the key set (in this case the selected rows in a table) and deletes them from the data source.

17.4 Setting Parameter Values Using a Command Component

There may be cases where an action on one page needs to set parameters that will be used to determine application functionality. For example, the results table on the SRSearch page will display only if the value of the `searchFirstTime` flag on the `userState` managed bean is `false`. When this bean is instantiated as the search page is rendered, the `isSearchFirstTime` method on the bean checks that parameter. If it is `null` (which it will be the first time the page is rendered), the bean sets the value to `true`.

A `setActionListener` component, which is nested in the command button used to execute this search, is then used to set the searchFirstTime flag to `false`, thus causing the results table to display once the search is executed. For information about using managed beans, see Section 17.2.1, "How to Use a Managed Bean to Store Information".

17.4.1 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 17.2, "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

17.4.2 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, or the actual value, the component will be able to access that value. When you set the `To` attribute to a target, the command component is able to set the value on the target. Example 17–5 shows the code on the JSF page for a command component that takes the value `false` and sets that as the value of the `searchFirstTime` flag on the `userState` managed bean.

Example 17–5 JSF Page Code for a Command Button Using a setActionListener Component

```
<af:commandButton actionListener="#{bindings.Execute.execute}"
                  text="#{res['srsearch.searchLabel']}">
  <af:setActionListener from="#{false}"
                             to="#{userState.searchFirstTime}"/>
</af:commandButton>
```

17.4.3 What Happens at Runtime

When a user clicks the command component, before navigation occurs, the `setActionListener` component sets the parameter value. In Example 17–5, the `setActionListener` takes the value `false` and sets it as the value for the `searchFirstTime` attribute on the `userState` managed bean. Now, any component that needs to know this value in determining whether or not to render, can access it using the EL expression `#{userState.searchFirstTime}`. For the complete example Section 18.5, "Conditionally Displaying the Results Table on a Search Page".

17.5 Overriding Declarative Methods

When you drop an operation or method as a command button, JDeveloper binds the button to the `execute` method for the operation or method. However, there may be occasions when you need to add logic before or after the logic executes. For example, after the `Delete` operation successfully executes, you may want to execute the `Commit` operation, so that the user does not have to click another button.

JDeveloper allows you to add logic to a declarative operation by creating a new method and property on a managed bean that provide access to the binding container. By default, this generated code executes the operation or method. You can then add logic before or after this code. JDeveloper automatically binds the command component to this new method instead of the `execute` property on the original operation or method. 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 set the service request's ID as a parameter value on the `JSFUtils` helper class after the method is executed.
- `SRMain.java`: The `onDeleteRequest` method overrides the `Delete` operation. It first executes the `Delete` action. If any errors occur, it redisplays the page. If there are no errors, it executes the `Commit` operation on the application module. If any errors occur, it redisplays the page. If there are no errors, the user is returned to the page that is held as the `ReturnNavigationRule` value on the `UserState` managed bean.

17.5.1 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 operation or 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 17.3, "Creating Command Components to Execute Methods".

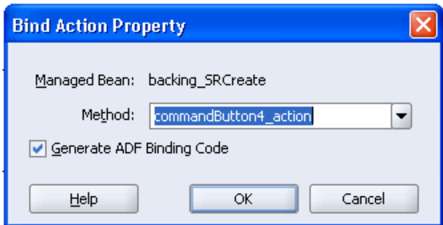
   For more information about creating command components from operations, see Section 13.4.2, "What Happens When Command Buttons Are Created Using the Data Control Palette"

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 17–4.

Figure 17–4 Bind Action Property Dialog for a Page with Auto-Binding Enabled



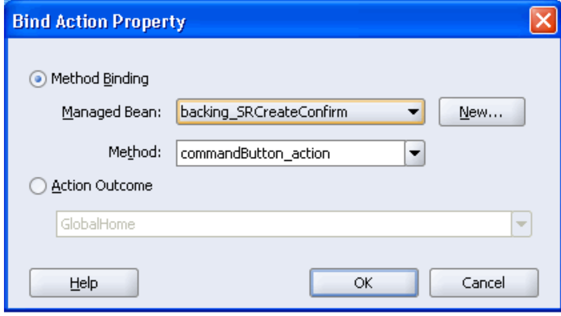
- To create a new method, enter a name for the method in the **Method** field, which initially displays a default name.

  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 17–5. For more information about auto-binding, see Section 11.5.4, "How to Use the Automatic Component Binding Feature".

Figure 17–5 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 17–6 shows the code inserted into the bean. In this example, a command button is bound to the Delete operation.

Example 17–6 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("Delete");
    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 17–7

Example 17–7 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`.

17.5.2 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 17–8 shows the code that JDeveloper adds to the chosen managed bean. Notice that the return `String "Complete"` was automatically added to the method, as that was the value of the `action` attribute.

Example 17–8 Generated Code in a Backing Bean to Access the Binding Object

```
private BindingContainer bindings;
...
public String createSRButton_action() {
    BindingContainer bindings = getBindings();
    OperationBinding operationBinding =
            bindings.getOperationBinding("createServiceRequest");
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
        return null;
    }

    return "Complete";
}
```

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 16.4, "Using Dynamic Navigation".

JDeveloper automatically rebinds the UI command component to the new method using the `Action` attribute, instead of the `ActionListener` attribute. For example, Example 17–9 shows the code on a JSF page for a command button created by dropping the `createServiceRequest` method. Notice that the `actionListener` attribute is bound to the `createServiceRequest` method, and the `action` attribute has a `String` outcome of `Complete`. 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 17–9 JSF Page Code for a Command Button Bound to a Declarative Method

```
<af:commandButton actionListener="#{bindings.createServiceRequest.execute}"
                  text="createServiceRequest"
                  disabled="#{!bindings.createServiceRequest.enabled}"
                  id="createSRButton"
                  action="Complete"/>
```

Example 17–10 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 17–10 JSF Page Code for a Command Button Bound to an Overridden Method

```
<af:commandButton text="createServiceRequest"
                  disabled="#{!bindings.createServiceRequest.enabled}"
                  id="createSRButton"
                  action="#{backing_SRCreateConfirm.createSRButton_action}"/>
```

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.

18 Creating a Search Form

This chapter describes how to create different types of search pages, for example search forms that work similarly to Oracle Forms EnterQuery/ExecuteQuery searches, or forms that are more like search forms you find on web sites.

This chapter includes the following sections:

- Section 18.1, "Introduction to Creating Search Forms"
- Section 18.2, "Creating a EnterQuery/ExecuteQuery Search Form"
- Section 18.3, "Creating a Web-type Search Form"
- Section 18.4, "Creating Search Page Using Named Bind Variables"
- Section 18.5, "Conditionally Displaying the Results Table on a Search Page"

18.1 Introduction to Creating Search Forms

You can create a search form that allows users to enter criteria into a form based on known attributes of an object. The criteria is then constructed into a query, and a query-by-example (QBE) search is executed. All records whose attributes match the entered criteria are returned and can then be displayed in a table, either on a separate page or on the same page. To create a QBE search from, you drop a collection from the Data Control palette as an ADF Search Form. When you do this, the input text components do not have any associated validators or converters, which then allows the end user to enter search criteria that might otherwise not pass validation or conversion, such as entering `> 1500` into a number field (for more information about conversion and validation, see Chapter 20, "Using Validation and Conversion").

In addition to iterating over collections, an iterator binding provides QBE capability by also being able to iterate over a collection of QBE criteria rows. These rows are created by the criteria entered by the user. Each criteria row has the same attribute structure as a row in its related data collection, except that the attribute values are all treated as a `String` data type. This data type allows the user to enter in the form fields query criteria containing comparison operators and wildcard characters.

For the iterator binding to have this capability, the binding must be set to `Find` mode. When an iterator binding is set to work in `Find` mode, the binding iterates over the these criteria rows instead of the standard collection data. The `Execute` operation is then used to execute the query against the collection. The `Execute` operation applies the user's query criteria and also disables the `Find` mode for the iterator binding, allowing the form to display data returned from the collection, as opposed to criteria. For more information about the Find mode, see Section 10.5.3, "How to Use Find Mode to Implement Query-by-Example".

By default, an ADF Search form contains command buttons bound to the `Find` and `Execute` operations. These button are useful if you want to be able to use the form to both search for records and view the current returned record. The user clicks the **Find** button to put the iterator in `Find` mode, thus making the input text fields available for criteria. When the user then clicks the **Execute** button, the criteria is used to create the query, the search is executed, and the form can then display the results. This type of search form is similar to how Oracle Forms EnterQuery/ExecuteQuery searches work. For more information on creating this type of search form, see Section 18.2, "Creating a EnterQuery/ExecuteQuery Search Form".

You can also force the iterator to always be in `Find` mode, thus negating the need for the **Find** button and allowing the user to view both their search criteria and the results. The user sees only a button bound to the `Execute` operation. When the user clicks that button, the iterator is toggled out of `Find` mode. To have it programatically set back to `Find` mode so that the user can always enter criteria, you insert an action binding into the page definition file that executes whenever the iterator is not in `Find` mode (for example after the `Execute` operation is invoked). However, this means that the results must be displayed in a separate form or table, as the iterator for the search form will always be in `Find` mode. This type of search is how a typical web page search works. For more information, see Section 18.3, "Creating a Web-type Search Form".

For a this type of web search, you can have the search form and results table on separate pages, or you can have the search form and the results table on the same page. However, when the search and results are on the same page, there must be one iterator that is always in `Find` mode for the search form, and a separate iterator for the results. See Section 18.3.4, "About Creating Search and Results on the Same Page" for more information. The SRSearch page uses this QBE search functionality, and displays the search form and results table on the same page, as shown in Figure 18–1.

Figure 18–1 Search and Results on the Same Page

You can also create a quick search form using named bind variables from a view object created just for the search. For example, the SRStaffSearch page in the SRDemo application uses the `StaffListByEmailNameRole` view object to create the search form. Instead of dropping the collection as an ADF Search Form, the `ExecuteWithParams` operation is dropped as a parameter form. When this search is executed, instead of building a QBE query dynamically each time the search is executed, the parameter form uses the view object's design-time SQL statement to execute the query. The text of the SQL statement never changes, so the database can reuse it efficiently, providing increased performance.

By contrast, in a QBE query the WHERE clause predicate is generated dynamically to match your search criteria. So if you keep searching with different combinations of criteria, the text of your SQL statement for that view object changes with the different executions. For example, three different searches might generate the following three different WHERE clauses for their SQL statements:

- AND (SVR_ID = 101)
- AND (SVR_ID = 102
- AND (PROBLEM_DESCRIPTION LIKE '%FOO%' AND ...)

Because the text of the SQL statement potential changes for a QBE search, the view object creates a new prepared statement each time, and the query must be reparsed on the database side. For this reason, when a search is expected to be frequently executed with the same statement, you may want to use a parameter search form for enhanced performance.

However, in order to create a parameterized search, you create a specific view object for the search. If you need to work programmatically with the rows using their generated custom row interfaces, it might not be practical if the view object for the search requires exactly the same structure and behavior as an existing view object. For example, if you create two view objects, `ServiceRequests` and `ServiceRequestSearch`, both with the same structure, and you need to edit the client code to get typesafe access to attributes, you'll need to edit both the `ServiceRequestsRow` and `SearchRequestsSearchRow` client interfaces. For this reason, you should use a QBE type search when you the query will require a view object that is the same as an existing view object, and the expected performance savings from a parameterized search will not be great.

In the SRDemo application, the SRSearch page uses a QBE query to find service requests based on the ID, status, problem description, and product name from the `ServiceRequests` view object. This search page uses an instance of the `ServiceRequests` view object named `SearchServiceRequests`, which is instantiated specifically for the search page by the application module. Another instance of the `ServiceRequests` view object, the `ServiceRequestsByStatus` instance, is used by the SRList page to show a filtered list of service requests by status.

For more information about view objects, view object instances, and named bind variables, see Chapter 5, "Querying Data Using View Objects"

18.2 Creating a EnterQuery/ExecuteQuery Search Form

You can create a form that allows users to enter search criteria, execute the search, and then review the results of the search in the same form. However, you will need to create two forms that will act as one: one for the search form, and one for the results.

To do this, you set the form used to enter the search criteria to display only when the iterator is in `Find` mode. You set the second form, which displays the results, to display only when the iterator is not in `Find` mode. You control the display of each using an EL expression for the `Rendered` attribute of the forms. This allows the user to enter in a criteria like `> 1500` into a number field when the form is in `Find` mode, but the same entry will raise an error when the user is editing data.

18.2.1 How to Create an EnterQuery/ExecuteQuery Search Page

As stated above, you actually create two forms, one for the query and one for the results. EL expressions are used to determine when to display the correct form.

1. From the Data Control Palette, drag a collection and from the context menu, select **Forms** > **ADF Search Form**.

   For example, if you want the query to execute over all service requests, you would drag the `ServiceRequests` collection.

2. Drag the same collection, but this time, drop it as an **ADF Form**. Select **Include Navigation Controls** in the Edit Form Fields dialog. This will be the form that displays the results. The navigation controls will allow the user to navigate through the objects returned from the query.

3. In the Structure window, select the `panelForm` component for the first form.

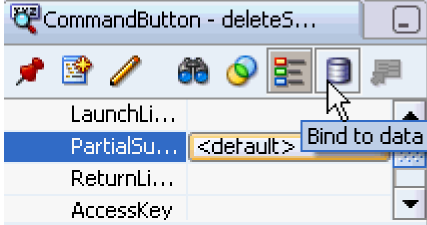
4. In the Property Inspector, in the **Rendered** field, enter the following expression, replacing <*iterator*> with the name of the iterator:

   ```
   #{bindings.<iterator>.findMode == true}
   ```

   This will cause the form to be rendered whenever the iterator is in `Find` mode.

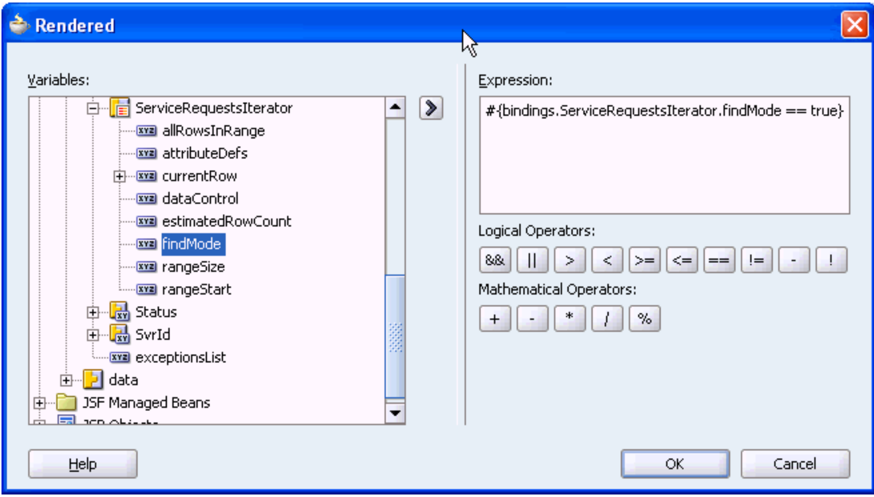
   If you are unsure of the name, you can use the Bind to Data tool to open the Expression Builder, as shown in Figure 18–2.

Figure 18–2 Clicking to Open the Bind to Data Tool



   In the **Variables** tree of the Expression Builder, select the `findMode` property under the correct iterator, and click the right arrow to add it to the **Expression** pane. You then add the logical operator and set the expression to check for `true`. Figure 18–3 shows the expression that would be built for the `ServiceRequestsIterator` iterator.

Figure 18–3 Expression Builder Used to Set the Rendered Attribute



5. In the Structure window, select the panel form for the second form. Set the `Rendered` property to:

```
#{bindings.<iterator_name>.findMode == false}
```

   This expression will cause this form to be displayed whenever the iterator is not in `Find` mode.

6. From the Data Control Palette, drag the `Find` operation associated with the collection and drop it as an **ADF Command Button** next to the navigation buttons in the second form. This button allows the user to toggle back into `Find` mode. Doing so will display the first form, so that the user can execute another query.

7. Rename the **Find** and **Execute** buttons to something more meaningful for the user. For example, you might rename the **Find** button in the top form to **Cancel Query**, since clicking this button will cause the iterator to toggle out of `Find` mode. You might rename the **Execute** button in the top form to **Execute Query**. Lastly, you might rename the bottom **Find** button to **Enter Query**, as that button will be used to place the iterator into Find mode, thereby causing the top form to display.

18.2.2 What Happens When You Create a Search Form

When you drop a search form onto a page, JDeveloper includes a command button bound to the `Find` operation and a command button bound to the `Execute` operation, as shown in Example 18–1. The `Find` operation places the associated iterator into `Find` mode. The `Execute` operation executes the query and places the iterator out of `Find` mode.

Example 18–1 Page Definition Code for Find and Execute Operations

```
<action id="Find" RequiresUpdateModel="true" Action="3"
            IterBinding="ServiceRequestsIterator"/>
<action id="Execute" RequiresUpdateModel="true" Action="2"
            IterBinding="ServiceRequestsIterator"/>
```

The following table shows the built-in search operations, along with the result of invoking the operation.

Table 18–1 Search Built-in Operations

Operation	Action Attribute Value	When invoked, the associated iterator binding will...
`Find`	3	Places the associated iterator into `Find` mode, allowing it to iterate over criteria instead of data.
`Execute`	2	Applies the criteria and executes the query using the criteria from the iterator when in `Find` mode. Toggles the associated iterator out of `Find` mode, so that the iterator can work with the results.

Additionally when you drop an ADF Search Form, JDeveloper adds an `outputText` component inside the `panelGroup` component that holds the **Find** and **Execute** buttons. This `outputText` component displays the words **Find Mode** whenever the iterator is in `Find` mode. This is used mostly for development purposes, allowing you to be able to easily determine when the iterator is in `Find` mode. You can safely delete this component.

18.3 Creating a Web-type Search Form

For a standard web-type search form, you may not want the user to have to manually place the form into `Find` mode. Instead, you may want them to simply enter the search criteria and then execute the search. To do this, you configure the iterator so that it is always in `Find` mode. You can then either have the results displayed on a separate page, or you can display them on the same page. If you have them displayed on the same page, you need to create a separate iterator for the results set that is not continually in `Find` mode.

18.3.1 How to Create a Search Form and Separate Results Page

To create a search form that is always in find mode, you drop the collection you are searching against as an **ADF Search Form**, and set a condition on the iterator that keeps it in `Find` mode. You then drop the same collection as a table or form on another page to display the results.

To create search and results on separate pages:

1. From the Data Control Palette, drag a collection and from the context menu, select **Forms** > **ADF Search Form**.

   For example, if you want the query to execute over all service requests, you would drag the `ServiceRequests` collection.

2. On another page, drag the same collection, but this time drop it as any type of table or form.

3. On the **Execute** button, create the navigation between the search form and the results page. This will allow the user to navigate to the results page as the query is done. For more information about creating navigation, see Chapter 16, "Adding Page Navigation".

   At this point, when the search form on the page renders, it displays with values in the text boxes from the first record. Users must click the **Find** button in order to set the iterator into `Find` mode and enter search criteria. When the user then clicks the **Execute** button, the user navigates to the results page, whose table displays the results from the query. If the user navigates back to the search form, it displays the attribute values of the first record in the results set.

You can follow the next set of procedures to eliminate the need for the Find button.

To set the iterator automatically in Find mode:

Follow these steps to automatically put the search form's iterator into Find mode:

1. Open the page definition file for the search page.
2. In the Structure Pane, right-click on the **Executables** node and choose **Insert inside executables** > **invokeAction**.
3. In the Insert invokeAction dialog, enter an ID for the action, such as `AlwaysFind`. From the Binds drop-down list, select **Find**. Do NOT click OK or close the dialog.
4. In the Insert invokeAction dialog, select the **Advanced Properties** tab.
5. For **Refresh Condition**, enter the following EL expression, which tells the application to invoke this Find action whenever the page is not in `Find` mode. Replace <*iterator*> with the name of the iterator:

   ```
   ${bindings.<iterator>.findMode == false}
   ```

6. In the search JSF page, delete the **Find** button. Doing this only deletes the component from the JSF page. The binding still exists in the page definition because it is being referenced by the EL expression on the `RefreshCondition`.

18.3.2 What Happens When You Create A Web-type Search Form

When you drop a search form onto a page, JDeveloper includes a command button bound to the `Find` operation and a command button bound to the `Execute` operation. For details, see Section 18.2.2, "What Happens When You Create a Search Form".

You use `invokeActions` to invoke an operation implicitly. For example, in a search page, instead of needing to have the user click the **Find** button, you can invoke this operation at a defined time or when a defined condition evaluates to `true`.

The `RefreshCondition` attribute provides a condition for invoking the action. The `Refresh` attribute determines at what point in the ADFm lifecycle the action should be invoked, when the `RefreshCondition` evaluates to `true`. For a web-type search page, you use an EL expression that evaluates to a condition that says to invoke the `Find` action whenever the iterator is not in `Find` mode. For example the page definition for the SRSearch page has the following entries:

Example 18–2 Page Definition Code for a Find InvokeAction and Related Binding

```
<executables>
  <invokeAction id="AlwaysFind" Binds="Find" Refresh="ifNeeded"
             RefreshCondition=
               "${bindings.SearchServiceRequestsIterator.findMode == false}"/>
  <iterator id="SearchServiceRequestsIterator" RangeSize="10"
            Binds="SearchServiceRequests" DataControl="SRService"/>
</executables>
    ...
<bindings>
    ...
   <action id="Find" RequiresUpdateModel="true" Action="3"
           IterBinding="SearchServiceRequestsIterator"/>
...
</bindings>
```

18.3.3 What You May Need to Know

By creating a condition that places the iterator in continuous `Find` mode, any other binding that references that iterator will also be referencing the iterator when it is in `Find` mode. Therefore, if you want to have other components on the page that also need to reference that iterator, but do not need it to be in `Find` mode, you must create a separate iterator based on the same collection. This is how you are able to place the search results on the same page as the search form. See the next section Chapter 18.3.4, "About Creating Search and Results on the Same Page" for details.

18.3.4 About Creating Search and Results on the Same Page

As stated above, when you place the iterator into `Find` mode, it is in `Find` mode for all associated bindings. When you drop the same collection as a table to display the search results, that table's binding uses the same iterator as the search form. Because that means the table component would be bound to an iterator binding in `Find` mode, it will display the current query-by-example view criteria rows, as shown in Figure 18–4, instead of the actual data until you hit **Execute** and the iterator is taken out of `Find` mode.

Figure 18–4 The Table Displays the Criteria Rows When in Find Mode

To avoid this, you must create a second iterator for the table that is not in `Find` mode, but instead displays the collection of data that meets the search criteria.

18.3.5 How To Create Search and Results on the Same Page

To create a page that has both a search form and results table on the same page, you follow the procedures for when they are on separate pages, except you must create a separate iterator for the results table.

To create a search form and results table on the same page:

1. From the Data Control Palette, drag a collection and from the context menu, select **Forms** > **ADF Search Form**.

   For example, if you want the query to execute over all service requests, you would drag the `ServiceRequests` collection.

2. Drag the same collection, but this time drop it as any type of table.

3. Open the associated page definition file.

4. In the Structure Pane, right-click on the **Executables** node and choose **Insert inside executables** > **invokeAction**.
5. In the Insert invokeAction dialog, enter an ID for the action, such as `AlwaysFind.` From the Binds drop-down list, select **Find**. Do NOT click OK or close the dialog.
6. In the Insert invokeAction dialog, select the **Advanced Properties** tab.
7. For **RefreshCondition**, enter the following EL expression, which tells the application to invoke this action whenever the page is not in `Find` mode. Replace *<iterator>* with the name of the iterator:

```
${bindings.<iterator>.findMode == false}
```

> **Note:** The `invokeAction` must appear before the iterator, so that it is executed first.

8. In the Structure Pane, right-click on the **Executables** node and choose **Insert inside executables** > **iterator**.
9. Select the same collection used for the search form, and in the Iterator ID field enter a more meaningful name, such as **ResultsIterator** and click **OK**.
10. In the Structure Pane, expand the **bindings** node, right-click on the binding for the results table, and choose **Properties**.
11. In the Table Binding Editor dialog, make sure the correct collection is selected in the Data Collection column.
12. Select the newly created iterator from the Iterator drop-down list, ensure that all correct attributes are in the Display Attributes column, and click **OK**.
13. In the JSF page, delete the **Find** button.

    Doing this only deletes the component from the JSF page. The binding still exists in the page definition file.

18.3.6 What Happens When Search and Results are on the Same Page

In the above steps, you created a new iterator for the table. This iterator was created using the same collection, however it does not have a find operation associated with it. Therefore, it will always display the results of the search.

For example, the page definition for SRSearch page has the following entries:

Example 18–3 Two Iterators Used to Create a Search Form and Results Table

```
<executables>
  <invokeAction id="AlwaysFind" Binds="Find" Refresh="ifNeeded"
              RefreshCondition=
                "${bindings.SearchServiceRequestsIterator.findMode == false}"/>
...
  <iterator id="SearchServiceRequestsIterator" RangeSize="10"
              Binds="SearchServiceRequests" DataControl="SRService"/>
  <iterator id="SearchServiceRequestsResultsIterator" RangeSize="10"
              Binds="SearchServiceRequests" DataControl="SRService" />
. . .
</executables>
. . .
<bindings>
. . .
  <action id="Find" RequiresUpdateModel="true" Action="3"
          IterBinding="SearchServiceRequestsIterator"
. . .
  <table id="AllServiceRequests" IterBinding="AllServiceRequestsResultsIterator">
   . . .
  </table>
</bindings>
```

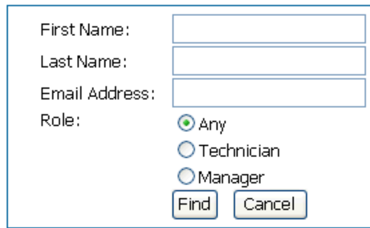
18.4 Creating Search Page Using Named Bind Variables

You can create a search form using a query from a view object that uses named bind variables to find matching objects. For example, the `StaffListByEmailNameRole` view object uses the following named bind variables:

- `TheFirstName`
- `TheLastName`
- `EmailAddress`
- `Role`

Figure 18–5 shows the SRStaffSearch search form, created using the `StaffListByEmailNameRole` view object. This page is used to find a staff user, given a first name, last name, email address, and role.

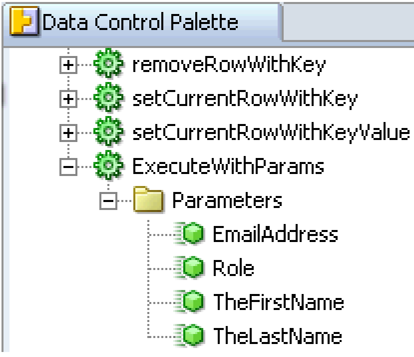
Figure 18–5 The SRStaffSearch Form



18.4.1 How to Create a Parameterized Search Form

Because a parameterized search uses variables that represent parameters, instead of using the `Execute` operation to invoke the query, you use the `ExecuteWithParams` operation. You create the search form by dropping `ExecuteWithParams` operation as a parameterized form. You then drop the corresponding collection as a table a table to display the results. Figure 18–5 shows the `ExecuteWithParams` operation for the `StaffListByEmailNameRole` collection.

Figure 18–6 The ExecuteWithParams Operation Uses Parameters



To create a search form and results table:

1. From the Data Control Palette, drag the `ExecuteWithParams` operation.
2. From the context menu, choose **Parameters** > **ADF Parameter Form.**
3. Use the Edit Form Fields dialog to change the display of the fields.
4. From the Data Control Palette, drag the corresponding collection, and drop it as any type of table or form.

18.4.2 What Happens When You Use Parameter Methods

When you drop the `ExecuteWithParams` operation as a parameter form, JDeveloper:

- Defines the following in the page definition file: variables to hold the data values, an action binding for the operation, and the attribute bindings for the associated attributes.
- Inserts code in the JSF page for the form using ADF Faces `inputText` components bound to the attribute bindings, and an ADF Faces `commandButton` component bound to the `ExecuteWithParams` operation. This code is the same as code for any other input form or command button.

Just as when you drop an operation such as `Execute` or `Next`, when you drop the `ExecuteWithParams` operation, JDeveloper creates an action binding. However, because the operation requires parameters to run, JDeveloper also creates `NamedData` elements for each parameter. These represent the named bind variables on the associated view object. Each `NamedData` element is bound to a value binding for the corresponding attribute. These bindings allow the operation to access the correct attribute's value for the parameter on execution.

For example, the action binding for the `ExecuteWithParams` operation on the `StaffListByEmailNameRole` collection contains a `NamedData` element for each of the named bind variables on the `StaffListByEmailNameRole` view object. The `EmailAddress NamedData` element is bound to the `StaffListByEmailNameRole_EmailAddres` attribute binding using an EL expression. Example 18–4 shows the action binding and some of the attribute bindings created when you drop the `ExecuteWithParameters` operation on the `StaffListByEmailNameRole` collection as a parameter form.

Example 18–4 Method Action Binding in the Page Definition File

```
<bindings>
  <action id="ExecuteWithParams"
          IterBinding="StaffListByEmailNameRoleIterator"
          InstanceName="SRService.StaffListByEmailNameRole"
          DataControl="SRService" RequiresUpdateModel="true" Action="95">
    <NamedData NDName="EmailAddress" NDType="java.lang.String"
               NDValue="${bindings.StaffListByEmailNameRole_EmailAddress}"/>
    <NamedData NDName="Role" NDType="java.lang.String"
               NDValue="${bindings.StaffListByEmailNameRole_Role}"/>
    <NamedData NDName="TheFirstName" NDType="java.lang.String"
               NDValue="${bindings.StaffListByEmailNameRole_TheFirstName}"/>
    <NamedData NDName="TheLastName" NDType="java.lang.String"
               NDValue="${bindings.StaffListByEmailNameRole_TheLastName}"/>
  </action>
  <attributeValues id="EmailAddress" IterBinding="variables">
    <AttrNames>
      <Item Value="StaffListByEmailNameRole_EmailAddress"/>
    </AttrNames>
  </attributeValues>
...
</bindings>
```

Because you dropped the `ExecuteWithParams` operation, the attributes reference a variable iterator that accesses the variables instead of a collection's iterator. This is because the operation (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 named bind variable. The variables are declared as children to the variable iterator, and are local, meaning they live only during a single request, and while they are carried across subsequent post-backs to the same form, they would be forgotten (and re-initialized) when a user navigates to some other page. Example 18–5 shows the variable iterator and variables created when dropping the `ExecuteWithParameters` operation on the `StaffListByEmailNameRole` collection. The variable iterator is used both by the form and by the button.

Example 18–5 Variable Iterator and Variables in the Page Definition File

```
<executables>
  <iterator id="StaffListByEmailNameRoleIterator" RangeSize="10"
            Binds="StaffListByEmailNameRole" DataControl="SRService"/>
  <variableIterator id="variables">
    <variableUsage DataControl="SRService"
                   Binds="StaffListByEmailNameRole.variablesMap.EmailAddress"
                   Name="StaffListByEmailNameRole_EmailAddress"
                   IsQueriable="false"/>
    <variableUsage DataControl="SRService"
                   Binds="StaffListByEmailNameRole.variablesMap.Role"
                   Name="StaffListByEmailNameRole_Role" IsQueriable="false"/>
    <variableUsage DataControl="SRService"
                   Binds="StaffListByEmailNameRole.variablesMap.TheFirstName"
                   Name="StaffListByEmailNameRole_TheFirstName"
                   IsQueriable="false"/>
    <variableUsage DataControl="SRService"
                   Binds="StaffListByEmailNameRole.variablesMap.TheLastName"
                   Name="StaffListByEmailNameRole_TheLastName"
                   IsQueriable="false"/>
  </variableIterator>
</executables>
```

18.4.3 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 named bind variables 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 18.5, "Conditionally Displaying the Results Table on a Search Page".

When the user enters Smith as the last name in the corresponding inputText component, and clicks the command button, the following happens:

- The StaffListByEmailNameRole_TheLastName variable is populated with the value Smith. If no values were entered for the other fields, then the corresponding variables use the default value set for the named bind variables on the view object. For more information, see Section 5.9, "Using Named Bind Variables".
- Because the attribute binding refers to the variable iterator, the attribute binding can get the value for TheLastName, and any other variable values:

```
<attributeValues id="TheLastName" IterBinding="variables">
  <AttrNames>
    <Item Value="StaffListByEmailNameRole_TheLastName"/>
  </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:

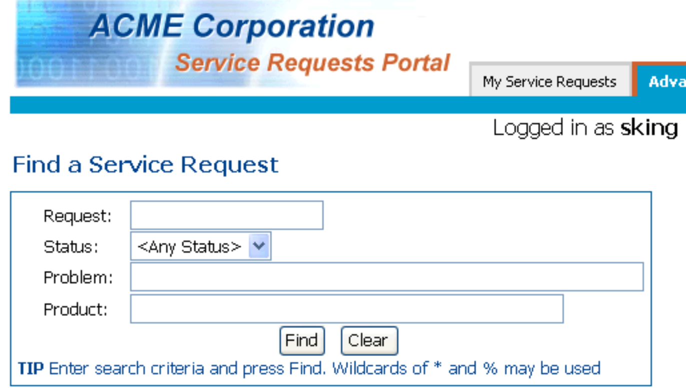
```
<NamedData NDName="TheLastName" NDType="java.lang.String"
    NDValue="${bindings.StaffListByEmailNameRole_TheLastName}"/>
```

- The `ExecuteWithParams` operation is executed with the parameters taking their values from the `NamedData` elements.
- The operation applies the named bind variable values and executes the query.
- The `StaffListByEmailNameRoleIterator` iterator iterates over the collection, allowing a table to display the results. For more information about tables at runtime, see Section 14.2.2, "What Happens When You Use the Data Control Palette to Create a Table".

18.5 Conditionally Displaying the Results Table on a Search Page

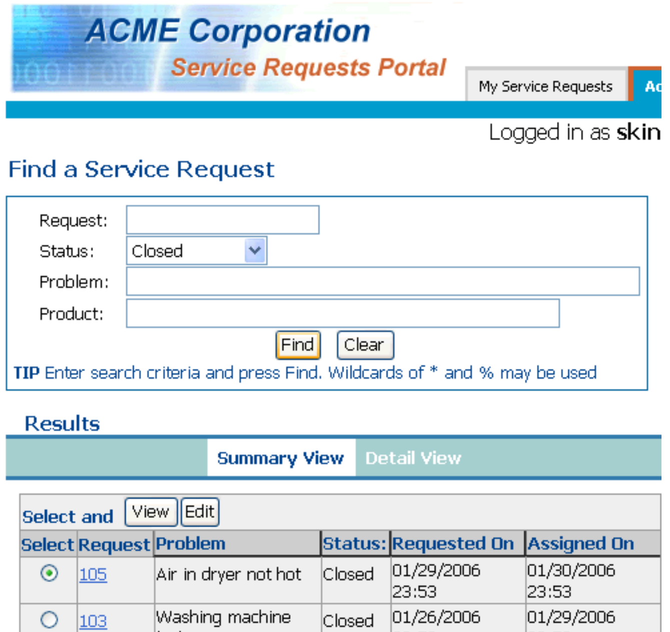
When a web search form and results table are on the same page, the first time a user accesses the page, the table displays all records in the current range from the iterator. You can make it so that the results table does not display until the user actually executes the search. Figure 18–7 shows the SRSearch page as it displays the first time a user accesses it.

Figure 18–7 Hidden Results Table for a Search Page



Once the user executes a search, the results table displays, as shown in Figure 18–8.

Figure 18–8 Results Table Displayed for a Search Page



18.5.1 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.
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 results table (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 17.2, "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 17.4, "Setting Parameter Values Using a Command Component".

   Example 18–6 shows the code for the **Search** button on the SRSearch page.

Example 18–6 Using a setActionListener Component to Set a Value

```
<af:commandButton actionListener="#{bindings.Execute.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 18–7 shows the EL expression used for the value for the `Rendered` attribute of the `panelGroup` component on the SRSearch page.

Example 18–7 JSF Code to Conditionally Display the Search Results Table

```
<af:panelGroup rendered="#{userState.searchFirstTime == false}">
```

   This EL expression causes the `panelGroup` component to render only if the `searchFirstTime` flag has a value of `False`.

18.5.2 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 SEARCH_FIRSTTIME_FLAG field to `true`.
- When the EL expression for the `panelGroup` component is evaluated, because he SEARCH_FIRSTTIME_FLAG field is `true`, the SRSearch page displays without rendering the `panelGroup` component. This is because the EL expression for the `Rendered` attribute evaluates to `True` only when SEARCH_FIRSTTIME_FLAG field is `false`.
- When the user enters search criteria and clicks the **Search** button, the associated `setActionListener` component sets the SEARCH_FIRSTTIME_FLAG field 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 SEARCH_FIRSTTIME_FLAG field is now set to `false`, when the page rerenders with the results, the `panelGroup` component displays the table with the result.

19 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 19.1, "Introduction to Complex UI Components"
- Section 19.2, "Using Dynamic Menus for Navigation"
- Section 19.3, "Using Popup Dialogs"
- Section 19.4, "Enabling Partial Page Rendering"
- Section 19.5, "Creating a Multipage Process"
- Section 19.6, "Providing File Upload Capability"
- Section 19.7, "Creating Selection Lists"
- Section 19.8, "Creating a Shuttle"

19.1 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.

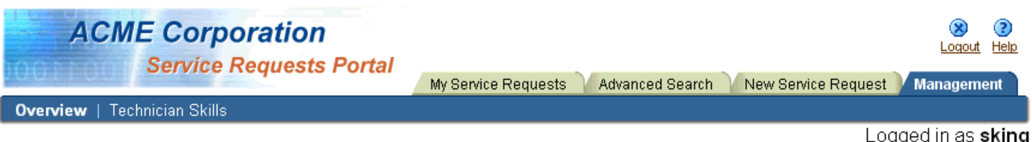
Read this chapter to understand:

- 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

19.2 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 19–1 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 19–1 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 via dynamic includes using `<f:subview>` and `<jsp:include>` tags. In some pages, *cacheable fragments* are used to contain the `menuTabs` 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 23, "Optimizing Application Performance with Caching".

19.2.1 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 19.2.1.1, "Creating a Menu Model")
2. Create a JSF page for each menu choice or item in the menu hierarchy. (See Section 19.2.1.2, "Creating the JSF Page for Each Menu Item")
3. Create one global navigation rule that has navigation cases for each menu item. (See Section 19.2.1.3, "Creating the JSF Navigation Rules")

19.2.1.1 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 19–1 shows the `MenuItem` class used in the SRDemo application.

Example 19–1 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 19–2 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 19–2 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 19–1, 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 19–3 shows the `MenuTreeModelAdapter` class used in the SRDemo application.

Example 19–3 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 19–2. Example 19–4 shows the managed bean for creating the menu tree model.

Example 19–4 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 19–5 shows the `MenuModelAdapter` class used in the SRDemo application.

Example 19–5 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 19–6 shows the managed bean code for creating the menu model.

Example 19–6 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>
```

19.2.1.1.1 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 19–1 lists the compatible bean scopes.

Table 19–1 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`

19.2.1.1.2 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 19–7 shows the `ResourceAdapter` class, and the `JSFUtils.getStringFromBundle()` method that retrieves a `String` from the bundle.

Example 19–7   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 19–8 shows the `resources` managed bean code that provides the access for other managed beans to the `String` resources.

Example 19–8 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.

19.2.1.2 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 19–9 shows the `menuTabs` component code that binds the component to a menu model.

Example 19–9 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 19–10. 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 19–10 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 19–1: 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 19–11 shows how this is done for `menuTabs` (a nonglobal component) and `menuButtons` (a global component).

Example 19–11 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 19–12 shows part of the menu code for a `panelPage` component.

Example 19–12 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.

19.2.1.2.1 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.

19.2.1.3 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 19–13.

Example 19–13 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>
```

19.2.2 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 19–14, 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 19–14 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.

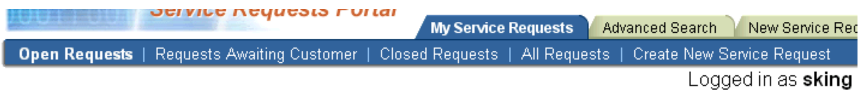
The menu model in conjunction with nodeStamp controls whether a menu item is rendered as selected. As described earlier, a menu model is created from a tree model, which contains viewId information for each node. ViewIdPropertyMenuModel, which is an instance of MenuModel, uses the viewId of a node to 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.

19.2.3 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 19–2. 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 19–2 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 19–15 shows part of the code for the menuBar component used in the SRList.jspx page.

Example 19–15 MenuBar Component with Children CommandMenuItem Components

```
<af:menuBar>
  <af:commandMenuItem text="#{res['srlist.menubar.openLink']}"
                      disabled="#{!bindings.ExecuteWithParams.enabled}"
                      selected="#{userState.listModeOpen}"
                      actionListener="#{bindings.ExecuteWithParams.execute}">
    <af:setActionListener from="#{'Open'}"
                          to="#{userState.listMode}"/>
  </af:commandMenuItem>
  <af:commandMenuItem text="#{res['srlist.menubar.pendingLink']}"
                      disabled="#{!bindings.ExecuteWithParams.enabled}"
                      selected="#{userState.listModePending}"
                      actionListener="#{bindings.ExecuteWithParams.execute}"
    <af:setActionListener from="#{'Pending'}"
                          to="#{userState.listMode}"/>
  </af:commandMenuItem>
  ...
  <af:commandMenuItem text="#{res['srlist.menubar.allRequests']}"
                      selected="#{userState.listModeAll}"
                      disabled="#{!bindings.ExecuteWithParams.enabled}"
                      actionListener="#{bindings.ExecuteWithParams.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.

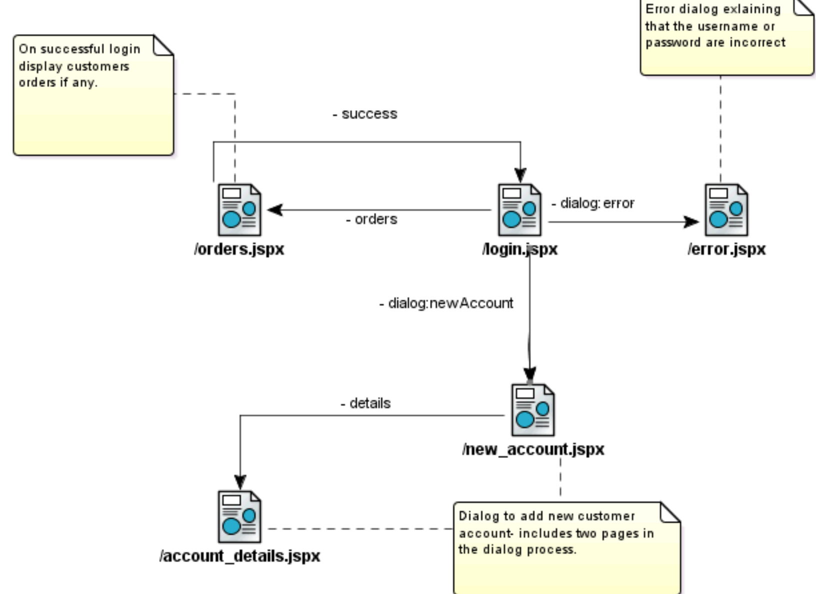
When the `commandMenuItem` component is activated, the `ExecuteWithParams` built-in operation is executed, passing the value of its parameter to the named bind variable of the `ServiceRequestsByStatus` view object instance in the data model. This updates the row set to contain only service requests whose status matches the parameter value passed in. The `commandMenuItem` components also use convenience functions in the `UserSystemState` bean class to evaluate whether the menu item should be marked as selected or not.

19.3 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.

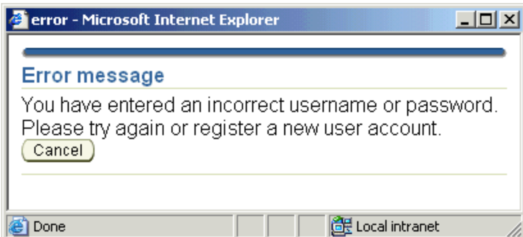
Consider a simple application that requires users to log in to see their orders. Figure 19–3 shows the page flow for the application, which consists of five pages—`login.jspx`, `orders.jspx`, `new_account.jspx`, `account_details.jspx`, and `error.jspx`.

Figure 19–3 Page Flow of a Dialog Sample Application



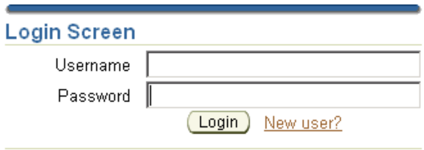
When an existing user logs in successfully, the application displays the Orders page, which shows the user's orders, if there are any. When a user does not log in successfully, the Error page displays in a popup dialog, as shown in Figure 19–4.

Figure 19–4 Error Page in a Popup Dialog



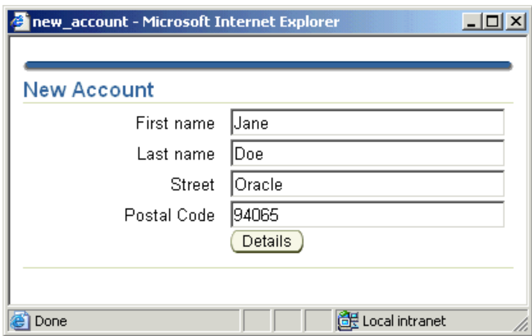
On the Error page there is a **Cancel** button. When the user clicks **Cancel**, the popup dialog closes and the application returns to the Login page, as shown in Figure 19–5.

Figure 19–5 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 19–6.

Figure 19–6 New Account Page in a Popup Dialog

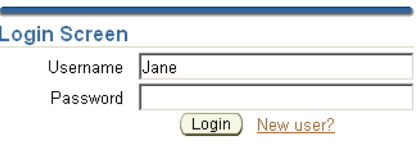


After entering information such as first name and last name, the user then clicks the **Details** button to display the Account Details page in the same popup dialog, as shown in Figure 19–7. In the Account Details page, the user enters other information and confirms a password for the new login account. There are two buttons on the Account Details page—**Cancel** and **Done**.

Figure 19–7 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 19–8. The new user can then proceed to enter the new password and log in successfully.

Figure 19–8 Login Page With the Username Field Populated



19.3.1 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.

19.3.1.1 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 19–3, 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 19–16 shows the navigation rule for the three navigation cases from the Login page (`login.jspx`).

Example 19–16 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>
```

19.3.1.1.1 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.

19.3.1.2 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 19–17 shows the code for the Login `commandButton` component.

Example 19–17 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 19–18.

Example 19–18 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 19–19 shows the code for the **New User** `commandLink` component that uses a static action outcome.

Example 19–19 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:`.

19.3.1.2.1 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 19.3.1.5, "Passing a Value into a Dialog".

19.3.1.3 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 19–20 shows how to accomplish this programmatically via a **Cancel** button on the Error page.

Example 19–20 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 19–21.

Example 19–21 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 19–22. The `af:returnActionListener` tag calls the `returnFromDialog` method on the `AdfFacesContext`—no backing bean code is needed.

Example 19–22 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 19–23 shows the code for the **Done** button.

Example 19–23 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 19–24 shows the code for the action listener method, where the return value is retrieved, and then returned via the `AdfFacesContext.returnFromDialog()` method.

Example 19–24 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.

19.3.1.3.1 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 19.3.1.4, "Handling the Return Value".

19.3.1.4 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 19–25 shows the code for the **New User** link component.

Example 19–25 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 19–26 shows the code for the return listener method that handles the return value.

Example 19–26 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`.

19.3.1.4.1 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.

19.3.1.5 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 19–27 shows the code for the `commandLink` component.

Example 19–27 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 19–28 shows the code for the launch listener method.

Example 19–28 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 19–29.

Example 19–29 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.

19.3.1.5.1 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`.

19.3.2 How the SRDemo Popup Dialogs Are Created

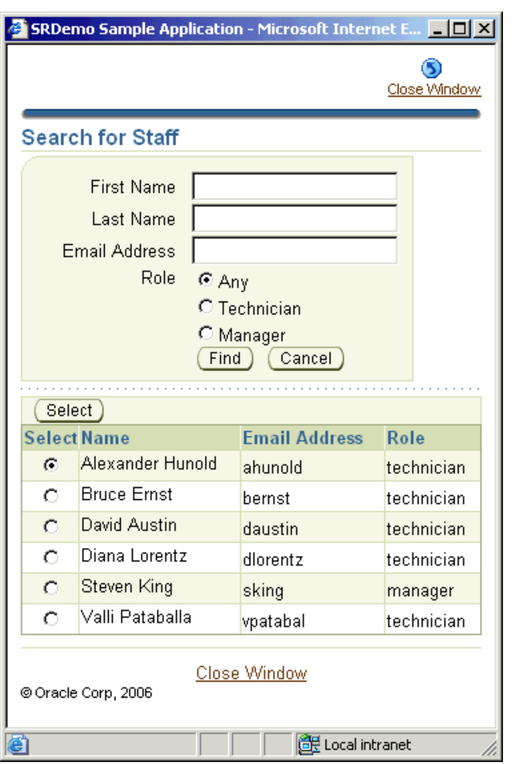
The SRDemo application uses a popup dialog to:

- 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 19–13), when the user clicks the **Frequently Asked Questions** link, the application displays a popup dialog showing the FAQ list.

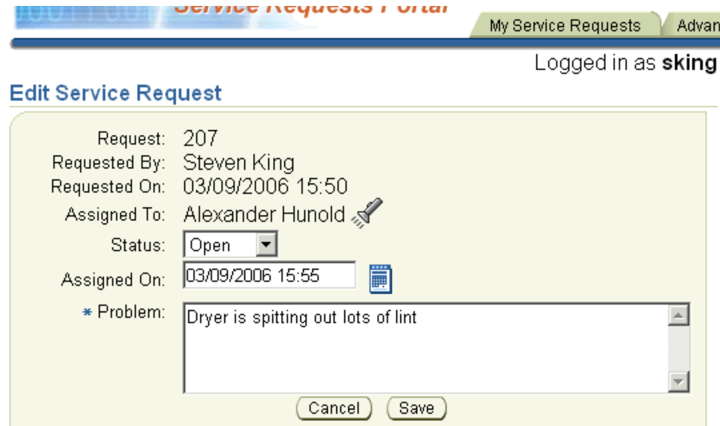
In the Edit Service Request page, when the user clicks the flashlight icon next to the **Assigned to** label (see Figure 19–12), the application displays the **Search for Staff** popup dialog. In the dialog (as shown in Figure 19–9), the user first makes a search based on user role. Then in the results section, the user clicks the radio button next to a name and clicks **Select**.

Figure 19–9 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 19–10. You don't need to write any code to facilitate the update as the fields are automatically updated when the foreign key `AssignedTo` is modified. For details, see Section 7.3, "Including Reference Entities in Join View Objects".

Figure 19–10 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 19–30. 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 19–30 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 19–31 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 19–31 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 19–32 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 19–32 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 name in the table. The `af:returnActionListener` tag on the **Select** button component calls the `returnFromDialog()` method, which closes the dialog and sends the return value from the dialog—no backing bean code is needed. Example 19–33 shows the code snippet for the **Select** button component.

Example 19–33 CommandButton Component for Selecting a Name and Closing the Popup Dialog

```
<af:tableSelectOne>
  <af:commandButton text="#{res['srstaffsearch.button.select']}">
    <af:setActionListener from="#{row.UserId}"
                          to="#{bindings.AssignedTo.inputValue}"/>
    <af:returnActionListener value="#{row.UserId}"/>
  </af:commandButton>
</af:tableSelectOne>
```

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 19–34 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 19–34 CommandLink Component for Closing the SRFaq Popup Dialog

```
<af:commandLink text="#{res['srdemo.close']}">
  <af:returnActionListener/>
</af:commandLink>
```

Note that the SRStaffSearch page definition file includes a binding to the `AssignedTo` attribute. In the SRStaffSearch dialog, when the user selects a radio button for a name and then clicks **Select**, the dialog is dismissed and the application returns to the SREdit page with the selected name populated in the **Assigned To** display-only fields. As shown in Example 19–33, the **Select** `commandButton` component has an `af:setActionListener` tag that is configured to set the value of the selected `UserId` to the value of the `AssignedTo` attribute. The **Select** `commandButton` also has the `af:returnActionListener` tag, which returns the value of the selected `UserId`. The `commandLink` component that launches the SRStaffSearch dialog has an attached return listener (see Example 19–31), but the return handler code (as shown in Example 19–35) does not use the return value. The return handler simply causes the **Assigned On** date field to refresh its value from the binding.

Example 19–35 Return Listener Method

```
public void handleStaffLOVReturn(ReturnEvent event) {
      getAssignedDate().resetValue();
}
```

19.3.3 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.

19.3.4 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.

19.4 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 19–11 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 19–11 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.

19.4.1 How to Enable PPR

The `SREdit.jspx` page of the SRDemo application uses partial page submits and partial triggers to support PPR.

Figure 19–12 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 (`selectInputDate` component) below **Status** are refreshed with the appropriate values; other parts of the edit page are not refreshed.

Figure 19–12 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 19–36 shows the code snippets for the command and read-only output components used in the `SREdit.jspx` page to illustrate PPR.

Example 19–36 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.Status.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.

19.4.2 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.

19.4.3 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.

19.5 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 19–13 shows the first page in the create service request process, where the user selects one appliance from a list box and enters a description of the problem in a text box. 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.

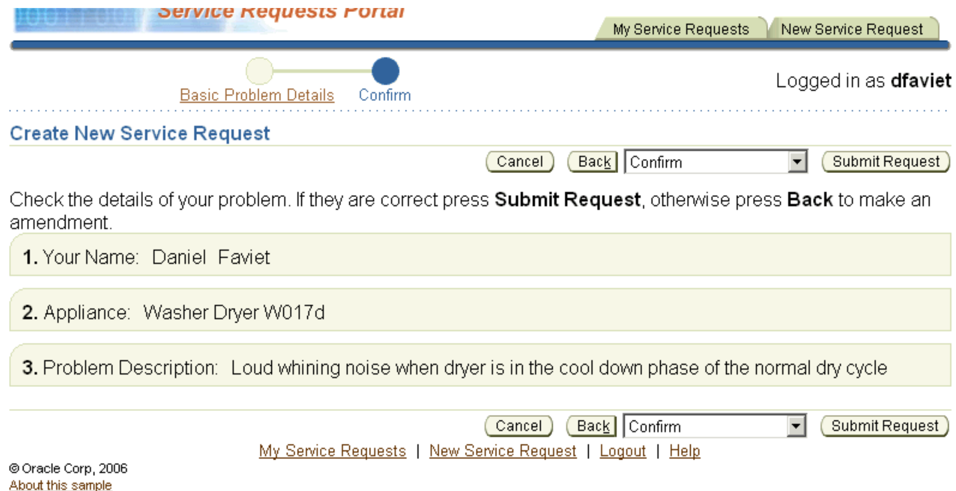
Figure 19–13 First Page of the Create New Service Request Process (SRCreate.jspx)

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 19–14.

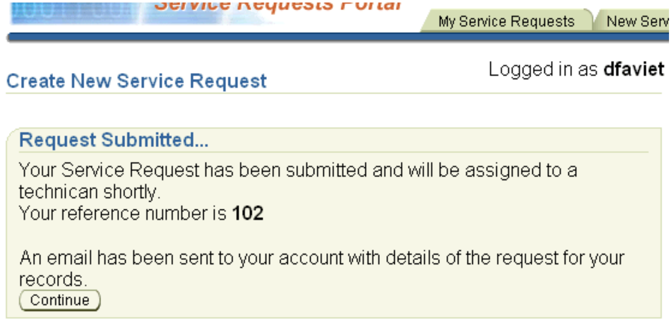
Figure 19–14 Second Page of the Create New Service Request Process (SRCreateConfirm.jspx)



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 19–15.

Figure 19–15 Request Submitted Page (SRCreateDone.jspx)



19.5.1 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 19.5.1.1, "Creating a Process Train Model")
2. Create the JSF page for each node in the train. (See Section 19.5.1.2, "Creating the JSF Page for Each Train Node")
3. Create a navigation rule that has navigation cases for each node. (See Section 19.5.1.3, "Creating the JSF Navigation Rules")

19.5.1.1 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 19.2.1, "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 19–37 shows part of the `MenuItem` class used in the SRDemo application.

Example 19–37 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 19–38 shows the managed bean code for the process train nodes in `faces-config.xml`.

Example 19–38 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 19–39 shows the managed bean code for creating the process train list.

Example 19–39 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 19–40 shows the `TrainModelAdapter` class used in the SRDemo application.

Example 19–40 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 19.5.1.1.1, "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 19–41 shows the managed bean code for creating the process train model.

Example 19–41 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>
```

19.5.1.1.1 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 19–41) 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 19–40).

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 19.5.1.2.1, "What You May Need to Know About the Immediate and ReadOnly Attributes".

19.5.1.2 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 19–42.

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 19–42 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 19–43 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 19–43 ProcessChoiceBar Component in the SRCreate.jspx File

```
<af:panelPage ..>
  <f:facet name="actions">
    <af:panelButtonBar>
      <af:commandButton text="#{res['srdemo.cancel']}"
                        action="GlobalHome"
                        actionListener=
                          "#{bindings.cancelNewServiceRequest.exec}"
                        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 19–13 and Figure 19–14, 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 19–14), 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**.

19.5.1.2.1 What You May Need to Know About the Immediate and ReadOnly Attributes

The two process scenarios provided by ADF Faces and described in Section 19.5.1.1.1, "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 19.5.1.1.1, 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.

19.5.1.3 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 19–44 shows the navigation rules and cases for the process.

Example 19–44 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>
```

19.5.2 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.

19.5.3 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 19.2, "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 19–38), add the new managed bean to the train list (Example 19–39), and add the navigation case for the new page (Example 19–44).

19.6 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.

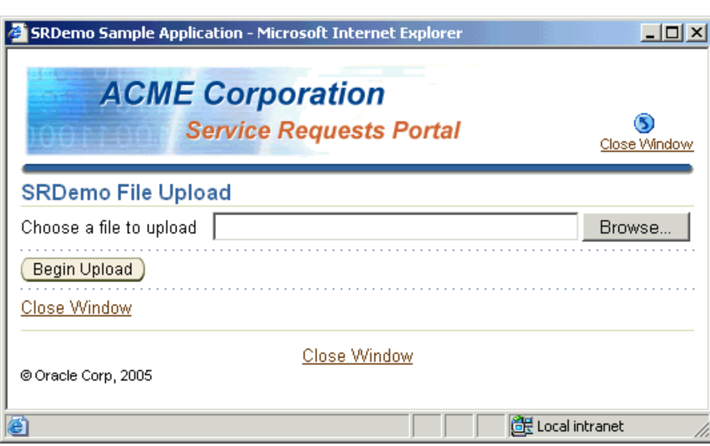
During file uploading, ADF Faces temporarily stores incoming files either in memory or on disk. You can set a default directory storage location, and default values for the amount of disk space and memory that can be used in any one file upload request.

Figure 19–16 shows the `SRMain.jspx` page of the SRDemo application, where users can upload files for a particular service request.

Figure 19–16 File Upload Button on the SRMain Page

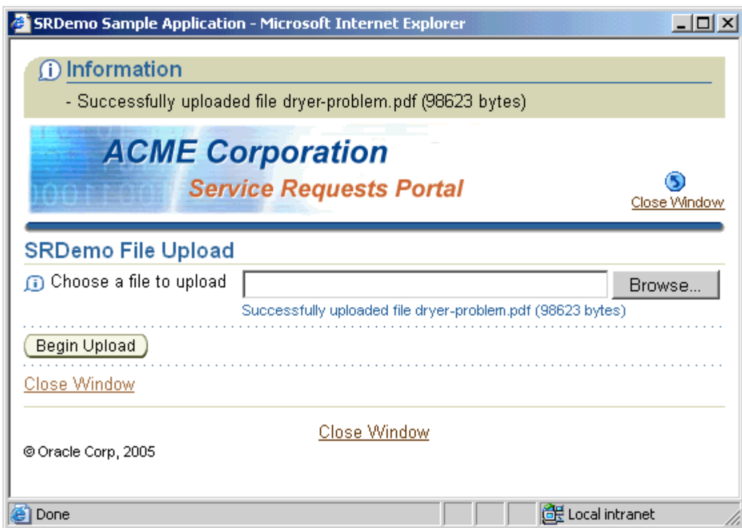
When the user clicks **Upload document**, the upload form displays in a popup dialog, as shown in Figure 19–17.

Figure 19–17 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 19–18. If uploading is unsuccessful for some reason, the application displays the error stack trace in the same popup dialog.

Figure 19–18 File Upload Success Information



19.6.1 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 19–45 shows the ADF Faces filter and mapping configuration setting in `web.xml`.

Example 19–45 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 19–46 shows the context parameter used in the SRDemo application for uploaded files.

Example 19–46 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 19–47 shows the managed bean code in `faces-config.xml` for the SRDemo file upload page.

Example 19–47 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 19–17.

   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 19–48 shows the code for the method binding expression in the `valueChangeListener` attribute of the `inputFile` component.

Example 19–48 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 19–49 shows the value change listener method that handles the value change event for file uploading in the SRDemo application.

Example 19–49 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.FILE_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
    DBSequence svrId =
     (DBSequence)ADFUtils.getBoundAttributeValue("SvrId");
    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 19–50 shows the `commandButton` code in the SRDemo file upload form, and also the action method code in the page's backing bean.

Example 19–50 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 19.3.1.3, "Creating the Dialog Page and Returning a Dialog Value". Example 19–51 shows the code for the `commandLink` component and the action method in the page's backing bean.

Example 19–51 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;
}
```

19.6.2 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 19.6.4, "Configuring File Uploading Initialization Parameters".

19.6.3 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 19.6.5, "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.

19.6.4 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 19–52 shows the context initialization parameters for file uploading that you use in `web.xml`.

Example 19–52 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.

19.6.5 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 19–53 shows the code for registering a custom `UploadedFileProcessor` implementation.

Example 19–53 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.

19.7 Creating Selection Lists

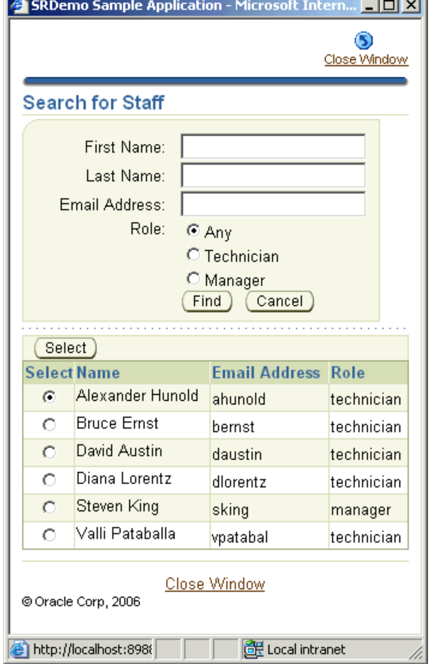
ADF Faces list components for selecting a single value from a list include `selectOneRadio`, `selectOneChoice` and `selectOneListbox`, which work in the same way as standard JSF list components. ADF Faces list components, however, provide extra functionality such as support for label and message display, automatic form submission, and partial page rendering.

In the SRDemo application, the SRStaffSearch page uses a `selectOneRadio` component to let users pick the staff role to perform a search on. The list of staff roles is bound to a static list of values that is provided by the developer. In the SRCreate page, a `selectOneListbox` component is used to let users select an appliance from a list of values that is populated dynamically at runtime. In the SRSkills page, a `selectOneChoice` component is used to let users traverse through the objects in a collection. In all cases, the `f:selectItems` tag is used to provide the list items for display and selection.

19.7.1 How to Create a List with a Fixed List of Values

The SRStaffSearch page uses a `selectOneRadio` component to let users pick the staff role to perform a search on. For example, the user can refine the search on staff members who have the role of Manager or Technician. Figure 19–19 shows the Search for Staff form in the SRDemo application.

Figure 19–19 SelectOneRadio Component for Selecting a Staff Role



The Search for Staff form uses a query from a view object that has named bind variables to find matching objects via the `ExecuteWithParams` operation. For information about how to create a search form using parameters, see Section 18.4.1, "How to Create a Parameterized Search Form".

In the SRStaffSearch page definition file, as shown in Example 19–54, the variable `StaffListByEmailNameRole_Role` is bound to the `Role` named bind variable in the `StaffListByEmailNameRole` view object. The `StaffListByEmailNameRole`

view object extends the StaffList view object to add the bind variables used by this search page.

The iterator StaffListByEmailNameRoleIterator iterates over the StaffListByEmailNameRole collection. The StaffListByEmailNameRoleIterator is also related to the ExecuteWithParams action, which encapsulates the details about how to invoke the action and what parameters the action is expecting. The NamedData elements show the parameters to be passed to the ExecuteWithParams action.

When the user selects a **Role** radio button, the selectOneRadio component in the SRStaffSearch page populates the appropriate page definition variable with the selected value.

For information about variable iterators and variables, see Section 12.5.2.2, "Binding Objects Defined in the executables Element".

Example 19–54 SRStaffSearch Page Definition File

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.35.83" id="app_staff_SRStaffSearchPageDef"
                Package="oracle.srdemo.view.pageDefs" ...>
  <parameters/>
  <executables>
    <iterator id="StaffListByEmailNameRoleIterator"
              Binds="StaffListByEmailNameRole"
              RangeSize="10" DataControl="SRService"/>
    <variableIterator id="variables">
      <variableUsage DataControl="SRService"
                     Binds="StaffListbyEmailNameRole.variablesMap.Role"
                     Name="StaffListByEmailNameRole_Role" IsQueriable="false"/>
      ...
      <variableUsage DataControl="SRService"
                     Binds="StaffListByEmailNameRole.variablesMap.TheFirstName"
                     Name="StaffListByEmailNameRole_TheFirstName"
                     IsQueriable="false"/>
      <variableUsage DataControl="SRService"
                     Binds="StaffListByEmailNameRole.variablesMap.TheLastName"
                     Name="StaffListByEmailNameRole_TheLastName"
                     IsQueriable="false"/>
    </variableIterator>
    ...
  </executables>
  <bindings>
    <action id="ExecuteWithParams" IterBinding="StaffListByEmailNameRoleIterator"
            InstanceName="SRService.StaffListByEmailNameRole"
            DataControl="SRService" RequiresUpdateModel="true"
            Action="95">
      <NamedData NDName="Role" NDType="java.lang.String"
                 NDValue="${bindings.StaffListByEmailNameRole_Role}"/>
      ...
      <NamedData NDName="TheFirstName" NDType="java.lang.String"
                 NDValue="${bindings.StaffListByEmailNameRole_TheFirstName}"/>
      <NamedData NDName="TheLastName" NDType="java.lang.String"
                 NDValue="${bindings.StaffListByEmailNameRole_TheLastName}"/>
    </action>
    ...
```

```
    <attributeValues id="TheFirstName" IterBinding="variables">
      <AttrNames>
        <Item Value="StaffListByEmailNameRole_TheFirstName"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="TheLastName" IterBinding="variables">
      <AttrNames>
        <Item Value="StaffListByEmailNameRole_TheLastName"/>
      </AttrNames>
    </attributeValues>
    ...
  </bindings>
</pageDefinition>
```

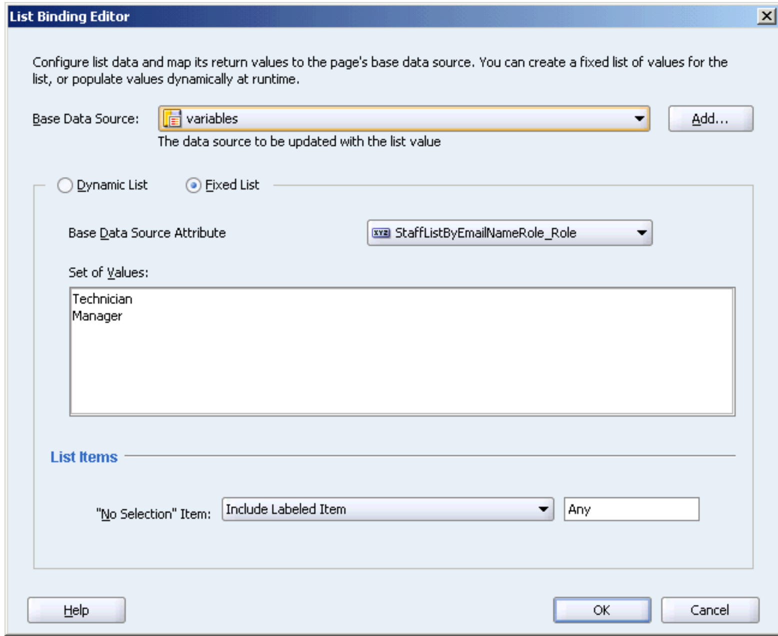
The following procedure assumes you've already created a parameter search form that uses the `ExecuteWithParams` operation.

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

1. Open the JSF page in the visual editor.
2. If necessary, delete the `inputText` component that was created for the **Role** field because you want to use a selection list component instead.
3. From the Data Control Palette, expand **StaffListByEmailNameRole > Operations > ExecuteWithParams > Parameters**.

   StaffListByEmailNameRole is the view object that contains the `Role` named bind variable.
4. Drag and drop the **Role** parameter to the page, and then choose **Create > Single Selections > ADF Select One Radio** from the context menu. The List Binding Editor displays, as illustrated in Figure 19–20.

Figure 19–20 List Binding Editor with the Fixed List Option Selected



5. In the List Binding Editor, make sure **variables** is selected in the **Base Data Source** dropdown list.

   Base Data Source is the target that will receive the value selected by the user at runtime.

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

   The **Fixed List** option lets users choose a value from a predefined list, which is useful when you want a data object attribute to be updated by a list of values that you code yourself, rather than getting the values from another data source.

7. From the **Base Data Source Attribute** dropdown list, select **StaffListByEmailNameRole_Role**.

   The **Base Data Source Attribute** is the variable name of the target.

8. Enter the following in the **Set of Values** box, pressing Enter to set a value before typing the next value:

   - `Technician`
   - `Manager`

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

9. In the **List Items** section, select **Include Labeled Item** from the **"No Selection" Item** dropdown list. Then enter `Any` in the box next to it.

   The `selectOneRadio` 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 8.

19.7.2 What Happens When You Create a List Bound to a Fixed List of Values

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 12.2.3, "What Happens When You Use the Data Control Palette".

Example 19–55 shows the code for the `selectOneRadio` component after you've completed the List Binding Editor.

Example 19–55 SelectOneRadio Component After You Complete Binding

```
<af:selectOneRadio value="#{bindings.Role.inputValue}"
                   label="#{bindings.Role.label}: ">
  <f:selectItems value="#{bindings.Role.items}"/>
</af:selectOneRadio>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `Role` list binding object in the binding container.

In the page definition file, JDeveloper adds the `Role` list binding object definition in the `bindings` element, as shown in Example 19–56.

Example 19–56 List Binding Object for the Fixed List in the Page Definition File

```
<bindings>
  ...
  <list id="Role" IterBinding="variables" ListOperMode="0" StaticList="true"
        NullValueFlag="1">
    <AttrNames>
      <Item Value="StaffListByEmailNameRole_Role"/>
    </AttrNames>
    <ValueList>
      <Item Value="Any"/>
      <Item Value="Technician"/>
      <Item Value="Manager"/>
    </ValueList>
  </list>
  ...
</bindings>
```

In the `list` element, the `id` attribute specifies the name of the list binding object. The `IterBinding` attribute references the variable iterator, whose current "row" is a row of attributes representing each variable in the binding container. The variable iterator exposes the variable values to the bindings in the same way as other collections of data. The `AttrNames` element specifies the attribute value returned by the iterator. The `ValueList` element specifies the fixed list of values to be displayed for selection.

At runtime, when a value is selected from the list, the base data source attribute `StaffListByEmailNameRole_Role`, which is the target of the `Role` list binding object, is updated to the selected value.

For more information about the page definition file and ADF data binding expressions, see Section 12.5, "Working with Page Definition Files" and Section 12.6, "Creating ADF Data Binding EL Expressions".

19.7.3 How to Create a List with a Dynamic List of Values

Instead of getting values from a static list, you can populate a selection list component with values dynamically at runtime. The steps for creating a list component bound to a dynamic list are almost the same as those for creating a list component bound to a fixed list, with the exception that you define two data sources—one for the list data source that provides the dynamic list of values, and the other for the base data source that is to be updated based on the user’s selection.

The SRCreate page, as shown in Figure 19–21, uses a `selectOneListbox` component to let users pick an appliance from a product list that is populated dynamically. The list data source is the `ProductList` collection because you want users to select an appliance (product) from the list. The base data source you want to update with the selected value is the `Globals` collection. The `Globals` view object is a transient view object, that is, one containing only transient attributes. Transient view objects are useful for holding temporary values in the data model that need to be presented on different pages of your application.

Figure 19–21 SelectOneListbox Component on the SRCreate Page

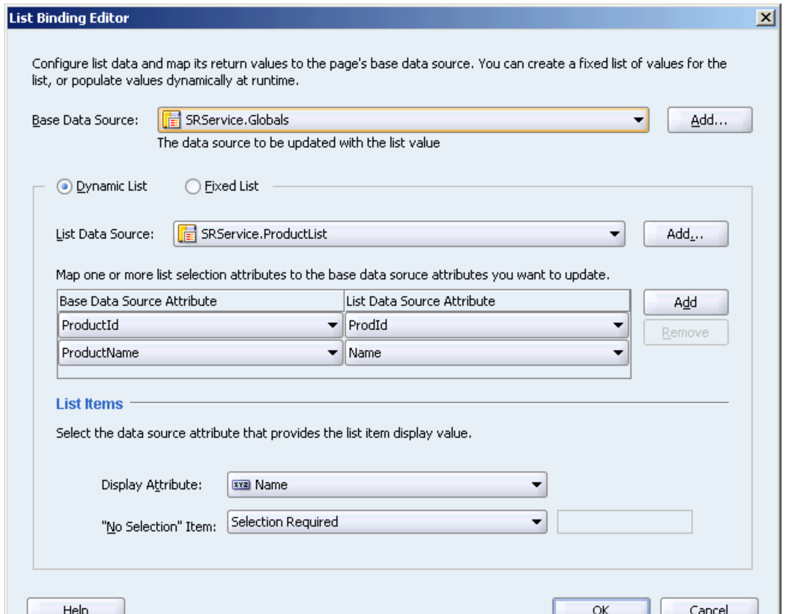
Service Requests Portal
My Service Requests | New Service Request
Basic Problem Details   Confirm
Logged in as **dfaviet**
Create New Service Request
Cancel   Basic Problem Details   Continue
Have you checked the Frequently Asked Questions?
Enter a *basic description of your problem*, be sure to include **any** information that you believe will be useful for the technician
1. Select your appliance: * Chest Freezer Z001w / Dryer D003 / Dryer D011 / Freezer Z002b / Freezer Z002s / Freezer Z002w
2. Describe the problem: *
Cancel   Basic Problem Details   Continue
My Service Requests | **New Service Request** | Logout | Help
© Oracle Corp, 2006
About this sample

To create a list box bound to a dynamic list of values:

1. Open the JSF page in the visual editor.
2. From the Data Control Palette, drag and drop any attribute of the base data source to the page. Choose **Create > Single Selections > ADF Select One Listbox** from the context menu. The List Binding Editor displays, as illustrated in Figure 19–22.

   For example, you might expand the **Globals** collection, then drag and drop the **ProductId** attribute to the page.

Figure 19–22 List Binding Editor with the Dynamic List Option Selected



3. In the List Binding Editor, the **Base Data Source** dropdown list should default to the collection of the attribute you dragged (for example, **SRService.Globals**).

   The **Base Data Source** is the source that is to be updated with the list value selected by a user at runtime.

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

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

5. Next to the **List Data Source** dropdown list, click **Add...** to select the list data source. In the **Add Data Source** dialog, select the collection that will provide the list of values dynamically. For example, in the **Add Data Source** dialog, expand **SRService**, and then select **ProductList**. Accept the default iterator name for the selected collection.

   The **List Data Source** is the source that provides the list of values users see in the list box at runtime.

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

6. In the mapping area, the default base data source attribute should be **ProductId** and the default list data source attribute should be **ProdId**, if you dragged the `ProductId` attribute of `Globals` in step 2, and selected the `ProductList` collection in step 5. The mapping specifies the list source attribute to the base source attribute you want to update. Click **Add** to add another mapping using the dropdown lists, if you wish.

   For example, you might select **ProductName** from the **Base Data Source Attribute** dropdown list, and then **Name** from the **List Data Source Attribute** dropdown list

7. In the **List Items** section, select the desired attribute of the list data source from the **Display Attribute** dropdown list. The list data source attribute populates the list values users see at runtime.

   For example, you might select **Name** from the **Display Attribute** dropdown list because you want users to select a product name from the list.

19.7.4 What Happens When You Create a List Bound to a Dynamic List of Values

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 12.2.3, "What Happens When You Use the Data Control Palette".

Example 19–57 shows the code for the `selectOneListbox` component after you've completed the List Binding Editor.

Example 19–57 SelectOneListbox Component After You Complete Binding

```
<af:selectOneListbox value="#{bindings.GlobalsProductId.inputValue}"
                    label="#{bindings.GlobalsProductId.label}">
  <f:selectItems value="#{bindings.GlobalsProductId.items}"/>
</af:selectOneListbox>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `GlobalsProductId` list binding object in the binding container. For further descriptions about ADF data binding expressions, see Section 12.6, "Creating ADF Data Binding EL Expressions".

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list binding object into the `bindings` element, as shown in Example 19–58.

Example 19–58 List Binding Object for the Dynamic List in the Page Definition File

```
<executables>
  ...
  <iterator id="ProductListIterator" Binds="ProductList" RangeSize="-1"
            DataControl="SRService"/>
  <iterator id="GlobalsIterator" RangeSize="10" Binds="Globals"
            DataControl="SRService"/>
</executables>
<bindings>
  ...
  <list id="GlobalsProductId" IterBinding="GlobalsIterator" StaticList="false"
        ListOperMode="0" ListIter="ProductListIterator">
    <AttrNames>
      <Item Value="ProductId"/>
      <Item Value="ProductName"/>
    </AttrNames>
    <ListAttrNames>
      <Item Value="ProdId"/>
      <Item Value="Name"/>
    </ListAttrNames>
    <ListDisplayAttrNames>
      <Item Value="Name"/>
    </ListDisplayAttrNames>
  </list>
  ...
</bindings>
```

By default, JDeveloper sets the `RangeSize` attribute on the `iterator` element for the `ProductList` iterator binding to a value of -1, thus allowing the iterator to furnish the full list of valid products for selection. In the `list` element, the `id` attribute specifies the name of the list binding object. The `IterBinding` attribute references the iterator that iterates over the `Globals` collection. The `ListIter` attribute references the iterator that iterates over the `ProductList` collection. The `AttrNames` element specifies the base data source attributes returned by the base iterator. The `ListAttrNames` element defines the list data source attributes that are mapped to the base data source attributes. The `ListDisplayAttrNames` element specifies the list data source attribute that populates the values users see in the list at runtime.

For complete information about page definition files, see Section 12.5, "Working with Page Definition Files".

19.7.5 How to Create a List with Navigation List Binding

The SRSkills page has a `selectOneChoice` component that uses *navigation list binding*. Navigation list binding is used to let users navigate through the objects in a collection. As the user changes the current object selection using the navigation list component, any other component that is also bound to the same collection through its attributes will display from the newly selected object. In addition, if the collection whose current row you change is the master view object instance in a data model master/detail relationship, the row set in the detail view object instance is automatically updated to show the appropriate data for the new current master row As illustrated later in Figure 19–26, when a manager selects a technician's name from the dropdown list, the shuttle component below the navigation list component is updated to show the selected technician's product skills.

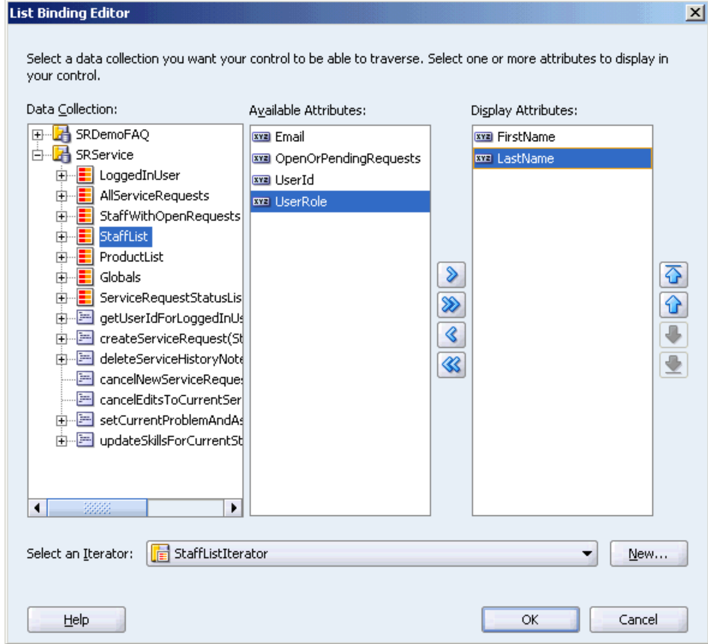
The dropdown list for selecting a technician name is derived from the `StaffList` collection, which has a detail collection, `StaffExpertiseAreas`, that is created via the view link `ExpertiseAreasForStaffMember`.

To create a list that uses navigation list binding:

1. From the Data Control Palette, expand **SRService**. Drag and drop the desired collection to the page, and then choose **Create > Navigation > ADF Navigation Lists** from the context menu. The List Binding Editor displays, as illustrated in Figure 19–23.

   For example, you might drop **StaffList** because you want users to navigate and select a technician name.

Figure 19–23 Navigation List Binding Editor



2. In the **Display Attributes** section, double-click the attributes of the `StaffList` collection that you don't want to be displayed in the list, moving them to the **Available Attributes** section.
3. In the **Select an Iterator** dropdown list, the default iterator name for the collection is shown. Accept the default or click **New** to create a new name for the iterator.

19.7.6 What Happens When You Create a List With Navigation List Binding

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 12.2.3, "What Happens When You Use the Data Control Palette".

By default JDeveloper creates the list using the `selectOneChoice` component when you use navigation list binding.

Example 19–59 shows the code for the `selectOneChoice` component after you've completed the List Binding Editor.

Example 19–59 SelectOneChoice Component After You Complete the Binding

```
<af:selectOneChoice id="navList1" autoSubmit="true"
                                  value="#{bindings.StaffList.inputValue}"
                                  label="#{bindings.StaffList.label}"
  <f:selectItems value="#{bindings.StaffList.items}"/>
</af:selectOneChoice>
```

The `f:selectItems` tag provides the list of items for traversing and selecting. In the page definition file, JDeveloper adds the list binding object into the `bindings` element, and the iterator binding object into the `executables` element, as shown in Example 19–60.

Example 19–60 List Binding Object for the Navigation List in the Page Definition File

```
<executables>
  <iterator id="StaffListIterator" Binds="StaffList" RangeSize="10"
            DataControl="SRService"/>
  ...
</executables>
<bindings>
  <list StaticList="false" ListOperMode="1" id="StaffList"
        IterBinding="StaffListIterator">
    <AttrNames>
      <Item Value="FirstName"/>
      <Item Value="LastName"/>
    </AttrNames>
  </list>
  ...
</bindings>
```

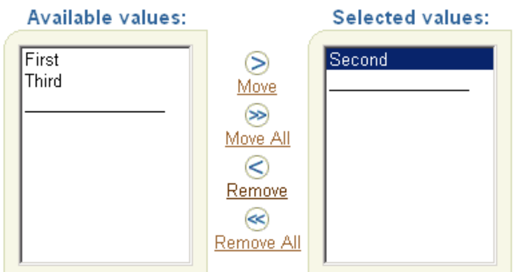
In the `list` element, the `id` attribute specifies the name of the list binding object. The `IterBinding` attribute references the iterator that iterates over the `StaffList` collection. The `AttrNames` element specifies the attributes returned by the iterator.

For more information about the page definition file and ADF data binding expressions, see Section 12.5, "Working with Page Definition Files" and Section 12.6, "Creating ADF Data Binding EL Expressions".

19.8 Creating a 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 19–24 shows an example of a rendered `selectManyShuttle` component. You can specify any text you want for the headers that display above the list boxes.

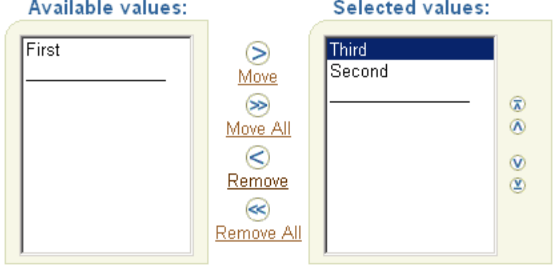
Figure 19–24 Shuttle (SelectManyShuttle) Component



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.

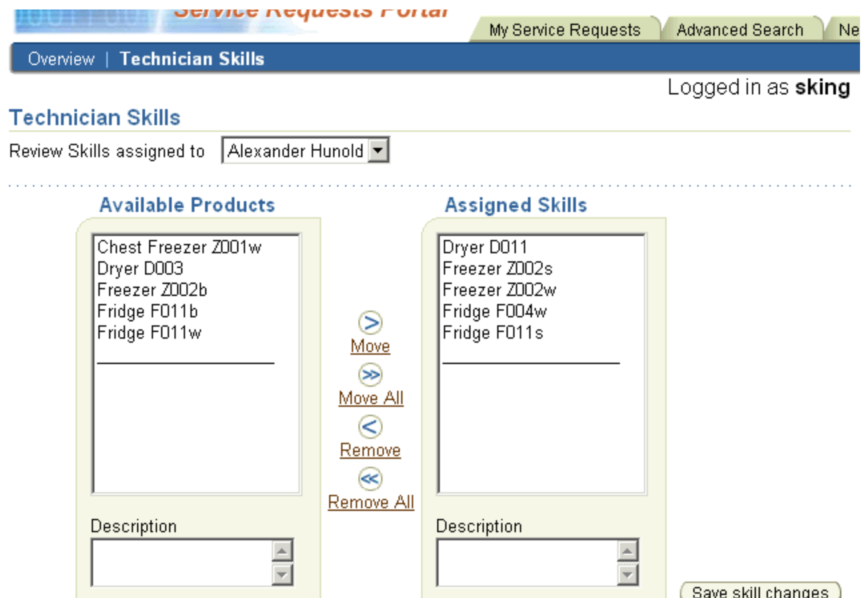
The only difference between `selectManyShuttle` and `selectOrderShuttle` is that in the `selectOrderShuttle` component, the user can reorder the items in the trailing list box by using the up and down arrow buttons on the side, as shown in Figure 19–25.

Figure 19–25 Shuttle Component (SelectOrderShuttle) with Reorder Buttons



In the SRDemo application, the SRSkills page uses a `selectManyShuttle` component to let managers assign product skills to a technician. Figure 19–26 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 19–26 SelectManyShuttle Component on the SRSkills Page



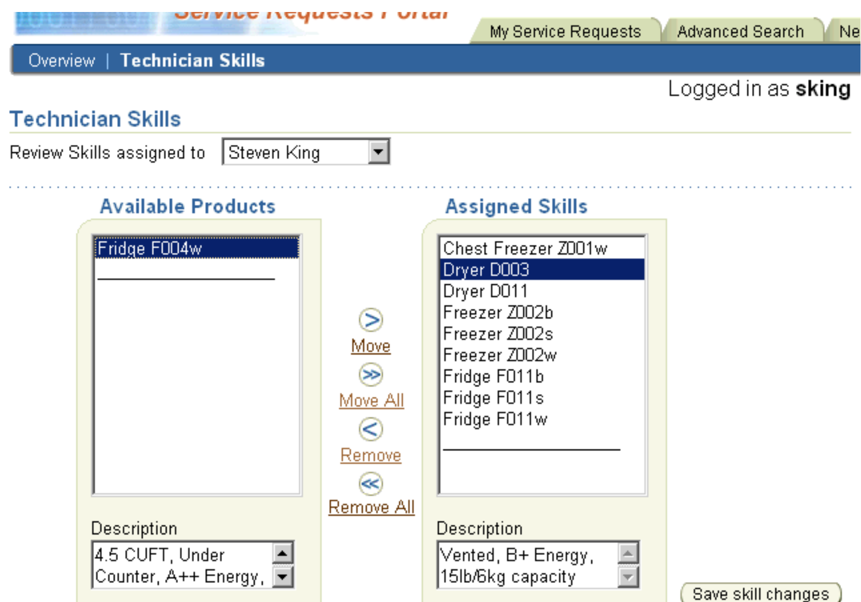
Only users with the manager role can access the SRSkills page. To review and change product skill assignments, a manager first selects a technician's name from the dropdown list above the shuttle component. The application then displays the technician's existing skill assignments in the trailing list (**Assigned Skills**).

To add new skill assignments for the selected technician, 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.

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, as shown in Figure 19–27.

Figure 19–27 Shuttle Component with Descriptions Shown



19.8.1 How to Create a Shuttle

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 generic class that can be used by any page that requires a shuttle. In the class, declare and include getter and setter methods for the properties that describe the view object instance names that should be used for the list of all available choices (leading list or available products) and the list of selected choices (trailing list or assigned skills). Example 19–61 shows the `ShuttlePageBackingBeanBase` class that is created to manage the population and selection state of the shuttle component on the SRSkills page.

Example 19–61 ShuttlePageBackingBeanBase Class

```
package oracle.srdemo.view.util;
import java.util.List;
import javax.faces.event.ValueChangeEvent;
public class ShuttlePageBackingBeanBase {
  String allItemsIteratorName;
  String allItemsValueAttrName;
  String allItemsDisplayAttrName;
  String allItemsDescriptionAttrName;
  String selectedValuesIteratorName;
  String selectedValuesValueAttrName;
  List selectedValues;
  List allItems;
  private boolean refreshSelectedList = false;

  public void setAllItemsIteratorName(String allItemsIteratorName) {
    this.allItemsIteratorName = allItemsIteratorName;
  }
  public String getAllItemsIteratorName() {
    return allItemsIteratorName;
  }
```

```
// other getter and setter methods are omitted
  public void setSelectedValues(List selectedValues) {
    this.selectedValues = selectedValues;
  }
  public void refreshSelectedList(ValueChangeEvent event) {
      refreshSelectedList = true;
    }
  public List getSelectedValues() {
    if (selectedValues == null || refreshSelectedList) {
      selectedValues = ADFUtils.
       attributeListForIterator(selectedValuesIteratorName,
                                selectedValuesValueAttrName);
    }
    return selectedValues;
  }
  public void setAllItems(List allItems) {
    this.allItems = allItems;
  }
  public List getAllItems() {
    if (allItems == null) {
       allItems = ADFUtils.selectItemsForIterator(allItemsIteratorName,
                                                  allItemsValueAttrName,
                                                  allItemsDisplayAttrName,
                                                  allItemsDescriptionAttrName);
    }
    return allItems;
  }
}
```

The `getAllItems()` method populates the shuttle's leading list. The `getSelectedValues()` method also returns a `List`, but the list defines the items in the shuttle's trailing list. Note that the `ShuttlePageBackingBeanBase` class calls several utility methods in the `ADFUtils` class.

The SRSkills page backing bean (`backing_SRSkills`), which extends the `ShuttlePageBackingBeanBase` class, is injected with values for several properties of the base bean. Example 19–62 shows the managed bean and managed properties configured in `faces-config.xml` for working with the shuttle component.

Example 19–62 Managed Bean for the Shuttle Component in the faces-config.xml File

```
<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-property>
    <property-name>allItemsIteratorName</property-name>
    <value>ProductListIterator</value>
  </managed-property>
  <managed-property>
    <property-name>allItemsValueAttrName</property-name>
    <value>ProdId</value>
  </managed-property>
  <managed-property>
    <property-name>allItemsDisplayAttrName</property-name>
    <value>Name</value>
  </managed-property>
```

```
  <managed-property>
    <property-name>allItemsDescriptionAttrName</property-name>
    <value>Description</value>
  </managed-property>
  <managed-property>
    <property-name>selectedValuesIteratorName</property-name>
    <value>StaffExpertiseAreasIterator</value>
  </managed-property>
  <managed-property>
    <property-name>selectedValuesValueAttrName</property-name>
    <value>ProdId</value>
  </managed-property>
</managed-bean>
```

The SRSkills page uses the following iterator objects:

- `StaffListIterator`: Iterates over the `StaffList` collection, which provides the technician names in the dropdown list above the shuttle.
- `StaffExpertiseAreasIterator`: Iterates over the `StaffExpertiseAreas` collection, which provides the list of skills assigned to a selected technician name.
- `ProductListIterator`: Iterates over the `ProductList` collection, which provides the list of product names.

All the bindings of the SRSkills page are defined in the file `app_management_SRSkillsPageDef.xml`. Example 19–63 shows part of the page definition file for the SRSkills page.

Example 19–63 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="app_management_SRSkillsPageDef"
                Package="oracle.srdemo.view.pageDefs">
  <parameters/>
  <executables>
    <iterator id="StaffListIterator" Binds="StaffList" RangeSize="10"
              DataControl="SRService"/>
    <iterator id="StaffExpertiseAreasIterator" Binds="StaffExpertiseAreas"
              RangeSize="10" DataControl="SRService"/>
    <iterator id="ProductListIterator" Binds="ProductList" RangeSize="10"
              DataControl="SRService"/>
  </executables>
  ...
</pageDefinition>
```

The following procedure assumes you've already created the `selectOneChoice` component for selecting a technician from a dropdown list. For instructions on how to create the dropdown list, see Section 19.7.5, "How to Create a List with Navigation List Binding".

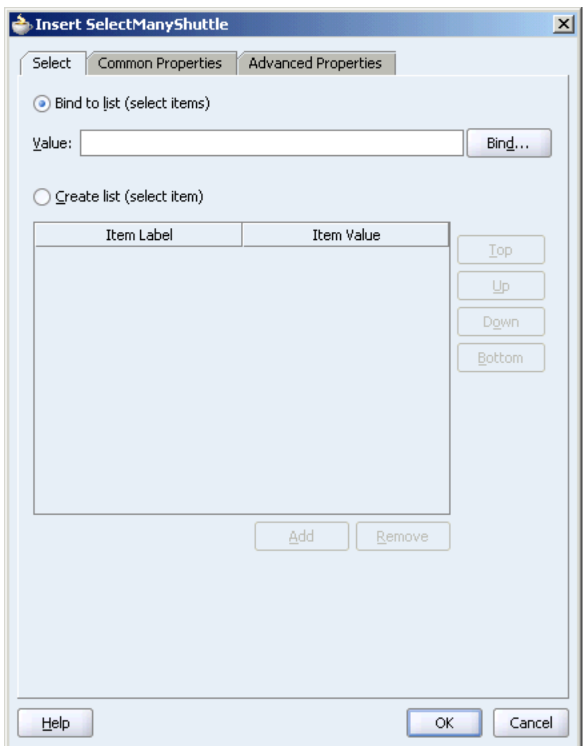
The procedure also assumes you've created the relevant iterator bindings in the page definition file (Example 19–63), a class similar to the `ShuttlePageBackingBeanBase` class (Example 19–61), and configured the required managed bean and managed properties in `faces-config.xml` (Example 19–62).

To create a shuttle component that is associated with a navigation list component:

1. In the JSF page that contains the navigation list component, select the `selectOneChoice` component. In the Property Inspector, set the **Id** attribute to a value (for example, `technicianList`), and then set the **ValueChangeListener** attribute to the `refreshSelectedList()` method in the `backing_SRSkills` managed bean (for example, `#{backing_SRSkills.refreshSelectedList}`).

2. From the **ADF Faces Core** page of the Component Palette, drag and drop **SelectManyShuttle** to the page. JDeveloper displays the Insert SelectManyShuttle dialog, as illustrated in Figure 19–28.

Figure 19–28 Insert SelectManyShuttle Dialog



3. Select **Bind to list (select items)** and click **Bind...** to open the Expression Builder.

4. In the Expression Builder, expand **JSF Managed Beans > backing_SRSkills**. Double-click **allItems** to build the expression `#{backing_SRSkills.allItems}`. Click **OK**.

   This binds the `f:selectItems` tag to the `getAllItems()` method that populates the shuttle's leading list.

5. In the Insert SelectManyShuttle dialog, click **Common Properties**. Click **Bind...** next to the **Value** field to open the Expression Builder again.

6. In the Expression Builder, expand **JSF Managed Beans > backing_SRSkills**. Double-click **selectedValues** to build the expression `#{backing_SRSkills.selectedValues}`. Click **OK**.

   This binds the `value` attribute of the `selectManyShuttle` component to the `getSelectedValues()` method that populates the shuttle's trailing list.

7. Close the Insert SelectManyShuttle dialog.
8. In the Property Inspector, set the `partialTriggers` attribute on the `selectManyShuttle` component to the Id of the `selectOneChoice` component (for example, **technicianList**) that provides the navigation dropdown list of technician names.

Example 19–64 shows the code for the `selectManyShuttle` component after you complete the Insert SelectManyShutle dialog.

Example 19–64 SelectManyShuttle Component in the SRSkills.jspx File

```
<af:selectManyShuttle value="#{backing_SRSkills.selectedValues}"
                      partialTriggers="technicianList"
                      ...
  <f:selectItems value="#{backing_SRSkills.allItems}"/>
</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

19.8.2 What Happens at Runtime

When the SRSkills page is first accessed, the iterator `StaffListIterator` executes and iterates over the `StaffList` collection. By default, the first item displayed in the `selectOneChoice` component is selected because the component has a navigation list binding, whose value is always set to the current row in the iterator. The initial selection automatically drives the master-detail coordination between the `StaffList` and `StaffExpertiseAreas` collections, thus the `selectManyShuttle` component is updated with the selected technician's product skills in the trailing list.

When the manager selects another technician name from the navigation dropdown list, the `selectOneChoice` component makes the new selection the current row in the `StaffList` iterator. Because the `selectOneChoice` component is listed as a `partialTriggers` component on the `selectManyShuttle` component, the shuttle rerenders with the newly selected technician's product skills in the trailing list.

The `partialTriggers` attribute setting on the `selectManyShuttle` component causes the shuttle to redisplay with the updated values in the leading and trailing lists.

When the **Save skill changes** command button (see Figure 19–26) is clicked, the current technician's associated `List` of product IDs (that is, assigned skills) are retrieved and sent to the `updateSkillsForCurrentStaff()` method.

Example 19–65 shows the code for the `commandButton` component on the `SRSkills.jspx` page.

Example 19–65 CommandButton Component in the SRSkills.jspx File

```
<af:commandButton action="#{backing_SRSkills.onUpdateSkills}"
                  actionListener="#{bindings.
                                  updateSkillsForCurrentStaff.execute}"../>
```

20

Using Validation and Conversion

This chapter describes how to add ADF Model validation and JSF 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 20.1, "Introduction to Validation and Conversion"
- Section 20.3, "Adding Validation"
- Section 20.4, "Creating Custom JSF Validation"
- Section 20.5, "Adding Conversion"
- Section 20.6, "Creating Custom JSF Converters"
- Section 20.7, "Displaying Error Messages"
- Section 20.8, "Handling and Displaying Exceptions in an ADF Application"

20.1 Introduction to Validation and Conversion

In an ADF Business Components application, virtually all validation code is defined in the shared, reusable business domain layer of entity objects. This ensures that your business information is validated consistently in every page where end users are allowed to make changes, and it simplifies maintenance by centralizing validation. For information about configuring the declarative runtime behavior for entity objects, see Section 6.6, "Configuring Declarative Runtime Behavior". For information about specifying and managing declarative validation rules in the entity or attribute level, see Section 6.7, "Using Declarative Validation Rules". For information about extending entity objects with custom code, see Section 9.3, "Using Method Validators".

In the model layer, ADF Model validation rules can be set for the attributes of a collection. In an ADF Business Components application, unless you use data controls other than the application module data controls, you don't need to add ADF Model validation rules.

In the view layer, ADF Faces input components have built-in validation capabilities. You set validation on a UI component either by setting the `required` attribute or by using one of the built-in 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.

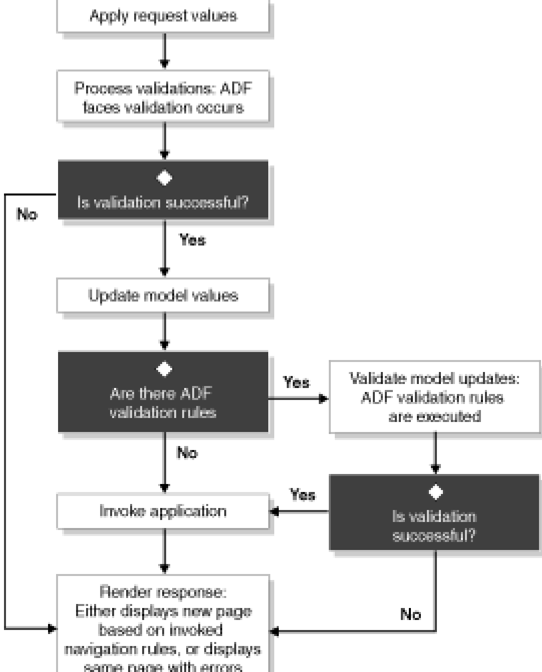
Read this chapter to understand:

- The ADF Faces validators and ADF Model validation, and how to use them in an 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

20.2 Validation, Conversion, and the Application Lifecycle

Figure 20–1 shows how validation and conversion work in the integrated JSF and ADF lifecycle.

Figure 20–1 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 the validation fails, the lifecycle proceeds to the Render Response phase. See Section 13.2.3, "What Happens at Runtime: The JSF and ADF Lifecycles" for more information. In your ADF Business Components application, note that unless you use data controls other than your application module data controls, you don't need to use additional ADF Model validation rules because the validators set in the business domain layer of entity objects are sufficient.

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 20.8, "Handling and Displaying Exceptions in an ADF Application".

20.3 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 20.3.1.1.1, "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 20.3.1.1.2, "Using JSF and ADF Faces Validators". You can also create your own validators. For information about custom validators, see Section 20.4.3, "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. Note that in an ADF Business Components application, unless you use data controls other than the application module data controls, you don't need to use additional ADF Model validation rules. If you want to use ADF Model validation, you declaratively set the validation rules on bindings to attributes of a collection. For more information, see Section 20.3.1.2, "Adding ADF Model Validation".

- Business layer: You set declarative validation rules for the entity object in the business layer, and any of the entity object attributes. By placing validation in the business domain layer of entity objects, the validation can be reused when that attribute's value is accessed by any page. Consider placing all, if not most, of your validation rules in the centralized, more reusable, and easier to maintain entity objects of your business domain layer.

For the purposes of this chapter, only the ADF Faces validators and ADF Model validation will be discussed. For information about using ADF Business Components declarative validation rules, see Section 6.7, "Using Declarative Validation Rules". For information about using or extending the basic set of declarative rules with custom rules and code of your own, see Section 9.3, "Using Method Validators".

20.3.1 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 20.7, "Displaying Error Messages".

20.3.1.1 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 20.3.1.1.1, "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 20.3.1.1.2, "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 20.4, "Creating Custom JSF Validation".

20.3.1.1.1 Using Validation Attributes

Many ADF Faces UI components have attributes that provide simple validation. Table 20–1 shows these attributes, along with a description of the validation logic they provide and the UI components that contain them.

Table 20–1 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 14.6, "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 20–3, " 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 search form was created using a parameter form, then the input components are usually bound to variables, since the attribute values are only temporarily stored in the binding container before passing them onto methods or parameterized actions. If a form was created using a collection, then the input component is probably bound to an attribute on an entity object.

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 20–1 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 20–1 shows a `RequiredMessageDetail` attribute that uses parameters.

Example 20–1 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**.

20.3.1.1.2 Using JSF and ADF Faces Validators

JSF and ADF Faces validators provide more complex validation routines. Table 20–2 describes the JSF reference implementation validators that have supplied tags.

Table 20–2 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 20–3 describes the built-in validators and tags supplied by ADF Faces.

Table 20–3 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. In an Oracle database table column of type `VARCHAR2(n)`, when using multi-byte character sets, the maximum allowed byte length might be different from `n`, depending on whether or not the size of the `VARCHAR2` in the database implies character or byte size.
`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 20.4, "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 20.3.2, "What Happens When You Create Input Fields Using the Data Control Palette". For information about ADF Model validation, see Section 20.3.1.2, "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.

20.3.1.2 Adding ADF Model Validation

Note that you don't need to add additional ADF Model validation if you have already set validation rules in the business domain layer of your entity objects. In an ADF Business Components application, unless you use data controls other than your application module data controls, you won't need to use ADF Model validation.

Table 20–4 describes the ADF Model validation rules that you can configure for an attribute.

Table 20–4 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**.

> **Note:** For information about ADF Business Components declarative validation rules, see Section 6.7, "Using Declarative Validation Rules".

20.3.2 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 20.7, "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 a collection similar to the `ProductList` collection from the Data Control Palette as a creation form. Example 20–2 shows the JSF code created by JDeveloper if such a collection were dropped.

Example 20–2 JSF Code for a Create Product Page

```
<afh:body>
  <af:messages/>
  <h:form>
    <af:panelForm>
      <af:inputText value="#{bindings.ProdId.inputValue}"
                    label="#{bindings.ProdId.label}"
                    required="#{bindings.ProdId.mandatory}"
                    columns="#{bindings.ProdId.displayWidth}">
        <af:validator binding="#{bindings.ProdId.validator}"/>
        <f:convertNumber groupingUsed="false"
                         pattern="#{bindings.ProdId.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>
        <f:facet name="footer">
          <af:commandButton text="Submit"/>
        </f:facet>
      </af:panelForm>
    </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`.

20.3.3 What Happens at Runtime

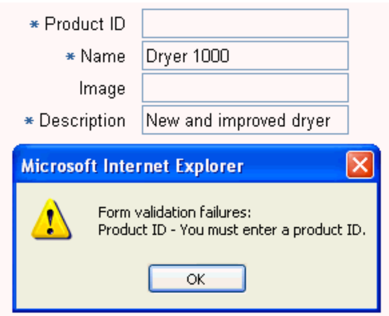
When the user submits the page, the ADF Faces `validate()` method first checks for a submitted value if the `required` attribute of a component is `true`. If the value is `null` or a zero-length string, the component is invalidated. At this point, what happens depends on whether or not client-side validation 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 20–2, a value for the `image` attribute is not required. However, all other values are required, as set by the `mandatory` property. This is denoted in the web page by asterisks next to the input text fields, as shown in Figure 20–2. Figure 20–2 also shows the alert dialog that is displayed if no data is entered for the product ID, and if client-side validation is enabled. If no data is entered for all three required fields, then the alert would show three error messages.

Figure 20–2 Client-Side Error for a Required Value

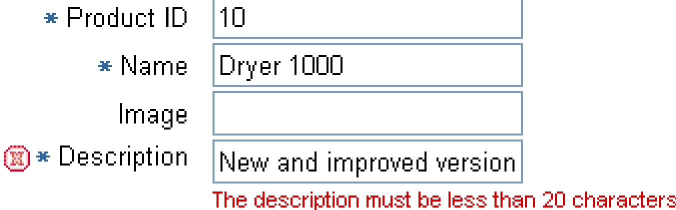


If the submitted value is a non-null value or a string value of at least one character, the validation process continues and all validators on the component are called one at a time. Because the `af:validator` tag on the component is bound to the `validator` property on the binding, any validation routines 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 20.8, "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 20–3 shows a server-side validation error.

Figure 20–3 Server-Side Validation Error



20.3.4 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.

20.4 Creating Custom JSF Validation

This section describes how to add backing bean validation methods, and provides information on how to create custom JSF validators. For information about using method validators to invoke custom validation code in ADF Business Components, or to extend the basic set of declarative rules with custom code, see Chapter 9, "Implementing Programmatic Business Rules in Entity Objects".

Sometimes you may need custom validation logic for a component on a single page. For example, if you have separate input fields for entering a date (month, day, year fields) and each has its own validator, users will not get an error if they enter February 30, 2005. By creating a validation method on the page's backing bean, the validation can access the separate components on the page, and thus validate the entire date.

If you need custom validation logic that will be reused by more than one page within the application, or if you want the validation to be able to run on the client side, you should implement a JSF validator class. You can then create an ADF Faces version, which will allow the validator to run on the client.

If you want to implement a custom, reusable validation rule to use as part of your domain business layer, see Section 26.9, "Implementing Custom Validation Rules".

20.4.1 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/`.

20.4.2 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 20–3 shows the code JDeveloper generates.

Example 20–3 Managed Bean Code for a Validation Method

```
public void inputText_validator(FacesContext facesContext,
                               UIComponent uiComponent, Object object) {
        // Add event code here...
}
```

JDeveloper also binds the `validator` attribute of the component to the backing bean's validation method using an EL expression. Example 20–4 shows the code JDeveloper adds to the component.

Example 20–4 JSF Code for a Custom Validation Method

```
<af:inputText value="#{bindings.SomeObject.inputValue}"
              label="#{bindings.SomeObject.label}"
              ...
              validator="#{backing_MyPage.inputText_validator}">
```

When the form containing the input component is submitted, the method to which the `validator` attribute is bound is executed.

20.4.3 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.

   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 20–5 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 20–5 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 20–6 shows the code on a JSF page for a validator using the `af:validator` tag.

Example 20–6 A Custom Validator Nested Within a Component on a JSF Page

```
<af:inputText id="empnumber" required="true">
  <af:validator validatorID="emValidator"/>
</af:inputText>
```

20.4.4 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`.

20.5 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 20–5 shows the converters provided by ADF Faces.

Table 20–5 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 20–5, the JavaScript-enabled converters are automatically used whenever needed. They do not have associated tags that can be nested in the component.

20.5.1 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.

20.5.2 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 20.7, "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 according to the format masks defined in the business layer objects, using an EL expression such as #{bindings.someObject.format}. For example, with the convertNumber converter, it might determine whether decimals are used.

For example, if you drop the ProdId attribute from the ProductList collection as an inputText component, JDeveloper automatically adds the convertNumber converter as a child of the input component, as shown in Example 20–7.

Example 20–7 Converter Tag in a JSF Page

```
<af:inputText value="#{bindings.ProductListProdId.inputValue}"
              required="#{bindings.ProductListProdId.mandatory}"
              columns="#{bindings.ProductListProdId.displayWidth}">
  ...
  <f:convertNumber groupingUsed="false"
                   pattern="#{bindings.ProductListProdId.format}"/>
</af:inputText>
```

The binding evaluates to the format property as it is set on the data control itself.

> **Tip:** If you drop a primary key attribute that is of type DBSequence from the Data Control Palette, JDeveloper does not add the f:convertNumber tag to the input component. If you create an edit form that uses a primary key of type Number, and later changed the type to DBSequence, you must either remove the f:convertNumber tag from the input component that was created earlier, or recreate the form. For information about DBSequence, see Section 6.6.3.8, "Trigger-Assigned Primary Key Values from a Database Sequence".

20.5.3 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.

20.6 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.

20.6.1 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 20–8 shows a custom converter referenced by the `converter` attribute of an `inputText` component.

Example 20–8 A Custom Converter on a JSF Page

```
<af:inputText value="#{bindings.ProductListName.inputValue}"
              label="#{bindings.ProductListName.label}"
              required="#{bindings.ProductListName.mandatory}"
              columns="#{bindings.ProductListName.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.

20.6.2 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.

20.7 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 20–4 shows the message displayed when data is not entered for an input component that has a `required` attribute set to `true`.

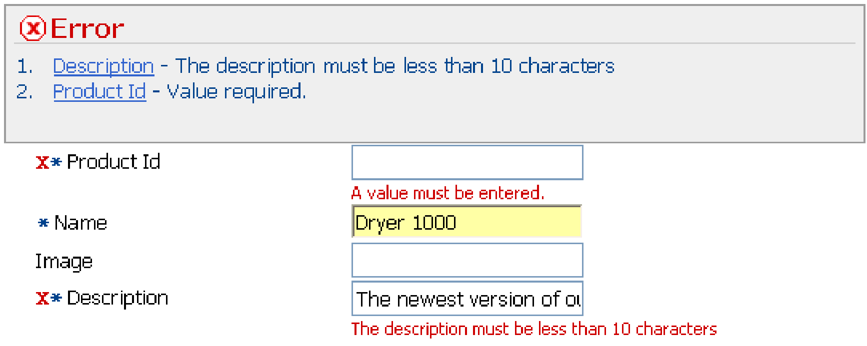
Figure 20–4 A Client-Side Error Message

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 22.4.1, "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 20–5 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 20–5 Displaying Server-Side Error Messages With the ADF Faces Messages Tag



20.7.1 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`.

20.7.2 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.

20.8 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.

20.8.1 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 20–9 shows the `SRDemoErrorHandler Class` that the SRDemo application uses to handle errors.

Example 20–9 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);
  }
  /**
   * Overridden ADF binding framework method to customize the way
   * that Exceptions are reported to the client.
   * Here we set the "append codes" flag to false on each JboException
   * in the exception (and any detail JboExceptions it contains)
   * @param bc BindingContainer
   * @param ex exception being reported
   */
  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 20–10 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 20–10 PrepareModel Method

```
public void prepareModel(LifecycleContext ctx) {
    if (!(ctx.getBindingContext().getErrorHandler() instanceof
          SRDemoErrorHandler)) {
        ctx.getBindingContext().setErrorHandler(new SRDemoErrorHandler(true));
    }
    //etc
    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.

   See the SRDemo for the `reportErrors()` method, which is found in the `frameworkExt.SRDemoPageLifecycle` class. The method customizes the error handling by "pruning" exceptions from the list that the user cannot directly do anything to correct. In general, the errors being pruned out are the "wrapper exceptions" that are added by the ADF Business Components bundled exception mode. For information about the bundled exception mode, see Section 10.5.7, "Understanding Bundled Exception Mode".
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 20–11 shows the `createPageLifecycle` method in the `frameworkExt.SRDemoADFPhaseListener` class.

Example 20–11 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.

20.8.2 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.

21

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 21.1, "Introduction to Adding ADF Bindings to Existing Pages"
- Section 21.2, "Designing Pages for ADF Bindings"
- Section 21.3, "Using the Data Control Palette to Bind Existing Components"
- Section 21.4, "Adding ADF Bindings to Text Fields"
- Section 21.5, "Adding ADF Bindings to Tables"
- Section 21.6, "Adding ADF Bindings to Actions"
- Section 21.7, "Adding ADF Bindings to Selection Lists"
- Section 21.8, "Adding ADF Bindings to Trees and Tree Tables"

21.1 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

21.2 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. You can use control hints on the entity object or view object attributes in your business service to centrally define the labels displayed by an ADF binding. 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 22.4, "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.

21.2.1 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 11.5, "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 22.3, "Using Skins to Change the Look and Feel".

21.2.2 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 21–1 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 21–1 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

21.2.3 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 22.4, "Internationalizing Your Application".

For information about creating JSF and ADF Faces components, see Section 11.4.1, "How to Add UI Components to a JSF Page".

21.2.3.1 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.

21.2.3.2 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.

21.2.3.3 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.

21.2.3.4 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.

21.2.3.5 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.

21.3 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 21.3.2, "What Happens When You Use the Data Control Palette to Add ADF Bindings".)

21.3.1 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:

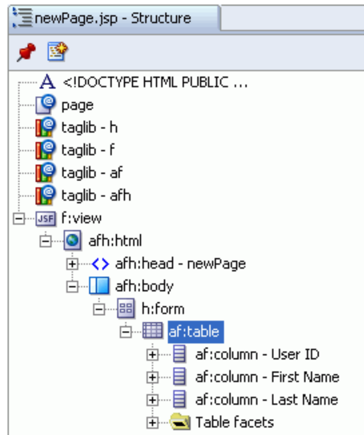
1. With your page displayed in the Design page of the visual editor, open the Structure window.

   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 21–1. When you select a component in the visual editor, JDeveloper simultaneously selects that component tag in the Structure window, as shown in Figure 21–1. 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.

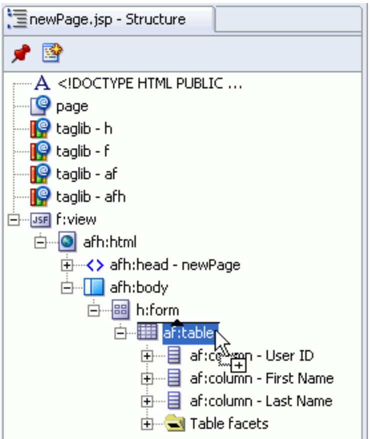
Figure 21–1 Structure Window with Tag Selected



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 12.2.1, "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 21–2. 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.

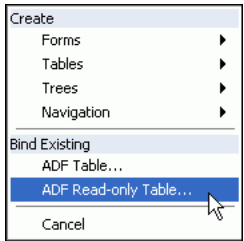
Figure 21–2 Dropping a Data Control Object on a UI Component in the Structure Window



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 21–3.

   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 21–1.

Figure 21–3 Context Menu for Binding to an Existing Component



21.3.2 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 12, "Displaying Data on a Page".

21.4 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.

21.4.1 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 21.3.1, "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 21–1. 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.

21.4.2 What Happens When You Add ADF Bindings to a Text Field

Example 21–1 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 21–1 Text Field Component Before ADF Bindings

```
<af:inputText label="First Name"/>
```

Example 21–2 displays the same text field *after* the `FirstName` attribute of the `StaffList` collection from the SRDemo data control was dropped on it. Notice that the label, which was a static string on the original tag, 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 21–2 Text Field Component After ADF Bindings Are Added

```
<af:inputText label="#{bindings.StaffListFirstName.label}"
              value="#{bindings.StaffListFirstName.inputValue}"
              required="#{bindings.StaffListFirstName.mandatory}"
              columns="#{bindings.StaffListFirstName.displayWidth}">
  <af:validator binding="#{bindings.StaffListFirstName.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 13, "Creating a Basic Page".

21.5 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.

21.5.1 How to Add ADF Bindings to Tables

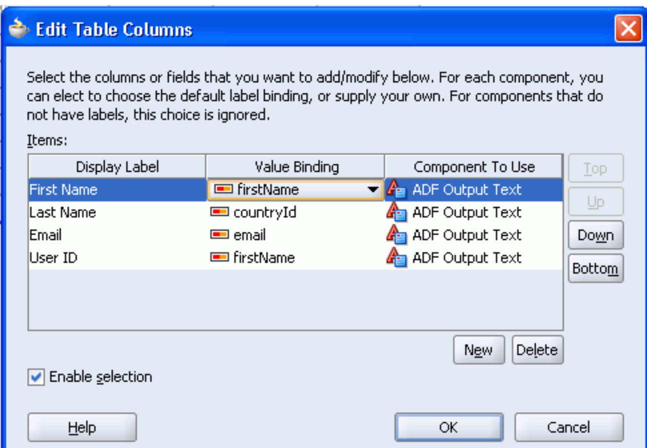
To add ADF bindings to a table, you drag a data collection from the Data Control Palette and drop it on the table tag displayed in the Structure window. For general tips about dropping items from the Data Control Palette onto the Structure window, see Section 21.3.1, "How to Add ADF Bindings Using the 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 21–1. 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 21–4.

Figure 21–4 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 21–5. 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 21–5 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 14, "Adding Tables".

21.5.2 What Happens When You Add ADF Bindings to a Table

Example 21–3 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 21–3 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 21–4 displays the same table *after* the `StaffList` data collection 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 `StaffList` binding object. However, the selection facet and banding from the original table remain intact. The `selectionState` and `selectionListener` attributes have been added with bindings on the `StaffList` 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 21–4 ADF Faces Table After ADF Bindings Are Added

```
<af:table emptyText="#{bindings.StaffList.viewable ? \'No rows yet.\' :
                       \'Access Denied.\'}"
          value="#{bindings.StaffList.collectionModel}" var="row"
          bandingInterval="1"
          rows="#{bindings.StaffList.rangeSize}"
          first="#{bindings.StaffList.rangeStart}">
          selectionState="#{bindings.StaffList.collectionModel.
                            selectedRow}"
          selectionListener="#{bindings.StaffList.collectionModel.
                                makeCurrent}">
  <af:column sortable="false"
             headerText="#{bindings.StaffList.labels.UserId}"
             sortProperty="UserId">
    <af:outputText value="#{row.UserId}">
      <f:convertNumber groupingUsed="false"
                       pattern="#{bindings.StaffList.formats.UserId}"/>
    </af:outputText>
  </af:column>
  <af:column sortable="false"
             headerText="#{bindings.StaffList.labels.UserRole}"
             sortProperty="UserRole">
    <af:outputText value="#{row.UserRole}"/>
  </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.

For more information about databound tables, see Chapter 14, "Adding Tables".

21.6 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 16, "Adding Page Navigation".

21.6.1 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 21.3.1, "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 21–1. 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 17, "Creating More Complex Pages".)

21.6.2 What Happens When You Add ADF Bindings to an Action

Example 21–5 displays a command button *before* the ADF bindings are added.

Example 21–5 ADF Faces Command Button Before ADF Bindings

```
<af:commandButton text="Create User"/>
```

Example 21–6 displays the same button after the `Create` operation from the SRDemo data control was dropped on it. 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 `Create` operation. The `actionListener` detects when the user clicks the button and executes the operation 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 16, "Adding Page Navigation".

Example 21–6 ADF Faces Command Button After ADF Bindings Are Added

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

In addition to adding the bindings to the button, JDeveloper automatically adds an iterator and action binding object to the page definition file.

For more information about databound buttons and links, see Chapter 13, "Creating a Basic Page".

21.7 Adding ADF Bindings to Selection Lists

You can add ADF bindings to any of the selection lists previously shown in Table 21–1. 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.

21.7.1 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 21.3.1, "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 21–1. 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 19.7, "Creating Selection Lists".

21.7.2 What Happens When You Add ADF Bindings to a Selection List

Example 21–7 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 21–7 ADF Faces Single-Selection Dropdown Before ADF Bindings

```
<af:selectOneChoice label="Product:"/>
```

Example 21–8 displays the same list after the `ProdID` attribute in the `ServiceRequests` 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 `ServiceRequestsProdId` 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 21–8 ADF Faces Single-Selection Dropdown After ADF Bindings Are Added

```
<af:selectOneChoice value="#{bindings.ServiceRequestsProdId.inputValue}"
                    label="#{bindings.ServiceRequestsProdId.label}">
  <f:selectItems value="#{bindings.ServiceRequestsProdId.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 the list binding object definition. The `ListDisplayAttrNames` element defines the data collection attributes that populate the values the user sees in the list and is added only if the list is a dynamic list. In dynamic lists, 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 19.7, "Creating Selection Lists".

21.8 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 15, "Displaying Master-Detail Data".

21.8.1 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 21.3.1, "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 15.4, "Using Trees to Display Master-Detail Objects".

21.8.2 What Happens When You Add ADF Bindings to a Tree or Tree Table

Example 21–9 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 21–9 ADF Faces Tree Before ADF Bindings

```
<af:tree value="users" var="service requests">
   <f:facet name="nodeStamp">
      <h:outputText/>
   </f:facet>
</af:tree>
```

Example 21–10 displays the same tree after the `LoggedInUser` data collection from the SRDemo data control was dropped on it. The `LoggedInUser` data collection will populate the root node, and the `ServiceRequestsByStatus` collection was defined as a branch off the root nodes. The binding replaced the `value` attribute with a binding on the `LoggedInUser` 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 21–10 ADF Faces Tree After ADF Bindings Are Added

```
<af:tree value="#{bindings.LoggedInUser.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 tree binding object definition. The `AttrNames` element lists all the attributes available in the collection, but only the attributes in the `nodeDefinition` element are displayed in the tree. The `Accesssors` element defines the view link accessor methods that will be used to retrieve the data that will populate the branches in the node.

For more information about trees and tree tables, see Chapter 15, "Displaying Master-Detail Data".

22

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 22.2, "Changing the Style Properties of a Component"
- Section 22.3, "Using Skins to Change the Look and Feel"
- Section 22.4, "Internationalizing Your Application"

22.1 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

22.2 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.

22.2.1 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 22–1 shows the code on the JSF page for the `outputText` component.

Example 22–1 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);':''}"/>
```

22.2.2 What Happens When You Format Text

As Example 22–1 shows, when you use the Property Inspector to set a style, JDeveloper adds the corresponding code for the component to the JSF page.

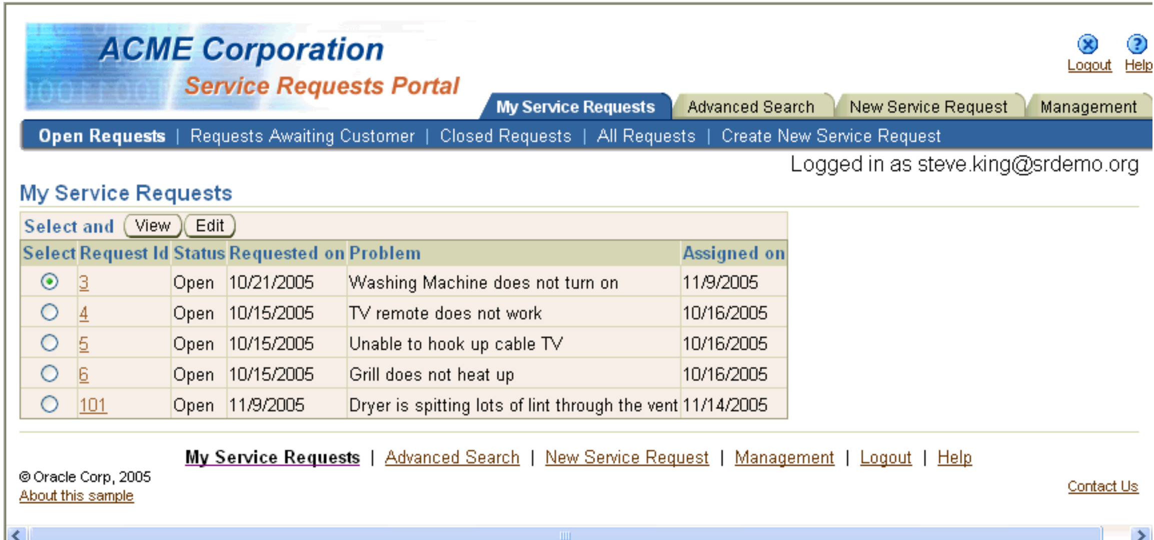
22.3 Using Skins to Change the Look and Feel

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 22–1 shows the SRList page with the Oracle skin applied.

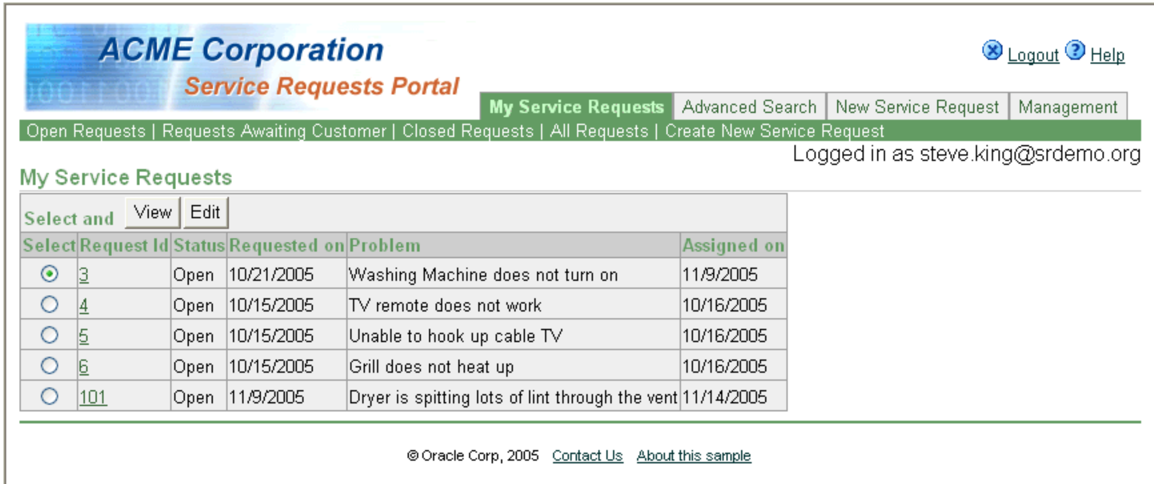
Note: The syntax in a skin style sheet is based on the CSS3 specification. However, many browsers do not yet adhere to this version. At runtime, ADF Faces converts the CSS to the CSS2 specification.

Figure 22–1 The SRList Page Using the Oracle Skin



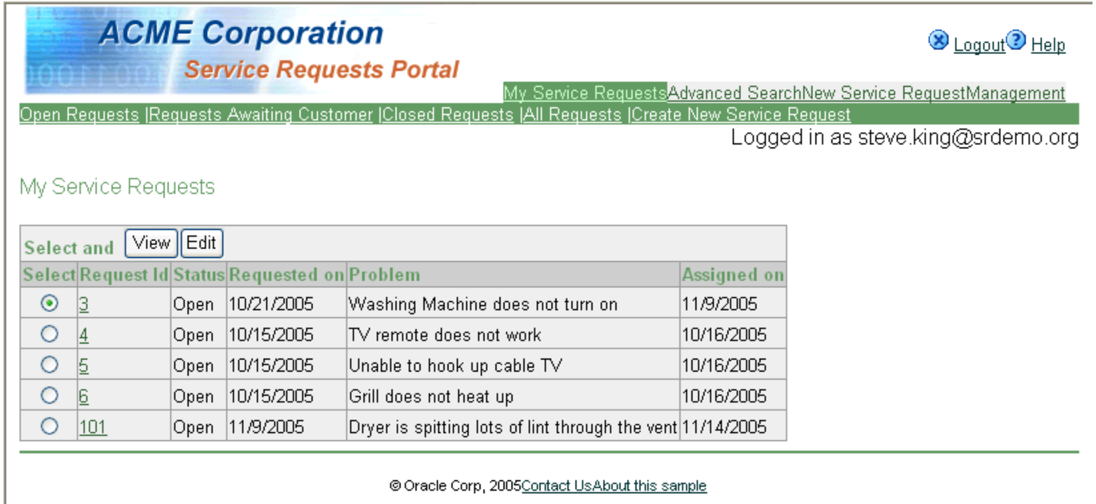
ADF Faces also provides two other skins. The Minimal skin provides some formatting, as shown in Figure 22–2. Notice that almost everything except the graphic for the page has changed, including the colors, the shapes of the buttons, and where the copyright information displays.

Figure 22–2 The SRList Page Using the Minimal Skin



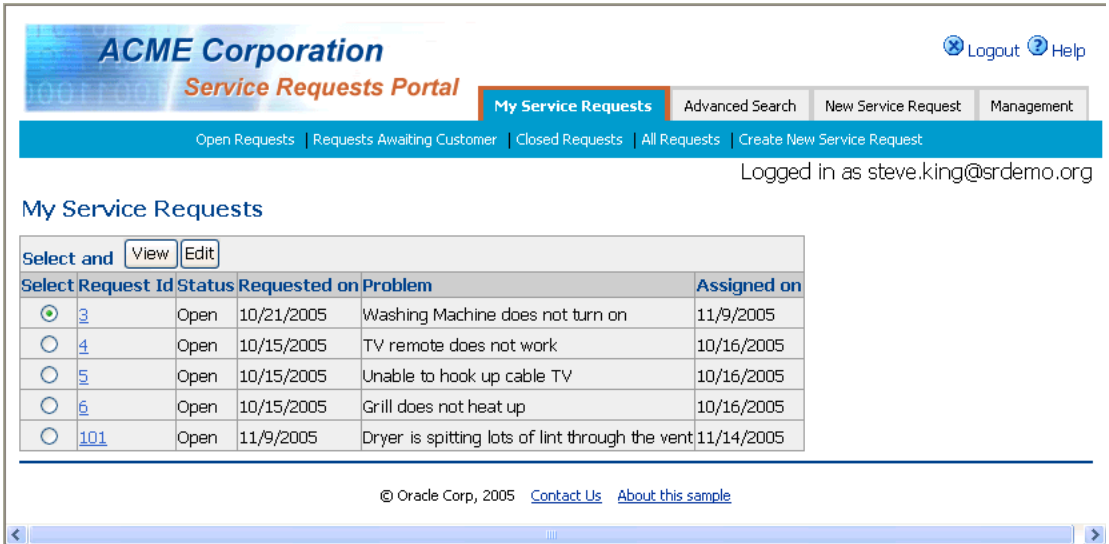
The third skin provided by ADF Faces is the Simple skin. This skin contains almost no special formatting, as shown in Figure 22–3.

Figure 22–3 The SRList Page Using the Simple Skin



The SRDemo application uses a custom skin created just for that application, as shown in Figure 22–4.

Figure 22–4 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 22–4 is determined using the skin's resource bundle. All the included skins use the same resource bundle.

22.3.1 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 22.4, "Internationalizing Your Application".

22.3.1.1 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 22.3.1.2, "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 22–2 shows a selector for an icon.

Example 22–2 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 22–3 shows the `rtl` psuedo class used for an icon.

Example 22–3 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 22–4.

Example 22–4 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 22.3.1.2, "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 22.4.1, "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 22.4.1, "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 22–5.

Example 22–5 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 22–6 shows the entry in the `adf-faces-skins.xml` file for the SRDemo skin.

Example 22–6 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>
```

22.3.1.2 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 22–7 shows the configuration to use the `srdemo` skin family.

Example 22–7 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 22–8 shows a `selectOneChoice` component that takes its selected value, and sets it as the value for the `skinFamily` attribute in `sessionScope`.

Example 22–8 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>
```

22.4 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 Italy, you can localize your application so that when a user's browser is set to use the Italian language, text strings in the application will appear in Italian.

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 Italy (Italian) language, any text contained within the components will automatically display in Italian. For more information on skins and this resource bundle, see Section 22.3.1, "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.

In the SRDemo application, all attribute labels and hints are implemented using control hints on entity objects and view objects instead of using resource bundles (for more information, see Section 5.4, "Defining Attribute Control Hints" and Section 6.5, "Defining Attribute Control Hints"). JSF resource bundles are used for all other UI strings, such as command buttons.

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 22–5 shows the SRList page from the SRDemo application in a browser set to use the English (United States) language.

Figure 22–5 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 22–9. 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 22–9 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']}">
```

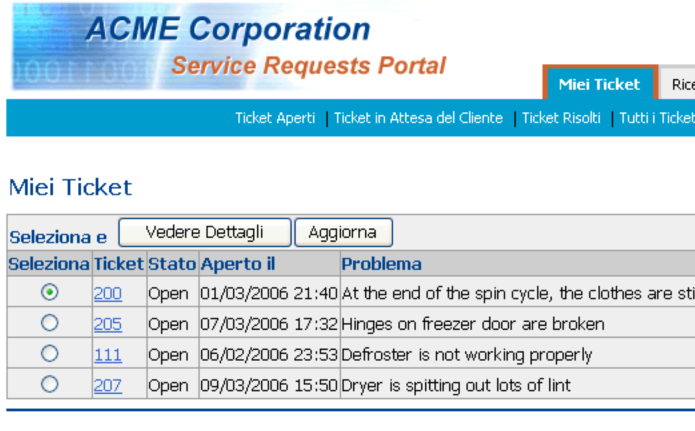
The UIResources resource bundle has an entry in the English language for all static text displayed on each page in the SRDemo application, as well as text for messages and global text, such as generic labels. Example 22–10 shows the keys for the SRList page.

Example 22–10 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 22–6 also shows the SRList page, but with the browser set to use the Italian (Italy) locale.

Figure 22–6 The SRList Page in Italian



Example 22–11 shows the resource bundle version for the Italian (Italy) language, UIResources_it. Note that there is not an entry for the selection facet's title, yet it was translated from "Select" to "Seleziona" automatically. That is because this text is part of the ADF Faces table component's selection facet.

Example 22–11 Resource Bundle Keys for the SRList Page Displayed in Italian

```
#SRList Screen
srlist.pageTitle=Miei Ticket
srlist.menubar.openLink=Ticket Aperti
srlist.menubar.pendingLink=Ticket in Attesa del Cliente
srlist.menubar.closedLink=Ticket Risolti
srlist.menubar.allRequests=Tutti i Ticket
srlist.menubar.newLink=Creare Nuovo Ticket
srlist.selectAnd=Seleziona e
srlist.buttonbar.view=Vedere Dettagli
srlist.buttonbar.edit=Aggiorna
```

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_it.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 Italian (Italy) 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. it_IT
2. it
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

22.4.1 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 Italian version of the `UIResources` bundle is `UIResources_it.properties`. You can add the ISO 3166 uppercase country code (for example it_CH, for Switzerland) 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_it.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 Italian version of the `UIResources` bundle might be `UIResources_it.java`. You can add the ISO 3166 uppercase country code (for example it_CH, for Switzerland) 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 22–12 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 22–12 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 22–13 shows the encoding for the SRList page.

Example 22–13 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 22–14. 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 22–14 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 22–15 shows the code for the View button on the SRList page.

Example 22–15 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.

22.4.2 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 22–16 shows a sample `adf-faces-config.xml` file with all the optional localization elements set.

Example 22–16 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>
```

23 Optimizing Application Performance with Caching

This chapter describes how to add caching support to existing application pages.

This chapter explains the following:

- About Caching
- Using ADF Faces Cache to Cache Content

23.1 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).

23.2 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 23–1 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 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 third 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 23–1 Create New Service Request Page in the SRDemo Application

Example 23–1 shows the code for the first fragment, the start of the page content.

Example 23–1 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 23–2 shows the code for the second fragment, the tabs across the top of the page.

Example 23–2 Menu Tabs Fragment

```
<f:facet name="menu1">
    <afc:cache duration="864000"
               varyBy="request.Session"
    <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 session scope. This attribute specifies to cache a version of the fragment for the entire session. The content is valid across one session regardless of the user.

Example 23–3 shows the code for the last fragment, which displays the **Logout** and **Help** menu items as a global menu fragment.

Example 23–3 Logout and Help Menu Fragment

```
<f:facet name="menuGlobal">
      <f:subview id="globalMenufragment">
        <afc:cache duration="86400">
           <jsp:include page="/app/globalMenu.jspx">
        </afc:cache>
      </f:subview>
</f:facet>
```

Figure 23–2 shows the `SRFaq.jspx` page. Its content is shareable among all users.

Figure 23–2 Frequently Asked Questions Dialog in the SRDemo Application

Example 23–4 shows the code for this page fragment.

Example 23–4 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.

23.2.1 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.

23.2.2 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

23.2.2.1 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 23–5 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 23–5 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.8 for further information about the `j2ee-logging.xml` file

23.2.2.2 AFC Statistics Servlet

The AFC Statistics servlet, shown in Figure 23–3, 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 23–3, resets these statistics, except for **Number of objects in cache**.

Figure 23–3 AFC Statistics Servlet

AFC Statistics

Statistics for Cache Instance

Number of objects in cache:	32
Number of cache hits:	193
Number of cache misses:	40
Number of invalidation requests:	0
Number of documents invalidated:	0

Cache Has been up for :2 day(s) 5 hour(s) 8 minute(s) 48 second(s)

Click here to Reset Stats

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

23.2.2.3 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

23.2.3 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**.

24

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. Finally, it explains how to write and run regression tests for your ADF Business Components-based business services.

This chapter includes the following sections:

- Section 24.1, "Getting Started with Oracle ADF Model Debugging"
- Section 24.2, "Correcting Simple Oracle ADF Compilation Errors"
- Section 24.3, "Correcting Simple Oracle ADF Runtime Errors"
- Section 24.4, "Understanding a Typical Oracle ADF Model Debugging Session"
- Section 24.5, "Setting Up Oracle ADF Source Code for Debugging"
- Section 24.5, "Debugging the Oracle ADF Model Layer"
- Section 24.6, "Tracing EL Expressions"
- Section 24.8, "Regression Testing an Application Module With JUnit"

24.1 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 24.2, "Correcting Simple Oracle ADF Compilation Errors"
- Section 24.3, "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 24.4, "Understanding a Typical Oracle ADF Model Debugging Session" to configure the ADF Logger to display detailed trace messages.

As of June 28th, 2005, supported Oracle ADF customers can request Oracle ADF source code from Oracle Worldwide Support. This can make debugging Oracle ADF Business Components framework code a lot easier. Read Section 24.5, "Setting Up Oracle ADF Source Code for Debugging" to understand how to configure JDeveloper to use the Oracle ADF source code.

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 24.5, "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 24.6, "Tracing EL Expressions" to work with JSF trace messages.

JDeveloper provides integration with JUnit for your ADF Business Components application through a wizard that generate regression test cases. Read Section 24.8, "Regression Testing an Application Module With JUnit" to understand how to write test suites for your application.

24.2 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 24.3, "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 12.6.2, "How to Use the Expression Builder".

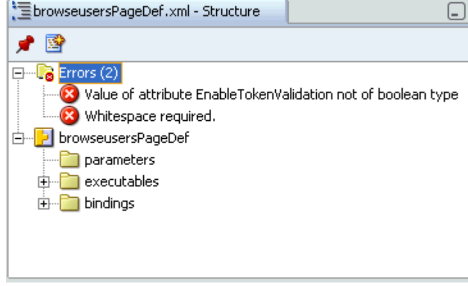
Example 24–1 illustrates simple compilation errors contained in the page definition file: "`fase`" instead of "`false`" and "`IsQueriable="false"/`" instead of "`IsQueriable="false"/>`" (missing a closing angle bracket).

Example 24–1 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"/
```

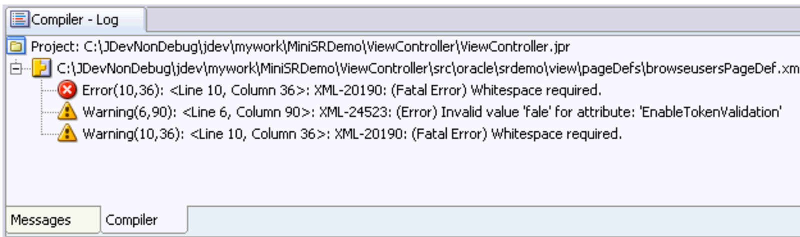
The Structure window for the above errors would display as shown in Figure 24–1.

Figure 24–1 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 24–2.

Figure 24–2 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.

24.3 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 24.6, "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 24.4, "Understanding a Typical Oracle ADF Model Debugging Session".

24.4 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 24.4.2, "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 24.5, "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.

24.4.1 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.

24.4.2 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.

24.4.3 Debugging an Application Module Using the Tester

Often you will find it useful to debug the ADF Business Components in your application without having to run the user interface of your ADF application. You are likely already aware that you can select any application module in the Application Navigator, and choose **Test** from its right-mouse context menu to launch the Business Components Tester tool.

What you might not know is that you can also launch the Tester tool in the debugger. This can be extremely useful and can make debugging ADF Business Components applications even easier than having to start up the complete frontend GUI of your application.

To launch an ADF Application Module in the tester in debug mode, do the following:

1. In the Application Navigator, select the desired application module.
2. In the Structure Window, expand the **Sources** folder, and select the Java implementation class for your application module.

   If you application module is called `MyModule`, its implementation class will be named `MyModuleImpl.java`.
3. Select **Debug** from the context menu on that application module implementation class to launch the debugger.

You'll notice that the `main()` method of your application module Java class looks like this:

```
public static void main(String[] args) {
  launchTester("com.yourcompany.yourapp.model", /* package name */
               "MyModuleLocal");                /* Configuration Name */
}
```

If you need to launch the application module in the debugger using a different configuration than the one indicated (e.g. `MyModuleLocal` in the code sample above), just change the string that gets passed as the second argument to `launchTester()` to be the name of the configuration you want to use instead.

> **Tip:** When you debug the application module implementation class (by selecting the `.java` file in the System Navigator), you don't have to change configuration name since it is written into the `main()` method as one of the two strings that get passed to the debugger. However, when you debug the application module using the Business Components Tester (by choosing **Test** from the context menu on the application module node in the Application Navigator), your configuration must use a JDBC URL connection (not a JDBC DataSource). For example, in the SRDemo application, the configuration `SRServiceLocalTesting` lets you run the Tester. By default, the selected configuration will be `SRServiceLocal`. If you do not change the configuration choice, an alert will indicate that the Tester cannot use a JDBC DataSource connection.

24.4.4 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.

24.4.5 Editing Breakpoints to For Improved 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.

24.4.6 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.

24.4.7 Interesting Oracle ADF Breakpoints to Set

Interesting breakpoints you can set in Oracle ADF source code when you're trying to debug a problem are:

- Exception breakpoint for `oracle.jbo.JboException`

  When you're not sure where to start, this is the base class of all ADF Business Components runtime exceptions.

- Exception breakpoint for `oracle.jbo.DMLException`

  This is the base class for exceptions originating from the database, like a failed DML operation due to an exception raised by a trigger or by a constraint violation.

- Source breakpoint in the `doIt()` method of `JUCtrlActionBinding` class (`oracle.jbo.uicli.binding` package).

  This is the method that will execute when any ADF action binding is invoked, and you can step into the logic and look at parameters if relevant.

- Method breakpoint in the `oracle.jbo.server.ViewObjectImpl.executeQueryForCollection` method.

  This is the method that will be called when a view object executes its SQL query.

- Method breakpoint in the `oracle.jbo.server.ViewRowImpl.setAttributeInternal` method.

  This is the method that will be called when any view row attribute is set.

- Method breakpoint in the `oracle.jbo.server.EntityImpl.setAttributeInternal` method.

  This is the method that will be called when any entity object attribute is set.

By looking at the stack window when you hit these breakpoints, and stepping through the source you can get a better idea of what's going on.

24.4.8 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.

24.5 Setting Up Oracle ADF Source Code for Debugging

You can obtain complete source code for Oracle ADF by opening a service request with Oracle Worldwide Support and requesting it. This section explains how to use Oracle ADF source code for debugging purposes inside the Oracle JDeveloper environment.

Note: The instructions below assume you have extracted the `adf_1013_3673_source.zip` archive into the root `C:\` directory to create a `C:\adf_1013_3673_source` directory, and that your JDeveloper home directory is `C:\jdev1013`. If you have JDeveloper installed in a different directory, you'll need to substitute that instead.

24.5.1 Setting Up the ADF Source System Library

The first step requires you to define a system library that points to the `adfsource.zip` file. To accomplish this task, follow these steps:

1. Select **Tools | Manage Libraries** to display the Manage Libraries dialog.
2. With the Libraries tab selected, click on the System Libraries folder in the tree at the left and click the **New** button
3. Enter a library name. For example, `ADF Source`.
4. Enter a source path of `C:\adf_1013_3673_source\adfsource.zip`

Note: Notice the Class Path field is blank. You only need to provide a value for the Source Path field.

---

5. Uncheck the **Deployed by Default** checkbox.
6. Click **OK** to close the Manage Libraries dialog.

24.5.2 Adding the ADF Source Library to a Project

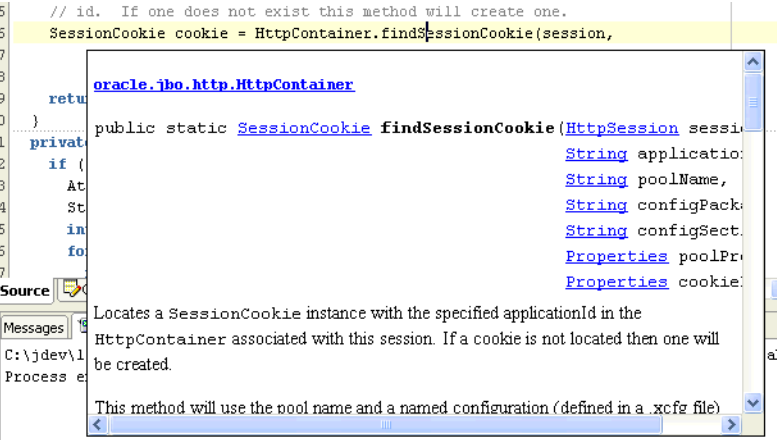
In order to debug any application using Oracle ADF inside JDeveloper, you just need to add your new `ADF Source` system library created above to the library list of the project you plan to debug. To accomplish this task, follow these steps:

1. Select the project that you will be debugging in the Application Navigator.
2. Choose **Project Properties** from the right mouse menu to show the Project Properties dialog.
3. Select the **Libraries** category in the tree at the left (by default, under the *Development* profile).
4. Select the **ADF Source** library in the Available Libraries list on the left
5. Click the (>) button to move this library into your project's Selected Libraries list.
6. Click **OK** to close the Project Properties dialog.

24.5.3 Seeing Better Information in the Code Editor

Once you have added the ADF Source library to your project, you instantly have access to the helpful **Quick JavaDoc** feature ([`Ctrl`]+[`D`]) that the JDeveloper Code Editor makes available. As shown in Figure 24–3, invoking the Quick JavaDoc on a method like `findSessionCookie()`.

Figure 24–3 Using Quick JavaDoc on ADF API's in Code Editor



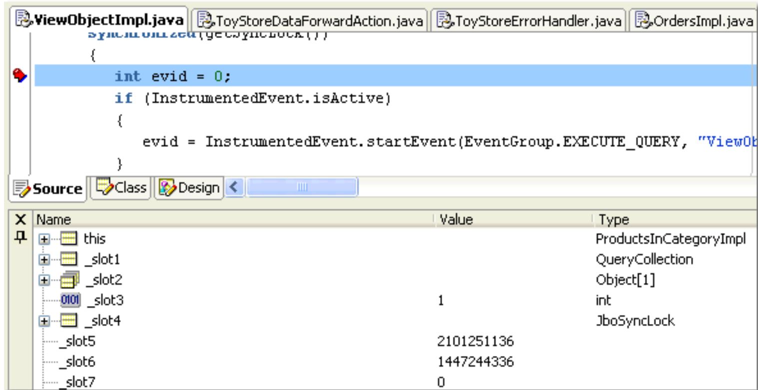
24.5.4 Setting Breakpoints and Debugging

After performing the two steps above, you can debug any Oracle ADF code for the current project in the same way as you have been doing for your own Java code. This means that you can press [`Ctrl`]+[`Minus`] to type in any class name in Oracle ADF, and JDeveloper will open its source file automatically so you can set breakpoints as desired.

24.5.5 Seeing Better Symbol Information Using Debug Libraries

When debugging Oracle ADF source, as shown in Figure 24–4 by default you will not see symbol information for parameters or member variables of the currently executing method.

Figure 24–4 Local Symbols Are Hard To Understand Without Debug Libraries

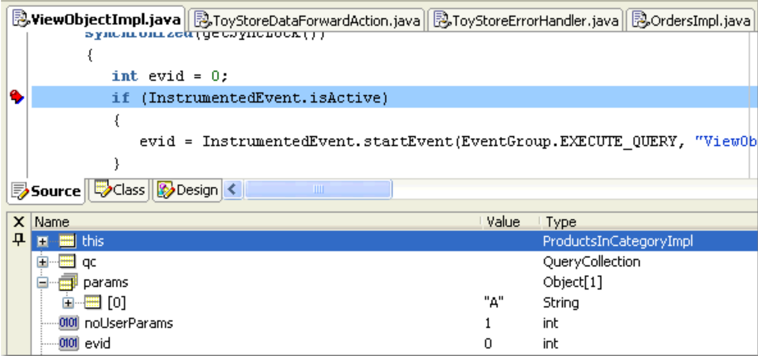


This can make debugging a little less useful. You can make the situation better by using the debug versions of the ADF JAR files supplied along with the source, while debugging in your development environment.

Note: The supplied debug libraries are not recommended for use in a test or production environment since they typically have slightly slower runtime performance than the optimized JAR files shipped with JDeveloper.

The `debuglib` subdirectory of `C:\adf_1013_3673_source` contains versions of Oracle ADF JAR files from JDeveloper 10.1.3 Build 3673 that have been compiled with additional debug information. When you use these debug JAR files instead of the default optimized JAR files you will see all of the information in the debugger as shown in Figure 24–5.

Figure 24–5 See Local Symbol Information in Debugger Using Debug Libraries



In order to use these debug JAR files, perform the following steps:

1. Make sure JDeveloper 10g is not running. If it's currently running, exit from the product before proceeding with the subsequent steps.
2. Make a `backup` subdirectory of all existing optimized JAR files in the `./BC4J/lib` directory of your JDeveloper installation:

```
C:\> cd jdev1013\BC4J\lib
C:\jdev1013\BC4J\lib> mkdir backup
C:\jdev1013\BC4J\lib> copy *.jar backup
```

3. Notice that the debuglib subdirectory of C:\adf_1013_3673_source contains debug versions of the same ADF JAR files as the ones you just made a backup of. The only difference is that each debug JAR file has the suffix _g.jar instead of just.jar.

4. For each ADF library that you want to have debug symbols for while debugging, copy the _g.jar version of the matching library over the existing, corresponding library in the C:\jdev1013\BC4J\lib directory. This is safe to do since you made a backup of the optimized JAR files in the backup directory in step 2 above.

   For example, for the main ADF Business Components runtime JAR file (bc4jmt.jar), you would copy the C:\adf_1013_3673_source\debuglib\bc4jmt_g.jar to C:\jdev1013\BC4J\lib\bc4jmt.jar.

Since debug libraries typically run a little slower than libraries compiled without debug information, this diagnostic message is to remind you not to use debug libraries for performance timing.

```
*************************************************************************
*** WARNING: Oracle BC4J debug build executing - do not use for timing ***
*************************************************************************
```

To change back to the optimized libraries, simply copy the JAR file(s) in question from the ./BC4J/lib/backup directory back to the ./BC4J/lib directory.

24.6 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 13.2.3, "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.

24.6.1 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.

24.6.1.1 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.

```
public void doFilter(
   ServletRequest request, ServletResponse response, FilterChain chain)
   throws IOException, ServletException
```

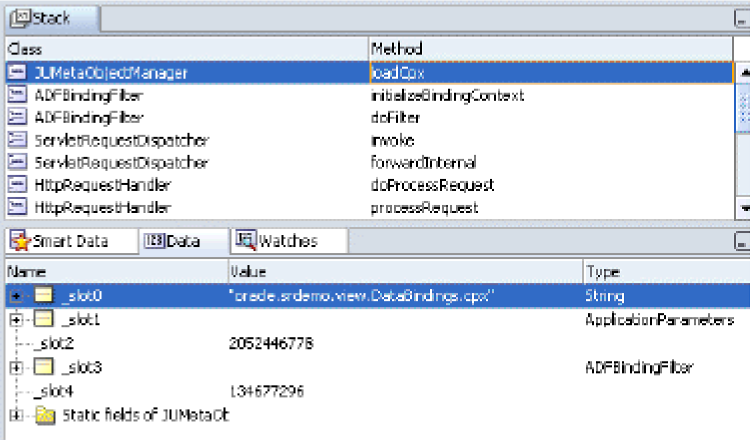
2. Set another break on `ctx.get(BindingContext.IS_INITIALIZED)` and step into this method.

```
// don't synchronized this in the filter lock.  can be expensive.
synchronized(sessionLock)
{
   BindingContext ctx = (BindingContext)sessionScope.get(
      BindingContext.CONTEXT_ID);

   if (ctx == null || ctx.get(BindingContext.IS_INITIALIZED) == null)
   {
      initializeBindingContext((HttpServletRequest)request);
   }
}
```

3. In the `oracle.jbo.uicli.mom.JUMetaObjectManager` class, set a break on `chain.getClientProjectExtension()` and step into this method.

```
public static void loadCpx(String sResource, Map userParams)
{
   String projExt = getClientProjectExtension();
```

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 24–2 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 24–2 Sample web.xml Servlet Context Parameter

```
<context-param>
        <param-name>CpxFileName</param-name>
        <param-value>oracle.srdemo.view.DataBindings</param-value>
</context-param>
```

24.6.1.2 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:

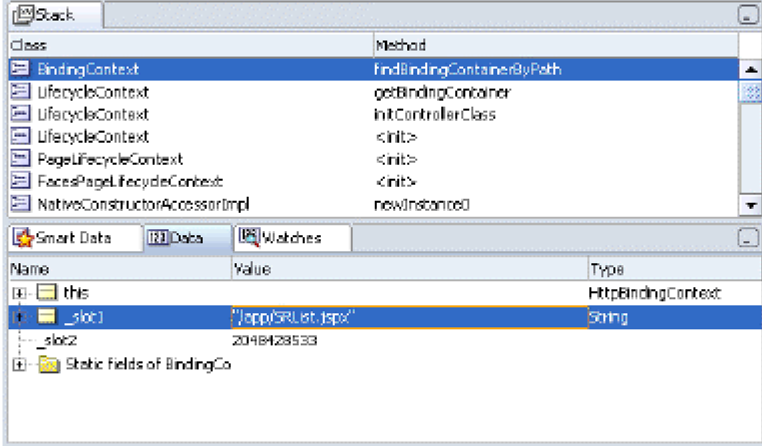
```
oracle.jbo.NoDefException: JBO-25002: Definition
oracle.<path>.pageDefs.<pagedefinitionName> of type Form Binding
Definition not found
```

To debug creating the binding container for the web page:

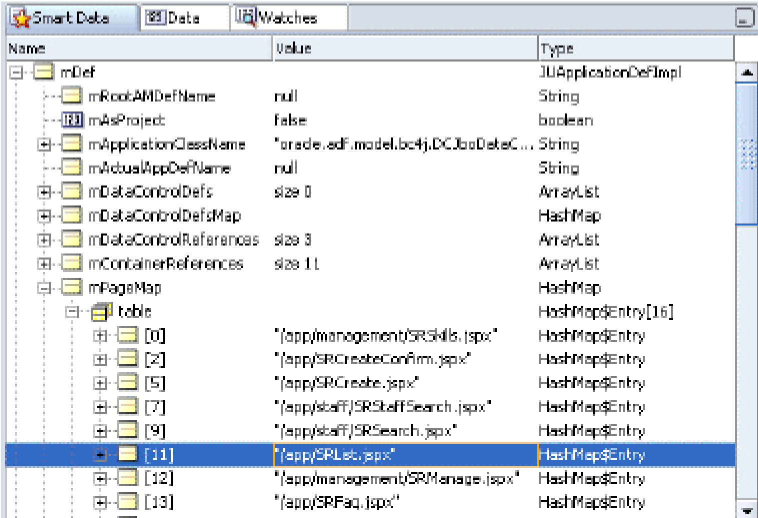
1. In the `oracle.adf.model.BindingContext` class, set a break on `findBindingContainerIdByPath()` and step into this method.

```
public DCBindingContainer findBindingContainerByPath(String p
{
   if (mDef == null)
   {
      initDef();
   }
   if (mDef != null)
   {
      String bcId;
      if ((bcId = mDef.findBindingContainerIdByPath(path, th
      {
         return findBindingContainer(bcId);
      }
   }
   return null;
}
```

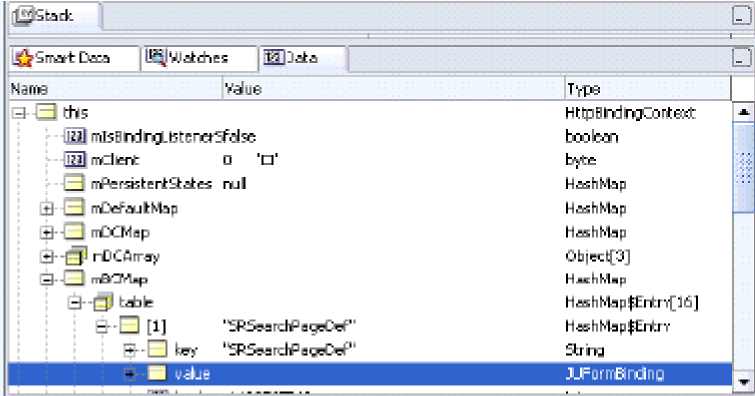
2. Look for the name of the databound web page associated with the binding container in the Data window.



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 24–3 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 24–3 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"/>
    <BC4JDataControl id="SRService" Package="oracle.srdemo.model"
            FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
            SupportsTransactions="true" SupportsFindMode="true"
            SupportsRangesize="true" SupportsResetState="true"
            SupportsSortCollection="true"
            Configuration="SRServiceLocal" syncMode="Immediate"
            xmlns="http://xmlns.oracle.com/adfm/datacontrol"/>
  </dataControlUsages>
</Application>
```

24.6.2 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.

24.6.2.1 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.

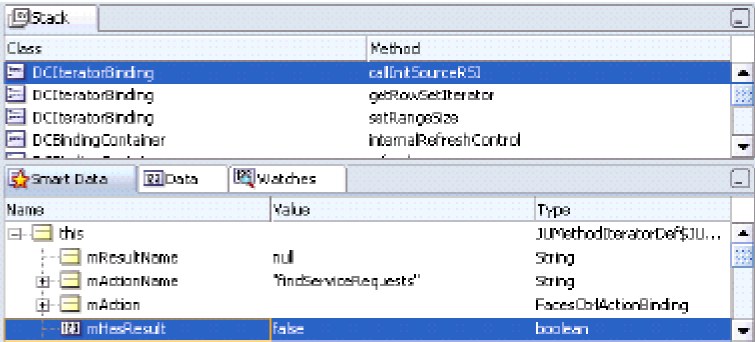
```
protected void internalRefreshControl (int refreshFlag, boolean executeIfNeeded)
{
   //this should register datacontrols that are part of this panelBinding into
   //context so that at the end of refresh, context could call sync on
   //datacontrols
   setRefreshed(true);
   mTransientRefreshType = refreshFlag;
```

   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.

```
private RowSetIterator callInitSourceRSI()
{
   DCIteratorBindingDef def = getDef();
   RowSetIterator rsi = null;
   {
      try
      {
         mCallInitSourceRSI = false;
         rsi = initSourceRSI();
         if (mSC == null && def != null && def.getSortCriteri
         {
```

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.

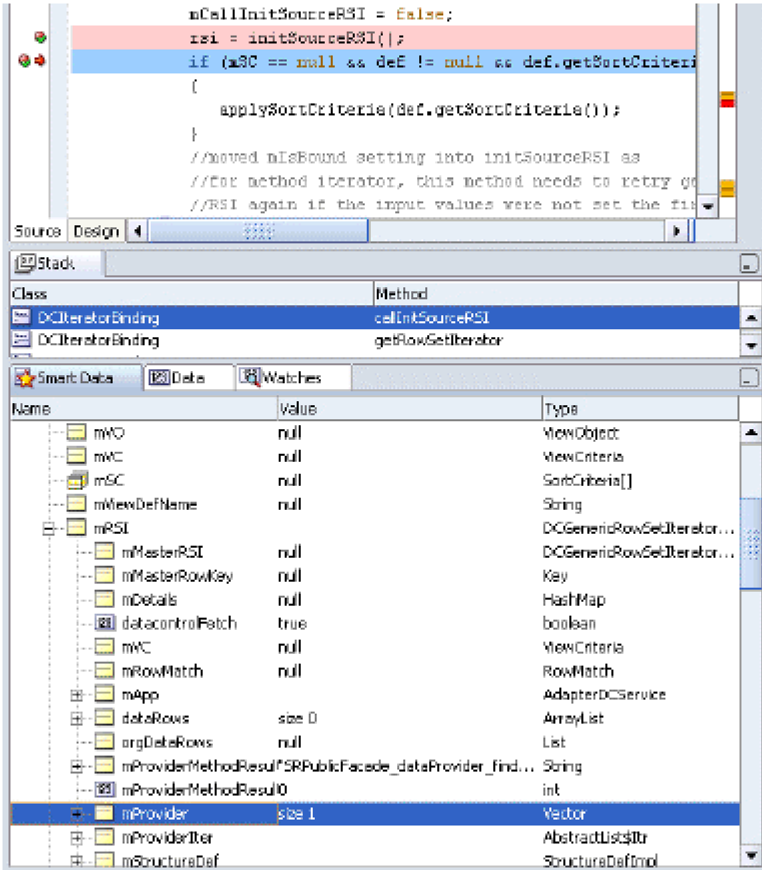
```
protected RowSetIterator initSourceRSI()
{
    //do not implicitly execute a method rsi. check if
    //is options allow refreshing by default
    DCExecutableBindingDef def = getDef();
    boolean isRefreshable = (def != null && def.isRefreshable(getBindingCon
```

2. Set a break on `invokeMethodAction()` to halt processing and step into the method.

```
//if invokeMethodAction has been called then mHasRes
if (!mHasResult && result == null)
{
    result = invokeMethodAction();
    if (result == null)
    {
```

Note that if the method returns a valid collection or a bean, then that object becomes the datasource for the rowset iterator that this iterator binding is bound to. For bean data controls, an instance of DCRowSetIteratorImpl is created to provide the rowset iterator functionality for the iterator binding to work with. (Note that for ADF Business Components, this method would ideally return a ADF Business Components rowset iterator so that ADF Business Components can manage the state of the collection.)

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.



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.6.1, "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 24–4 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 detail iterator positioned after its master binding iterator.

Example 24–4 Sample Page Definition Executables

```
<executables>
    <iterator id="StaffListIterator" Binds="StaffList" RangeSize="10"
              DataControl="SRService"/
    <iterator id="StaffExpertiseAreasIterator" Binds="StaffExpertiseAreas"
              RangeSize="10" DataControl="SRService"/>
<executables>
```

24.6.2.2 Fixing Render Value Errors Before Submit

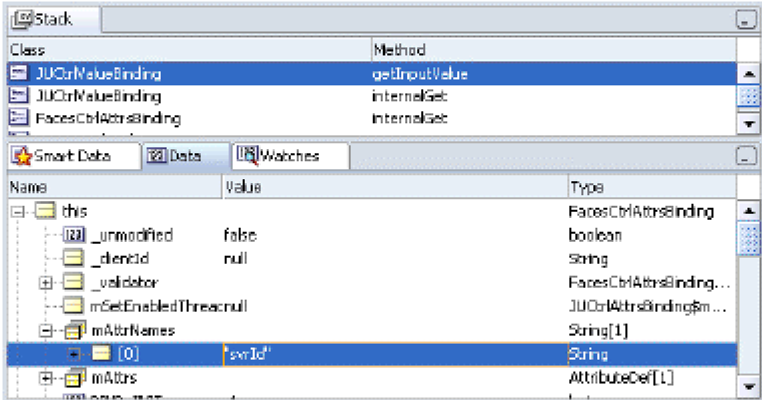
During the prepareRender phase of the ADF lifecycle, the bindings determine the data to display, and properties on the bindings determine the conditions in which to display the data. When the web page is rendered the first time, each EL expression that points to a binding gets resolved by the `BindingContainer` 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:

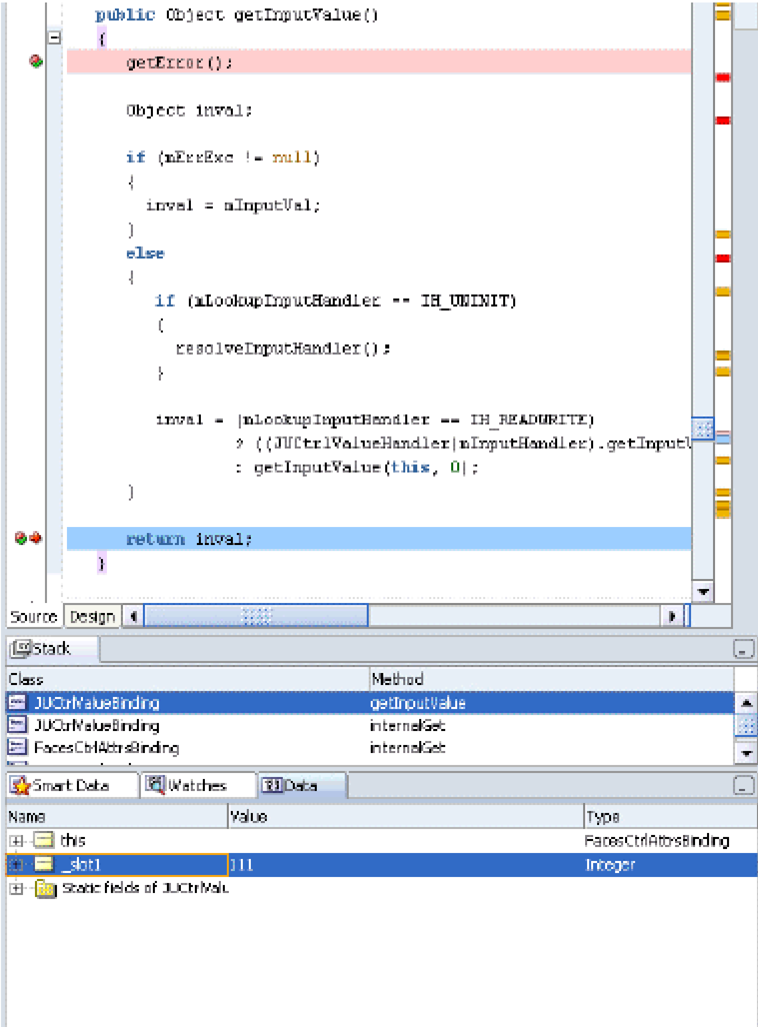
1. In the `oracle.jbo.uicli.binding.JUCtrlValueBinding` class, set a break in `getInputValue()` and step into the method.

```
public Object getInputValue()
{
   getError();

   Object inval;

   if (mErrExc != null)
   {
     inval = mInputVal;
   }
   else
   {
      if (mLookupInputHandler == IH_UNINIT)
      {
        resolveInputHandler();
      }

      inval = (mLookupInputHandler == IH_READWRITE)
              ? ((JUCtrlValueHandler)mInputHandler).getInput\
              : getInputValue(this, 0);
   }

   return inval;
}

public Object getInputValue(JUCtrlValueBinding binding, int i
{
   return this.getInputValueInRow(binding, internalGetRow());
}
```

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 24–5 shows a sample `pagedef.xml` file for the SRDemo application.

Example 24–5 Sample Page Definition Value Bindings

```
<bindings>
    <attributeValues IterBinding="GlobalsIterator" id="ProblemDescription">
      <AttrNames>
        <Item Value="ProblemDescription"/>
      </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="GlobalsIterator" id="ProductId">
      <AttrNames>
        <Item Value="ProductId"/>
      </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="GlobalsIterator" id="ProductName">
      <AttrNames>
        <Item Value="ProductName"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="FirstName"
                     IterBinding="LoggedInUserIterator">
      <AttrNames>
        <Item Value="FirstName"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="LastName"
                     IterBinding="LoggedInUserIterator">
      <AttrNames>
        <Item Value="LastName"/>
      </AttrNames>
    </attributeValues>
    ...
</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.

24.6.3 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.

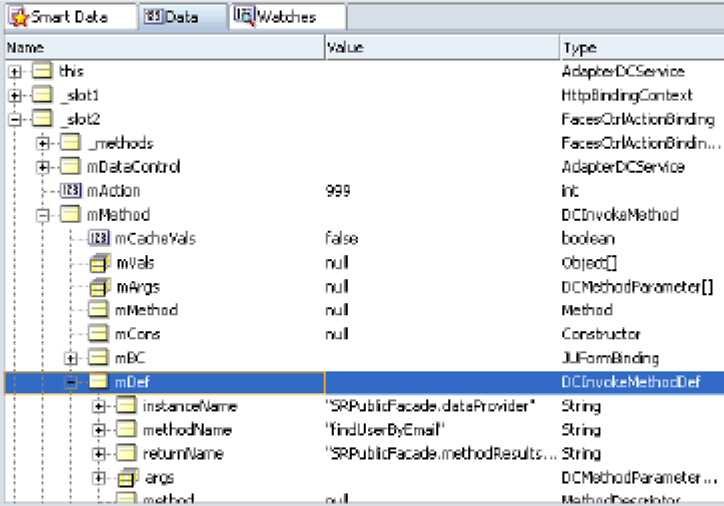
The entry point for action and method execution is the `DCDataControl.invokeOperation()` method. Although `JUCtrlActionBinding.invoke()` is another potential entry point, method iterator bindings also use it to invoke methods implicitly. Instead, debugging on `DCDataControl.invokeOperation()` allows 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:

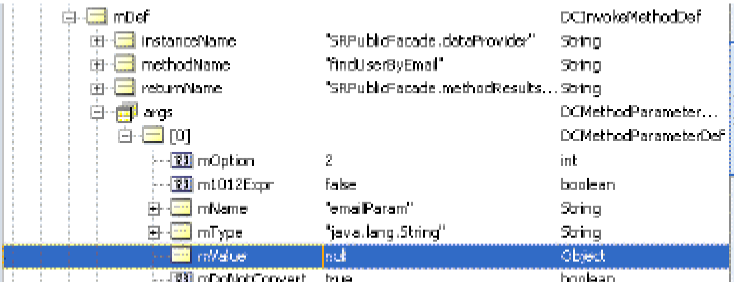
1. In the `oracle.adf.model.binding.DCDataControl` class, set a break on `invokeOperation()` as the entry point to debug an action or method invocation.

```
protected Object invokeMethod(DCInvokeMethod method, Operati
{
   return method.invokeMethod(this, params);
}

public boolean invokeOperation(Map ctx, oracle.binding.Opera
{
   if (action instanceof oracle.jbo.uicli.binding.JUCtrlActi
   {
      ((oracle.jbo.uicli.binding.JUCtrlActionBinding)action)
      return true;
   }
```

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.



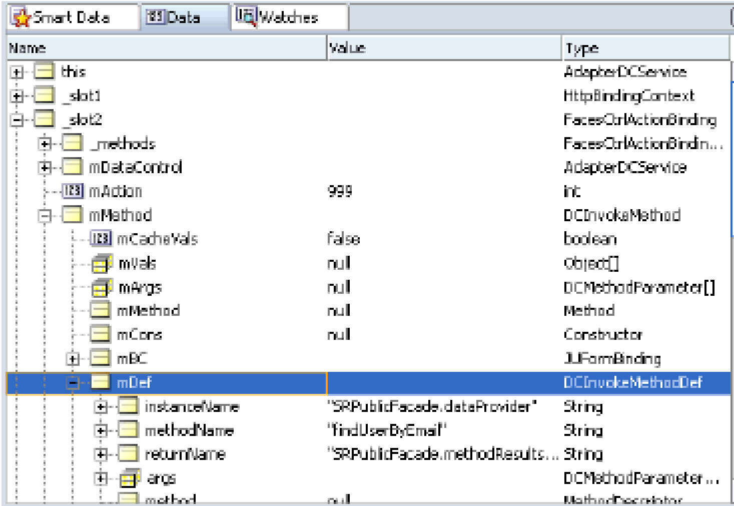
To debug a custom method invocation for the binding container:

1. In your class, set a breakpoint on the desired custom method.

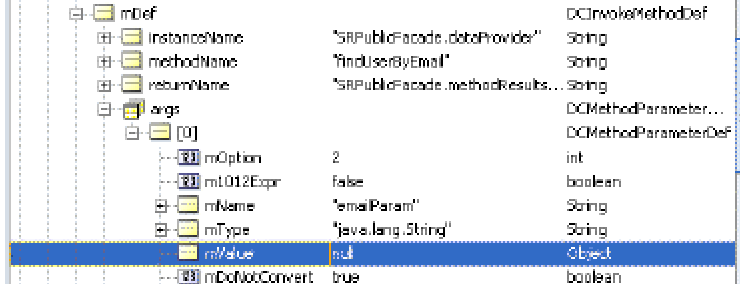
2. In `oracle.adf.model.generic.DCGenericDataControl` class, set a break on `invokeMethod()` to halt processing before looking into the Data window.

```
protected Object invokeMethod(DCInvokeMethod method, Operati
{
   if (mAdapter != null)
   {
      DCInvokeMethod methodInfo = null;
      Object params[] = null;
      if (mAdapter.invokeOperation(getBindingContext(), acti
      {
         Object result = ((JUCtrlActionBinding)action).getRe

         cacheMethodResult(method, result, params);
         return result;
      }
   }
   return super.invokeMethod(method, action, paramsMap);
}
```

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.6.1, "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 24–6 shows a sample of the custom method binding definitions in the `pagedef.xml` file for the SRDemo application.

Example 24–6 Sample Page Definition Executables and Action Bindings

```
<executables>
    <invokeAction id="AlwaysFind" Binds="Find" Refresh="ifNeeded"
                  RefreshCondition=
                   "${bindings.SearchServiceRequestsIterator.findMode == false}"/>
    <invokeAction id="insertBlankViewCriteriaRowIfThereAreNone" Binds="Create"
                  Refresh="renderModel"
                  RefreshCondition=
                     "${bindings.SearchServiceRequestsIterator.findMode and
                 bindings.SearchServiceRequestsIterator.estimatedRowCount == 0}"/>
 ...
</executables>
```

24.7 Tracing EL Expressions

EL is not well supported with exceptions to inform you of specific failures. However, Example 24–4 shows one common exception you are likely to see when the resolver is unable to completely evaluate the expression.

Example 24–7 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.

24.8 Regression Testing an Application Module With JUnit

Testing your business services is an important part of your application development process. By creating a set of JUnit regression tests that exercise the functionality provided by your application module, you can ensure that new features, bug fixes, or refactorings do not destabilize your application. JDeveloper's integrated support for creating JUnit regression tests makes it easy to follow this best practice. Its integrated support for running JUnit tests means that any developer on the team can run the test suite with a single mouse click, greatly increasing the chances that every team member can run the tests to verify their own changes to the system. Furthermore, by using JDeveloper's integrated support for creating and running Apache Ant build scripts, you can easily incorporate running the tests into your nightly build process as well. This section explains how to create a JUnit test for your application module, how to run it, and how to integrate the tests into an Ant build script.

24.8.1 How to Create a JUnit Test Suite for an Application Module

Typically you will create a separate project to contain your regression tests. For example, the SRDemo application has it JUnit regression test suite in the UnitTests project, while the business components that comprise its `SRService` application module belong to the DataModel project. After creating a new project to contain your JUnit test suite, you can use the Create Business Components Test Suite wizard in the context of that new project to create a JUnit test suite for an application module. You can find this wizard in the **General > Unit Tests (JUnit)** category in the New Gallery.

> **Tip:** If you don't see the Business Components Test Suite wizard, use JDeveloper's **Help | Check for Updates** feature to install the **JUnit Integration for Business Components** extension before continuing.

When the Test Suite Wizard dialog appears, perform the following steps on the **Select Application** page:

1. Select the business components project in your workspace that contains the application module.
2. Select the application module in that project for which you want to create a test suite.
3. Enter a configuration name to use for running the tests.
4. Click **Finish** to create the test suite.

24.8.2 What Happens When You Create a JUnit Test Suite for an Application Module

When you create a JUnit test suite for an application module using the wizard, JDeveloper updates the current project to have a dependency on the project containing the application module. In addition, assuming the application module for which you created the test suite were named `devguide.example.ExampleModule`, it generates the following skeleton classes in the `devguide.example.test` package:

- A test suite class named `ExampleModuleAllTests`,
- A test fixture class named `ExampleModuleConnectFixture`
- A test class named *`ViewInstanceName`*`Test` for each view object instance in the application module's data model

In this example, you would run the test suite at any time by running the `ExampleModuleAllTests` class.

24.8.3 What You May Need to Know

24.8.3.1 Test Suite Class Adds Test Cases to the Suite

By convention, a JUnit test suite is a class containing a `public static` method named `suite()` that returns an object that implements the `Test` interface in the `junit.framework` package. Typically, this will be an instance of the `TestSuite` class in that same package. The generated `ExampleModuleAllTests` class follows this convention to create and return an instance of this `TestSuite` object. As shown in Example 24–8, before returning the test suite it calls the `addTestSuite()` method to add one or more test case classes to the suite.

Example 24–8 Test Suite Class Adds Test Cases to the Suite

```
package devguide.example.test;
import junit.framework.Test;
import junit.framework.TestSuite;
public class ExampleModuleAllTests {
  public static Test suite() {
    TestSuite suite;
    suite = new TestSuite("ExampleModuleAllTests");
    suite.addTestSuite(ProductsTest.class);
    suite.addTestSuite(ServiceHistoriesTest.class);
    suite.addTestSuite(ServiceRequestsTest.class);
    // etc.
    return suite;
  }
}
```

24.8.3.2 Test Fixture Class Encapsulates Access to the Application Module

The generated `ExampleModuleConnectFixture` is a JUnit test fixture that encapsulates the details of acquiring and releasing an application. It contains a `setUp()` method that uses the `createRootApplicationModule()` method of the `Configuration` class to create an instance of an application module. Its `tearDown()` method calls the matching `releaseRootApplicationModule()` method to release the application module instance.

Each test case class contains a `setUp()` and `tearDown()` method that JUnit invokes to allow initializing resources required by the test case and to later clean them up. These test case methods invoke the corresponding `setUp()` and `tearDown()` methods to prepare and clean up the text fixture for each test case execution. Any time a test in the test case needs access to the application module, it access it using the test fixture's `getApplicationModule()` method. This returns the same application module instance, saved in a member field of the test fixture class, between the initial call to `setUp()` and the final call to `tearDown()` at the end of the test case.

24.8.3.3 JUnit Tests for an Application Module Must Use a JDBC URL Connection

The application module configuration that you use for a JUnit test suite needs to use a JDBC URL connection, rather than a JDBC datasource. This is due to the fact that JDBC datasources are defined by the J2EE application server and can only be referenced in the context of an application running inside the server. Your JUnit tests run outside the server environment. For example, the SRDemo application has two configurations defined for its `SRService` application module. The `SRServiceLocal` configuration uses a JDBC datasource. It is referenced in the `DataBindings.cpx` file of the `UserInterface` project for use in the SRDemo application. In contrast, the SRDemo's JUnit tests reference the `SRServiceLocalTesting` configuration. This uses a JDBC URL connection instead of a datasource.

24.8.3.4 Test Case Classes Contain One or More Test Methods with Assertions

Each generated test case can contain one or more test methods that the JUnit framework will execute as part of executing that test case. You can add a test to the test case simply by creating a public void method in the class whose name begins with the prefix `test`. For example, it might look like this:

```
// In ViewInstanceNameTest.java test case class
  public void testSomeMeaningfulName() {
  // test assertions here
  }
```

The wizard generates skeleton test case classes for each view object instance in the data model, each of which contains a single test method named `testAccess()`. This method contains a call to the `assertNotNull()` method to test that the view object instance exists.

Your own testing methods can use any of the programmatic APIs available in the `oracle.jbo` package to work with the application module and view object instances in its data model. You can also cast the `ApplicationModule` interface to a custom interface to have your tests invoke your custom service methods as part of their job. During each test, you will call one or more `assert`*`Xxxx`*`()` methods provided by the JUnit framework to assert what the expected outcome of a particular expression should be. When you run the test suite, if any of the tests in any of the test cases contains assertions that fail, JDeveloper's JUnit Test Runner window displays the failing tests with a red failure icon.

24.8.4 Running a JUnit Test Suite as Parts of an Ant Build Script

Apache Ant is a popular, cross-platform build utility for which JDeveloper offers excellent design-time support. You can incorporate the automatic execution of JUnit tests and test output report generation by using Ant's built-in <junit> and <junitreport> tasks. Example 24–9 shows a task called `tests` from the SRDemo's Ant `build.xml` file in the `BuildAndDeploy` project. It depends on the `build` and `buildTests` targets that Ant will ensure have been executed before running the `tests` target.

The <junit> tag contains a nested <test> tag that identifies the test suite class to execute and specifies a directory in which to report the results. The <junitreport> tag allows you to format the test results into a collection of HTML pages that resemble the format of JavaDoc. To try running the SRDemo's JUnit test from this Ant task, select the `build.xml` file in the Application Navigator, and choose **Run Ant Target > tests** from the context menu.

Example 24–9 Ant Build Target Runs JUnit Test Suite and Generates Report

```
<!-- In SRDemo's build.xml -->
<target name="tests" description="Run the model layer Unit tests"
        depends="build, buildTests">
  <mkdir dir="${tests.reporting.dir}"/>
  <junit printsummary="true" fork="true">
    <formatter usefile="true" type="xml"/>
    <test name="oracle.srdemo.tests.model.SRServiceAllTests"
          todir="${tests.reporting.dir}"/>
    <sysproperty key="jbo.debugoutput" value="file"/>
    <classpath path="${model.build.dir}"/>
    <classpath path="${tests.build.dir}"/>
    <classpath refid="tests.classpath"/>
  </junit>
  <junitreport todir="${tests.reporting.dir}">
    <fileset dir="${tests.reporting.dir}">
      <include name="TEST-*.xml"/>
    </fileset>
    <report format="frames" todir="${tests.reporting.dir}"/>
  </junitreport>
</target>
```

24.8.5 Customizing the Default JUnit Test Classes

The SRDemo's `UnitTest` project includes examples of a few customizations you can make to the generated test case classes. This section describes how to customize the text fixture to authenticate a particular user and how to refactor common test case code in to a base class.

24.8.5.1 Customizing the Test Fixture to Run as an Authenticated User

The `SRServiceFixture` has a custom constructor that accepts a username and a password. Its `setUp()` method uses a custom ADF Business Components `EnvInfoProvider` implementation to supply runtime configuration parameters to the application module. In particular, to authenticate as the given username with the given password, the `JUnitFixureLoginInfoProvider` class implements the `EnvInfoProvider` interface's `getInfo()` method to return:

- The value `Must` for the `jbo.security.enforce` property
- The username value for the `java.naming.security.principal` property (referenced using the `JboContext.SECURITY_PRINCIPAL` constant)
- The password value for the `java.naming.security.credentials` property (referenced using the `JboContext.SECURITY_CREDENTIALS` constant)

Each of the three test cases in the `oracle.srdemo.tests.model.unittests` package creates its instance of the `SRServiceFixture` with a different combination of username and password values. For example, the `SRServiceTestAsManagerRole` test case creates a `SRServiceFixture` for a user that has the manager role (`sking`). The other two tests each do the same, but for a user of the other two roles `Technician` and `Manager`.

Tip: The `UnitTests` project in the SRDemo includes the `ConfigurationData` directory as one of its project contents paths. This ensures that the `jazn-data.xml` file (contained in its `META-INF` subdirectory) which defines all the users and roles for the demo, is readable by the JUnit tests at runtime. XML files in the project source path are copied to the project's output directory during compilation since their `*.xml` file extension is included by default in the **Copy File Types to Output Directory** field of the **Compiler** page in the Project Properties dialog.

24.8.5.2 Refactoring Common Test Case Code Into a Base Class

All three of the test case classes extend the `SRUnitTestBase` class. This class contains code that is common to all of the test cases. For example, it includes a number of helper methods for getting the current date, setting the current row in a view object by a stringified key, and creating a key of appropriate type for a given view object. It also defines the abstract method `createSRServiceFixtureForTest()` that each subclass overrides to return the `SRServiceFixture` constructed with the appropriate username and password for the authentication.

Part IV

Advanced Topics

Part IV contains the following chapters:

- Chapter 25, "Advanced Business Components Techniques"
- Chapter 26, "Advanced Entity Object Techniques"
- Chapter 27, "Advanced View Object Techniques"
- Chapter 28, "Application Module State Management"
- Chapter 29, "Understanding Application Module Pooling"
- Chapter 30, "Adding Security to an Application"
- Chapter 31, "Creating Data Control Adapters"
- Chapter 32, "Working Productively in Teams"
- Chapter 33, "Working with Web Services"
- Chapter 34, "Deploying ADF Applications"

25 Advanced Business Components Techniques

This chapter describes advanced techniques that apply to all types of ADF Business Components.

This chapter includes the following sections:

- Section 25.1, "Globally Extending ADF Business Components Functionality"
- Section 25.2, "Creating a Layer of Framework Extensions"
- Section 25.3, "Customizing Framework Behavior with Extension Classes"
- Section 25.4, "Creating Generic Extension Interfaces"
- Section 25.5, "Invoking Stored Procedures and Functions"
- Section 25.6, "Accessing the Current Database Transaction"
- Section 25.7, "Working with Libraries of Reusable Business Components"
- Section 25.8, "Customizing Business Components Error Messages"
- Section 25.9, "Creating Extended Components Using Inheritance"
- Section 25.10, "Substituting Extended Components In a Delivered Application"

Note: To experiment with a working version of the examples in this chapter, download the `AdvancedExamples` workspace from the *Example Downloads* page at `http://otn.oracle.com/documentation/jdev/b25947_01`.

25.1 Globally Extending ADF Business Components Functionality

One of the powerful features of framework-based development is the ability to extend the base framework to change a built-in feature to behave differently or to add a new feature that can be used by all of your applications. This section describes:

- What framework extension classes are
- How to create an extension class and base ADF components you create on it
- How to adopt the best practice of using a whole custom layer of framework extension classes for your component or specific project

25.1.1 What Are ADF Business Components Framework Extension Classes?

An ADF Business Components framework extension class is Java class you write that extends one of the framework's base classes to:

- Augment a built-in feature works with additional, generic functionality
- Change how a built-in feature works, or even to
- Workaround a bug you encounter in a generic way

Once you've created a framework extension class, any new ADF components you create can be based on your *customized* framework class instead of the base one. Of course, you can also update the definitions of existing components to use the new framework extension class as well.

25.1.2 How To Create a Framework Extension Class

To create a framework extension class, follow these steps:

1. Identify a project to contain the framework extension class.

   You can create it in the same project as your business service components if you believe it will only be used by components in that project. Alternatively, if you believe you might like to reuse the framework extension class across multiple ADF applications, create a separate `FrameworkExtensions` project (as shown in the SRDemo application) to contain the framework extension classes.

2. Ensure the **BC4J Runtime** library is in the project's libraries list.

   Use the **Libraries** page of the **Project Properties** dialog to verify this and to add the library if missing.

3. Create the new class using the **Create Java Class** dialog.

   This dialog is available in the **New Gallery** in the **General** category.

4. Specify the appropriate framework base class from the `oracle.jbo.server` package in the **Extends** field.

   Figure 25–1 illustrates what it would look like to create a custom framework extension class named `CustomAppModuleImpl` in the `com.yourcompany.fwkext` package to customize the functionality of the base application module component. To quickly find the base class you're looking for, use the **Browse** button next to the **Extends** field that launches the JDeveloper **Class Browser**. Using its **Search** tab, you can type in part of the class name (including using `*` as a wildcard) to quickly subset the list of classes to find the one you're looking for.

Figure 25–1 Creating a Framework Extension Class for an Application Module

When you click **OK**, JDeveloper creates the custom framework extension class for you in the directory of the project's source path corresponding to the package name you've chosen.

Note: Some ADF Business Components component classes exist in both a server-side and a remote-client version. For example, if you use the JDeveloper **Class Browser** and type `ApplicationModuleImpl` into the **Match Class Name** field on the **Search** tab, the list will show two ApplicationModuleImpl classes: one in the `oracle.jbo.server` package and the other in the `oracle.jbo.client.remote` package. When creating framework extension classes, use the base ADF classes in the `oracle.jbo.server` package.

25.1.3 What Happens When You Create a Framework Extension Class

After creating a new framework extension class, it will not automatically be used by your application. You must decide which components in your project should make use of it. The following sections describe the available approaches for basing your ADF components on your own framework extension classes.

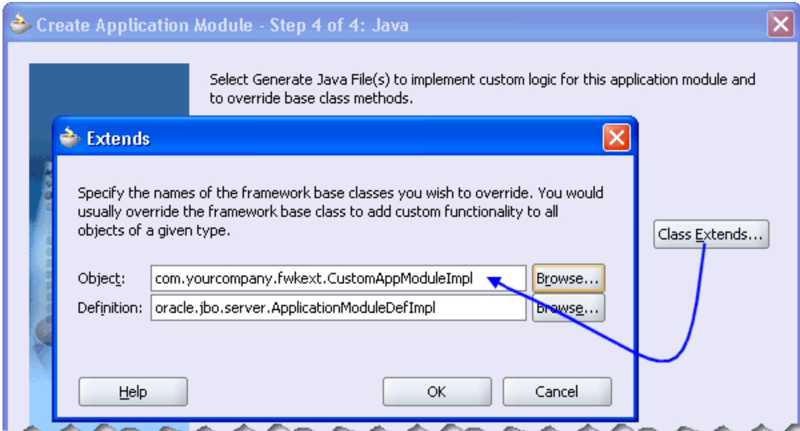
25.1.4 How to Base an ADF Component on a Framework Extension Class

You can set the base classes for any ADF component using the **Java** page of any ADF Business Components wizard or editor. Before doing so, review the following checklist:

- If you have decided to create your framework extension classes in a separate project, ensure that you have visited the **Dependencies** page of the **Project Properties** dialog for the project containing your business components in order to mark the `FrameworkExtension` project as a project dependency.
- If you have packaged your framework extension classes in a Java archive (JAR) file, ensure that you have created a named library definition to reference its JAR file and also listed that library in the library list of the project containing your business components. To create a library if missing, use the **Manage Libraries** dialog available from the **Tools | Manage Libraries** main menu item. To verify or adjust the project's library list, use the **Libraries** page of the **Project Properties** dialog.

After you ensure the framework classes are available to reference, Figure 25–2 shows how you would use the `CustomAppModuleImpl` class as the base class for a new application module. By clicking the **Class Extends** button of the **Java** page of the wizard, the **Extends** dialog displays to let you enter the fully-qualified name of your framework extension class (or use the **Browse** button to use the JDeveloper **Class Browser** to find it quickly).

Figure 25–2 Specifying a Custom Base Class for a New Application Module



The same **Class Extends** button appears on the **Java** page of every ADF Business Components wizard and editor, so you can use this technique to choose your desired framework extension base class(es) both for new components or existing ones.

Note: When using the JDeveloper **Class Browser** in the **Extends** dialog of an ADF Business Components wizard or editor to select a custom base class for the component, the list of available classes is automatically filtered to show only classes that are appropriate. For example, when clicking **Browse** in Figure 25–2 to select an application module **Object** base class, the list will only show classes available in the current project's library list which extend the `oracle.jbo.server.ApplicationModule` class either directly or indirectly. If you don't see the class you're looking for, either you extended the incorrect base class or you have chosen the wrong component class name to override.

25.1.5 What Happens When You Base a Component on a Framework Extension Class

When an ADF component you create extends a custom framework extension class, JDeveloper updates its XML component definition to reflect the custom class name you've chosen.

25.1.5.1 Basing an XML-Only Component on a Framework Extension Class

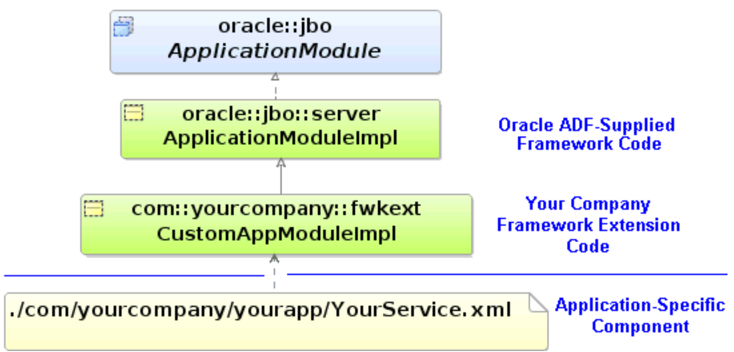
For example, assume you've created the `YourService` application module above in the `com.yourcompany.yourapp` package, with a custom application module base class of `CustomAppModuleImpl`. If you have opted to leave the component as an XML-only component with no custom Java file, its XML component definition (YourService.xml) will look like what you see in Example 25–1. The value of the `ComponentClass` attribute of the <AppModule> tag is read at runtime to identify the Java class to use to represent the component.

Example 25–1 Custom Base Class Names Are Recorded in XML Component Definition

```
<AppModule
   Name="YourService"
   ComponentClass="com.yourcompany.fwkext.CustomAppModuleImpl" >
  <!-- etc. -->
</AppModule>
```

Figure 25–3 illustrates how the XML-only `YourService` application module relates to your custom extension class. At runtime, it uses the `CustomAppModuleImpl` class which inherits its base behavior from the `ApplicationModuleImpl` class.

Figure 25–3 XML-Only Component Reference an Extended Framework Base Class



25.1.5.2 Basing a Component with a Custom Java Class on a Framework Extension Class

If your component requires a custom Java class, as you've seen in previous chapters you open the **Java** page of the component editor and check the appropriate checkbox to enable it. For example, when you enable a custom application module class for the `YourServer` application module, JDeveloper creates the appropriate `YourServiceImpl.java` class. As shown in Example 25–2, it also updates the component's XML component definition to reflect the name of the custom component class.

Example 25–2 Custom Component Class Recorded in XML Component Definition

```
<AppModule
   Name="YourService"
   ComponentClass="com.yourcompany.yourapp.YourServiceImpl" >
  <!-- etc. -->
</AppModule>
```

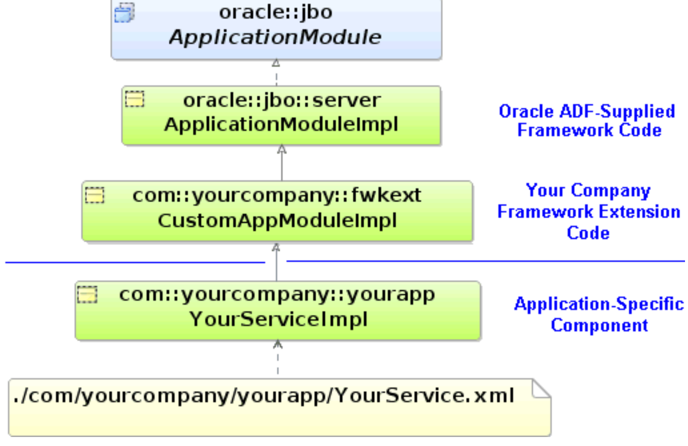
JDeveloper also updates the component's custom Java class to modify its `extends` clause to reflect the new custom framework base class, as shown in Example 25–3.

Example 25–3 Component's Custom Java Class Updates to Reflect New Base Class

```
package com.yourcompany.yourapp;
import com.yourcompany.fwkext.CustomAppModuleImpl;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
public class YourServiceImpl extends CustomAppModuleImpl {
  /**This is the default constructor (do not remove)   */
  public YourServiceImpl() {}
  // etc.
}
```

Figure 25–4 illustrates how the `YourService` application module with its custom `YourServiceImpl` class is related to your framework extension class. At runtime, it uses the `YourServiceImpl` class which inherits its base behavior from the `CustomAppModuleImpl` framework extension class which, in turn, extends the base `ApplicationModuleImpl` class.

Figure 25–4 Component with Custom Java Extending Customized Framework Base Class



25.1.6 What You May Need to Know

25.1.6.1 Don't Update the Extends Clause in Custom Component Java Files By Hand

If you have an ADF component with a custom Java class and later decide to base the component on a framework extension class, use the **Class Extends** button on the **Java** page of the component editor to change the component's base class. Doing this updates the component's XML component definition to reflect the new base class, and *also* modifies the `extends` clause in the component's custom Java class. If you manually update the `extends` clause without using the component editor, the component's XML component definition will not reflect the new inheritance and the next time you open the editor, your manually modified `extends` clause will be overwritten with what the component editor believes is the correct component base class.

25.1.6.2 You Can Have Multiple Levels of Framework Extension Classes

In the examples above, you've seen a single `CustomAppModuleImpl` class that extends the base `ApplicationModuleImpl` class. However, there is no fixed limit on how many levels of framework extension classes you create. After creating a company-level `CustomAppModuleImpl` to use for all application modules in all ADF applications your company creates, some later project team might encounter the need to further customize that framework extension class. That team could create a `SomeProjectCustomAppModuleImpl` class that extends the `CustomAppModuleImpl` and then include the project-specific custom application module code in there:

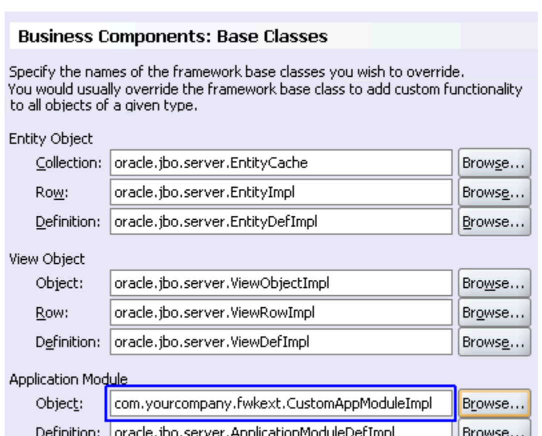
```
public class SomeProjectCustomAppModuleImpl
       extends CustomAppModuleImpl {
  /*
   * Custom application module code specific to the
   * "SomeProject" project goes here.
   */
}
```

Then, any application modules created as part of the implementation of this specific project can use the `SomeProjectCustomAppModuleImpl` as their base class instead of the `CustomAppModuleImpl`.

25.1.6.3 Setting up Project-Level Preferences for Framework Extension Classes

If you decide to use a specific set of framework extension classes as a standard for a given project, you can open the **Business Components > Base Classes** page in the **Project Properties** dialog, as shown in Figure 25–5, to define your preferred base classes for each component type. For example, to indicate that any new application modules created in the project should use the `CustomAppModuleImpl` class by default, enter the fully-qualified name of that class in the **Application Module Object** class name field as shown. Setting these preferences for base classes does not affect any existing components in the project, but the component wizards will use the preferences for any new components created.

Figure 25–5 Setting Project-Level Preferences for ADF Component Base Classes



25.1.6.4 Setting Up Framework Extension Class Preferences at the IDE Level

By choosing the **Tools | Preferences** main menu item in JDeveloper, you can open the **Business Components > Base Classes** page to set framework base classes preferences at a global level. This setting does not affect any existing projects containing ADF components, but JDeveloper will use these as the default values for the project-level business components base classes preferences for any new projects created.

25.2 Creating a Layer of Framework Extensions

Before you begin to develop application-specific business components, Oracle recommends that you consider creating a complete layer of framework extension classes and setting up your project-level preferences to use that layer by default. You might not have any custom code in mind to put in some (or any!) of these framework extension classes yet, but the first time you encounter a need to:

- Add a generic feature that all your company's application modules require
- Augment a built-in feature with some custom, generic processing
- Workaround a bug you encounter in a generic way

You will be glad you heeded this recommendation. Failure to set up these preferences at the outset can present your team with a substantial inconvenience if you discover mid-project that all of your entity objects, for example, require a new generic feature, augmented built-in feature, or a generic bug workaround. Putting a complete layer of framework classes in place to be automatically used by JDeveloper at the start of your project is an insurance policy against this inconvenience and the wasted time related to dealing with it later in the project.

25.2.1 How to Create Your Layer of Framework Extension Layer Classes

A common set of customized framework base classes in a package name of your own choosing like *`com.yourcompany.adfextensions`*, each importing the `oracle.jbo.server.*` package, would consist of the following classes:

- `public class CustomEntityImpl extends EntityImpl`
- `public class CustomEntityDefImpl extends EntityDefImpl`
- `public class CustomViewObjectImpl extends ViewObjectImpl`
- `public class CustomViewRowImpl extends ViewRowImpl`
- `public class CustomApplicationModuleImpl extends ApplicationModuleImpl`
- `public class CustomDBTransactionImpl extends DBTransactionImpl2`
- `public class CustomDatabaseTransactionFactory extends DatabaseTransactionFactory`

To make your framework extension layer classes easier to package as a reusable library, Oracle recommends creating them in a separate project from the projects that use them. In the SRDemo application, the `FrameworkExtensions` project contains all of the framework extension classes used by the application.

> **Note:** For your convenience, the `FrameworkExtensions` project in the `AdvancedExamples` workspace contains a set of these classes. You can select the `com.yourcompany.adfextensions` package in the Application Navigator and choose the **Refactor > Rename** option from the context menu to change the package name of all the classes to a name you prefer.

For completeness, you may also want to create customized framework classes for the following classes as well, note however that overriding anything in these classes would be a fairly rare requirement.

- `public class CustomViewDefImpl extends ViewDefImpl`
- `public class CustomEntityCache extends EntityCache`
- `public class CustomApplicationModuleDefImpl extends ApplicationModuleDefImpl`

25.2.2 How to Package Your Framework Extension Layer in a JAR File

Use the **Create Deployment Profile: JAR File** dialog to create a JAR file containing the classes in your framework extension layer. This is available in the **New Gallery** in the **General > Deployment Files** category.

> **Note:** If you don't see the **Deployment Profiles** category in the **New Gallery**, set the **Filter By** dropdown list at the top of the dialog to the **All Technologies** choice to make it visible.

Give the deployment profile a name like `FrameworkExtensions` and click **OK**. By default the JAR file will include all class files in the project. Since this is exactly what you want, when the **JAR Deployment Profile Properties** dialog appears, you can just click **OK** to finish.

Finally, to create the JAR file, select the `FrameworkExtensions.deploy` node in the Application Navigator under the **Resources** folder, and choose **Deploy to JAR File** on the context menu. A **Deployment** tab appears in the JDeveloper **Log window** that should display feedback like:

```
----  Deployment started.  ----    Apr 28, 2006 1:42:39 PM
Running dependency analysis...
Wrote JAR file to ...\FrameworkExtensions\deploy\FrameworkExtensions.jar
Elapsed time for deployment:  less than one second
----  Deployment finished.  ----    Apr 28, 2006 1:42:39 PM
```

25.2.3 How to Create a Library Definition for Your Framework Extension JAR File

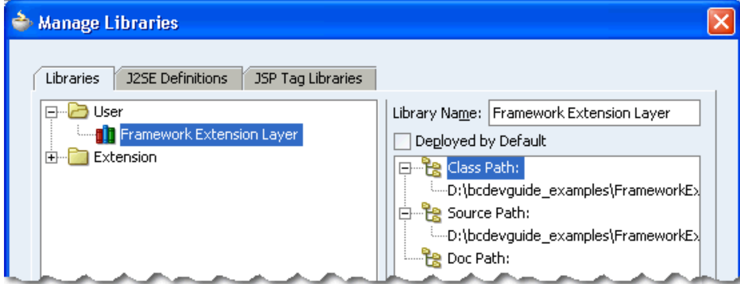
JDeveloper uses named libraries as a convenient way to organize the one or more JAR files that comprise reusable component libraries. To define a library for your framework extensions JAR file, do the following:

1. Choose **Tools > Manage Libraries** from the JDeveloper main menu.
2. In the **Manage Libraries** dialog, select the **Libraries** tab.
3. Select the **User** folder in the tree and click the **New** button.

4. In the **Create Library** dialog that appears, name the library "Framework Extension Layer" and select the **Class Path** node and click **Add Entry**.
5. Use the **Select Path Entry** dialog that appears to select the `FrameworkExtensions.jar` file that contains the class files for the framework extension components, then click **Select**.
6. Select the **Source Path** node and click **Add Entry.**
7. Use the **Select Path Entry** dialog that appears to select the `..\FrameworkExtensions\src` directory where the source files for the framework extension classes reside, then click **Select**.
8. Click **OK** to dismiss the **Create Library** dialog and define the new library.

When finished, you will see your new "Framework Extension Layer" user-defined library, as shown in Figure 25–6. You can then add this library to the library list of any project where you will be building business services, and your custom framework extension classes will be available to reference as the preferred component base classes.

Figure 25–6 New User-Defined Library for Your Framework Extensions Layer



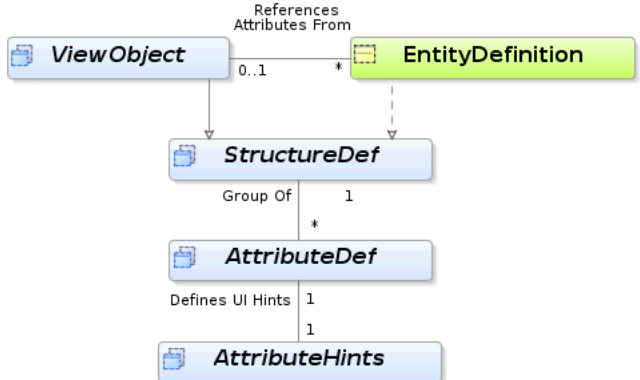
25.3 Customizing Framework Behavior with Extension Classes

One of the common tasks you'll perform in your framework extension classes is implementing custom application functionality. Since framework extension code is written to be used by all components of a specific type, the code you write in these classes often needs to work with component attributes in a generic way. To address this need, ADF provides API's that allow you to access component metadata at runtime. It also provides the ability to associate custom metadata properties with any component or attribute. You can write your generic framework extension code to leverage runtime metadata and custom properties to build generic functionality, which if necessary, only is used in the presence of certain custom properties.

25.3.1 How to Access Runtime Metadata For View Objects and Entity Objects

Figure 25–7 illustrates the three primary interfaces ADF provides for accessing runtime metadata about view objects and entity objects. The `ViewObject` interface extends the `StructureDef` interface. The class representing the entity definition (`EntityDefImpl`) also implements this interface. As its name implies, the `StructureDef` defines the structure and the component and provides access to a collection of `AttributeDef` objects that offer runtime metadata about each attribute in the view object row or entity row. Using an `AttributeDef`, you can access its companion `AttributeHints` object to reference hints like the display label, format mask, tooltip, etc.

Figure 25–7 Runtime Metadata Available for View Objects and Entity Objects



25.3.2 Implementing Generic Functionality Using Runtime Metadata

In Section 7.9.3, "What You May Need to Know About Enabling View Object Key Management for Read-Only View Objects" you learned that for read-only view objects the `findByKey()` method and the `setCurrentRowWithKey` built-in operation only work if you override the `create()` method on the view object to call `setManageRowsByKey(true)`. This can be a tedious detail to remember if you create a lot of read-only view objects, so it is a great candidate for automating in a framework extension class for view objects.

The SRDemo application contains an `SRViewObjectImpl` class in the `FrameworkExtensions` project that is the base class for all view objects in the application. That framework extension class for view objects extends the base `ViewObjectImpl` class and overrides the `create()` method as shown in Example 25–4 to automate this task. After calling the `super.create()` to perform the default framework functionality when a view object instance is created at runtime, the code tests whether the view object is a read-only view object with at least one attribute marked as a key attribute. If this is the case, it invokes `setManageRowsByKey(true)`.

The `isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute()` helper method determines whether the view object is read-only by testing the combination of the following conditions:

- `isFullSql()` is `true`

  This method returns true if the view object's SQL query is completely specified by the developer, as opposed to having the select list derived automatically based on the participating entity usages.

- `getEntityDefs()` is `null`

  This method returns an array of `EntityDefImpl` objects representing the view object's entity usages. If it returns `null`, then the view object has no entity usages.

It goes on to determine whether the view object has any key attributes by looping over the `AttributeDef` array returned by the `getAttributeDefs()` method. If the `isPrimaryKey()` method returns true for any attribute definition in the list, then you know the view object has a key.

Example 25–4 Automating Setting Manage Rows By Key

```
public class SRViewObjectImpl extends ViewObjectImpl {
  protected void create() {
    super.create();
    if (isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute()) {
      setManageRowsByKey(true);
    }
  }
  boolean isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute() {
    if (getViewDef().isFullSql() && getEntityDefs() == null) {
      for (AttributeDef attrDef : getAttributeDefs()) {
        if (attrDef.isPrimaryKey()) {
          return true;
        }
      }
    }
    return false;
  }
  // etc.
}
```

25.3.3 Implementing Generic Functionality Driven by Custom Properties

In JDeveloper, when you create application modules, view objects, and entity objects you can open the **Custom Properties** tab of the editor to define custom metadata properties for any component. These are name/value pairs that you can use to communicate additional declarative information about the component to the generic code that you write in framework extension classes. You can use the `getProperty()` method in your code to conditionalize generic functionality based on the presence of, or the specific value of, one of these custom metadata properties.

For example, the `SRViewObjectImpl` framework extension class in the SRDemo application overrides the view object's `insertRow()` method to conditionally force a row to be inserted and to appear as the last row in the row set. If any view object extending this framework extension class defines a custom metadata property named `InsertNewRowsAtEnd`, then this generic code executes to insert new rows at the end. If a view object does not define this property, it will have the default `insertRow()` behavior. In the SRDemo application, the `ServiceHistories` view object defines this custom metadata property so any new rows added to it get inserted at the bottom.

Example 25–5 Conditionally Inserting New Rows at the End of a View Object's Default RowSet

```
public class SRViewObjectImpl extends ViewObjectImpl {
  private static final String INSERT_NEW_ROWS_AT_END = "InsertNewRowsAtEnd";
  public void insertRow(Row row) {
    super.insertRow(row);
    if (getProperty(INSERT_NEW_ROWS_AT_END) != null) {
      row.removeAndRetain();
      last();
      next();
      getDefaultRowSet().insertRow(row);
    }
  }
  // etc.
}
```

In addition to defining component-level custom properties, you can also define properties on view object attributes, entity object attributes, and domains. At runtime,

you access them using the getProperty() method on the AttributeDef interface for a given attribute.

25.3.4 What You May Need to Know

25.3.4.1 Determining the Attribute Kind at Runtime

In addition to providing information about an attribute's name, Java type, SQL type, and many other useful pieces of information, the AttributeDef interface contains the getAttributeKind() method that you can use to determine the kind of attribute it represents. This method returns a byte value corresponding to one of the public constants in the AttributeDef interface listed in Table 25–1.

Table 25–1 Entity Object and View Object Attribute Kinds

Public AttributeDef Constant	Attribute Kind Description
ATTR_PERSISTENT	Persistent attribute
ATTR_TRANSIENT	Transient attribute
ATTR_ENTITY_DERIVED	View object attribute mapped to an entity-level transient attribute
ATTR_SQL_DERIVED	SQL-Calculated attribute
ATTR_DYNAMIC	Dynamic attribute
ATTR_ASSOCIATED_ROWITERATOR	Accessor attribute returning a RowSet of set of zero or more Rows
ATTR_ASSOCIATED_ROW	Accessor attribute returning a single Row

25.3.4.2 Configuring Design Time Custom Property Names

Once you have written framework extension classes that depend on custom properties, you can set a JDeveloper preference so that your custom property names show in the dropdown list on the **Custom Properties** tab of the appropriate component editor. To set up these pre-defined custom property names, choose **Tools | Preferences** from the JDeveloper main menu and open the **Business Components > Property Names** tab in the **Preferences** dialog.

25.3.4.3 Setting Custom Properties at Runtime

You may find it handy to programmatically set custom property values at runtime. While the setProperty() API to perform this function is by design not available to clients on the ViewObject, ApplicationModule, or AttributeDef interfaces in the oracle.jbo package, code you write *inside* your ADF components' custom Java classes can use it.

> **Note:** You can experiment with an example of this technique by using the ProgrammaticallySetProperties project in the AdvancedExamples workspace. See the note at the beginning of this chapter for download instructions.

25.4 Creating Generic Extension Interfaces

In addition to creating framework extension classes, you can create custom interfaces that all of your components can implement by default. This section considers an example for an application module, however, the same functionality is possible for a custom extended view object and view row interface as well.

Note: The examples in this section refer to the `CustomizedExtensionInterface` project in the `AdvancedExamples` workspace. See the note at the beginning of this chapter for download instructions.

Assume that you have a `CustomApplicationModuleImpl` class that extends `ApplicationModuleImpl` and that you want to expose two custom methods like this:

```
public void doFeatureOne(String arg);
public int anotherFeature(String arg);
```

Perform the following steps to create a custom extension interface `CustomApplicationModule` and have your `CustomApplicationModuleImpl` class implement it.

1. Create a custom interface that contains the methods you would like to expose globally on your application module components. For this scenario, that interface would look like this:

   ```
   package devguide.advanced.customintf.fwkext;
   /**
    * NOTE: This does not extend the
    * ====  oracle.jbo.ApplicationModule interface.
    */
   public interface CustomApplicationModule  {
     public void doFeatureOne(String arg);
     public int anotherFeature(String arg);
   }
   ```

   Notice that the interface does *not* extend the `oracle.jbo.ApplicationModule` interface.

2. Modify your `CustomApplicationModuleImpl` application module framework extension class to implement this new `CustomApplicationModule` interface.

   ```
   package devguide.advanced.customintf.fwkext;
   import oracle.jbo.server.ApplicationModuleImpl;
   public class CustomApplicationModuleImpl
          extends ApplicationModuleImpl
          implements CustomApplicationModule {
     public void doFeatureOne(String arg) {
       System.out.println(arg);
     }
     public int anotherFeature(String arg) {
       return arg == null ? 0 : arg.length();
     }
   }
   ```

3. Rebuild your project.

The ADF wizards will only "see" your interfaces after they have been successfully compiled.

Then, to create a new `ProductModule` application module which exposes the global extension interface `CustomApplicationModule` and is based on the `CustomApplicationModuleImpl` framework extension class, do the following:

1. Ensure that your application module has a custom Java class.

   If it doesn't, you can enable one on the **Java** page of the Application Module Editor and clicking the **Apply** button.

2. Select `CustomApplicationModuleImpl` as the base class for the application module.

   To do this, again use the **Java** tab of the Application Module Editor and open the **Extends** dialog by clicking the **Class Extends** button.

3. Indicate that you want the `CustomApplicationModule` interface to be one of the custom interfaces that clients can use with your component.

   To do this, open the **Client Interface** page. Click on the **Interfaces** button. Shuttle the `CustomApplicationModule` interface from the **Available** to the **Selected** list, then click **OK**.

4. Insure that at least one method appears in the **Selected** list on the **Client Interface** page.

Note: You need to select at least one method in the **Selected** list on the **Client Interface** page, even if it means redundantly selecting one of the methods on the global extension interface. Any method will do in order to get JDeveloper to generate the custom `ProductModule` interface.

When you dismiss the Application Module editor, JDeveloper generates the application module custom interface `ProductModule`. It automatically extends *both* the base `ApplicationModule` interface and your `CustomApplicationModule` extension interface like this:

```
package devguide.advanced.customintf.common;
import devguide.advanced.customintf.fwkext.CustomApplicationModule;

import oracle.jbo.ApplicationModule;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---------------------------------------------------------------------
public interface ProductModule
       extends CustomApplicationModule, ApplicationModule {
  void doSomethingProductRelated();
}
```

Once you've done this, then client code can cast your `ProductModule` application module to a `CustomApplicationModule` interface and invoke the generic extension methods it contains in a strongly-typed way.

Note: The basic steps are the same for exposing methods on a `ViewObjectImpl` framework extension class, as well as for a `ViewRowImpl` extension class.

25.5 Invoking Stored Procedures and Functions

You can write code in the custom Java classes for your business components to invoke database stored procedures and functions. Here you'll consider some simple examples based on procedures and functions in a PL/SQL package; however, using the same techniques, you also can invoke procedures and functions that are not part of a package.

Consider the following PL/SQL package:

```
create or replace package devguidepkg as
  procedure proc_with_no_args;
  procedure proc_with_three_args(n number, d date, v varchar2);
  function  func_with_three_args(n number, d date, v varchar2) return varchar2;
  procedure proc_with_out_args(n number, d out date, v in out varchar2);
end devguidepkg;
```

The following sections explain how to invoke each of the example procedures and functions in this package.

Note: The examples in this section refer to the `StoredProcedureInvocation` project in the `AdvancedExamples` workspace. See the note at the beginning of this chapter for download instructions.

25.5.1 Invoking Stored Procedures with No Arguments

If you need to invoke a stored procedure that takes no arguments, you can use the `executeCommand()` method on the `DBTransaction` interface (in the `oracle.jbo.server` package as shown in Example 25–6.

Example 25–6 Executing a Stored Procedure with No Arguments

```
// In StoredProcTestModuleImpl.java
public void callProcWithNoArgs() {
  getDBTransaction().executeCommand(
    "begin devguidepkg.proc_with_no_args; end;");
}
```

25.5.2 Invoking Stored Procedure with Only IN Arguments

Invoking stored procedures that accept only `IN`-mode arguments — which is the default PL/SQL parameter mode if not specified — requires using a JDBC `PreparedStatement` object. The `DBTransaction` interface provides a `createPreparedStatement()` method to create this object for you in the context of the current database connection. You could use a helper method like the one shown in Example 25–7 to simplify the job of invoking a stored procedure of this kind using a `PreparedStatement`. Importantly, by using a helper method, you can encapsulate the code that closes the JDBC `PreparedStatement` after executing it. The code performs the following basic tasks:

1. Creates a JDBC `PreparedStatement` for the statement passed in, wrapping it in a PL/SQL `begin...end` block.
2. Loops over values for the bind variables passed in, if any.

3. Sets the value of each bind variable in the statement.

   Notice that since JDBC bind variable API's use one-based numbering, the code adds one to the zero-based for loop index variable to account for this.

4. Executes the statement.

5. Closes the statement.

Example 25–7 Helper Method to Simplify Invoking Stored Procedures with Only IN Arguments

```
protected void callStoredProcedure(String stmt, Object[] bindVars) {
  PreparedStatement st = null;
  try {
    // 1. Create a JDBC PreparedStatement for
    st = getDBTransaction().createPreparedStatement("begin "+stmt+";end;",0);
    if (bindVars != null) {
      // 2. Loop over values for the bind variables passed in, if any
      for (int z = 0; z < bindVars.length; z++) {
        // 3. Set the value of each bind variable in the statement
        st.setObject(z + 1, bindVars[z]);
      }
    }
    // 4. Execute the statement
    st.executeUpdate();
  }
  catch (SQLException e) {
    throw new JboException(e);
  }
  finally {
    if (st != null) {
      try {
        // 5. Close the statement
        st.close();
      }
      catch (SQLException e) {}
    }
  }
}
```

With a helper method like this in place, calling the `proc_with_three_args` procedure above would look like this:

```
// In StoredProcTestModuleImpl.java
public void callProcWithThreeArgs(Number n, Date d, String v) {
  callStoredProcedure("devguidepkg.proc_with_three_args(?,?,?)",
                      new Object[]{n,d,v});
}
```

Notice the question marks used as JDBC bind variable placeholders for the arguments passed to the function. JDBC also supports using named bind variables, but using these simpler positional bind variables is also fine since the helper method is just setting the bind variable values positionally.

25.5.3 Invoking Stored Function with Only IN Arguments

Invoking stored functions that accept only `IN`-mode arguments requires using a JDBC `CallableStatement` object in order to access the value of the function result after executing the statement. The `DBTransaction` interface provides a `createCallableStatement()` method to create this object for you in the context of the current database connection. You could use a helper method like the one shown in Example 25–8 to simplify the job of invoking a stored function of this kind using a `CallableStatement`. As above, the helper method encapsulates both the creation and clean up of the JDBC statement being used.

The code performs the following basic tasks:

1. Creates a JDBC `CallableStatement` for the statement passed in, wrapping it in a PL/SQL `begin...end` block.
2. Registers the first bind variable for the function return value.
3. Loops over values for the bind variables passed in, if any.
4. Sets the value of each bind user-supplied bind variable in the statement.

   Notice that since JDBC bind variable API's use one-based numbering, and since the function return value is already the first bind variable in the statement, the code adds *two* to the zero-based for loop index variable to account for these.
5. Executes the statement.
6. Returns the value of the first bind variable.
7. Closes the statement.

Example 25–8 Helper Method to Simplify Invoking Stored Functions with Only IN Arguments

```
// Some constants
public static int NUMBER = Types.NUMERIC;
public static int DATE = Types.DATE;
public static int VARCHAR2 = Types.VARCHAR;

protected Object callStoredFunction(int sqlReturnType, String stmt,
                                    Object[] bindVars) {
  CallableStatement st = null;
  try {
    // 1. Create a JDBC CallabledStatement
    st = getDBTransaction().createCallableStatement(
           "begin ? := "+stmt+";end;",0);
    // 2. Register the first bind variable for the return value
    st.registerOutParameter(1, sqlReturnType);
    if (bindVars != null) {
      // 3. Loop over values for the bind variables passed in, if any
      for (int z = 0; z < bindVars.length; z++) {
        // 4. Set the value of user-supplied bind vars in the stmt
        st.setObject(z + 2, bindVars[z]);
      }
    }
    // 5. Set the value of user-supplied bind vars in the stmt
    st.executeUpdate();
    // 6. Return the value of the first bind variable
    return st.getObject(1);
  }
```

```
  catch (SQLException e) {
    throw new JboException(e);
  }
  finally {
    if (st != null) {
      try {
        // 7. Close the statement
        st.close();
      }
      catch (SQLException e) {}
    }
  }
}
```

With a helper method like this in place, calling the `func_with_three_args` procedure above would look like this:

```
// In StoredProcTestModuleImpl.java
public String callFuncWithThreeArgs(Number n, Date d, String v) {
  return (String)callStoredFunction(VARCHAR2,
                            "devguidepkg.func_with_three_args(?,?,?)",
                            new Object[]{n,d,v});
}
```

Notice the question marks as above that are used as JDBC bind variable placeholders for the arguments passed to the function. JDBC also supports using named bind variables, but using these simpler positional bind variables is also fine since the helper method is just setting the bind variable values positionally.

25.5.4 Calling Other Types of Stored Procedures

Calling a stored procedure or function like `devguidepkg.proc_with_out_args` that includes arguments of `OUT` or `IN OUT` mode requires using a `CallableStatement` as in the previous section, but is a little more challenging to generalize into a helper method. Example 25–9 illustrates the JDBC code necessary to invoke the `devguidepkg.proc_with_out_args` procedure.

The code performs the following basic tasks:

1. Defines a PL/SQL block for the statement to invoke.
2. Creates the `CallableStatement` for the PL/SQL block.
3. Registers the positions and types of the `OUT` parameters.
4. Sets the bind values of the `IN` parameters.
5. Executes the statement.
6. Creates a JavaBean to hold the multiple return values

   The `DateAndStringBean` class contains bean properties named `dateVal` and `stringVal`.
7. Sets the value of its `dateVal` property using the first `OUT` param.
8. Sets value of its `stringVal` property using second `OUT` param.
9. Returns the result.
10. Closes the JDBC `CallableStatement`.

Example 25–9 Calling a Stored Procedure with Multiple OUT Arguments

```
public Date callProcWithOutArgs(Number n, String v) {
  CallableStatement st = null;
  try  {
    // 1. Define the PL/SQL block for the statement to invoke
    String stmt = "begin devguidepkg.proc_with_out_args(?,?,?); end;";
    // 2. Create the CallableStatement for the PL/SQL block
    st = getDBTransaction().createCallableStatement(stmt,0);
    // 3. Register the positions and types of the OUT parameters
    st.registerOutParameter(2,Types.DATE);
    st.registerOutParameter(3,Types.VARCHAR);
    // 4. Set the bind values of the IN parameters
    st.setObject(1,n);
    st.setObject(3,v);
    // 5. Execute the statement
    st.executeUpdate();
    // 6. Create a bean to hold the multiple return values
    DateAndStringBean result = new DateAndStringBean();
    // 7. Set value of dateValue property using first OUT param
    result.setDateVal(new Date(st.getDate(2)));
    // 8. Set value of stringValue property using 2nd OUT param
    result.setStringVal(st.getString(3));
    // 9. Return the result
    return result;
  } catch (SQLException e)  {
    throw new JboException(e);
  } finally  {
    if (st != null) {
      try {
        // 10. Close the JDBC CallableStatement
        st.close();
      }
      catch (SQLException e) {}
    }
  }
}
```

The `DateAndString` bean used in Example 25–9 is a simple JavaBean with two bean properties like this:

```
package devguide.advanced.storedproc;
import java.io.Serializable;
import oracle.jbo.domain.Date;
public class DateAndStringBean implements Serializable {
  Date dateVal;
  String stringVal;
  public void setDateVal(Date dateVal) {this.dateVal=dateVal;}
  public Date getDateVal() {return dateVal;}
  public void setStringVal(String stringVal) {this.stringVal=stringVal;}
  public String getStringVal() {return stringVal;}
}
```

Note: In order to allow the custom method to be a legal candidate for inclusion in an application module's custom service interface (if desired), the bean needs to implement the `java.io.Serializable`. interface. Since this is a "marker" interface, this involves simply adding the `implements Serializable` keywords without needing to code the implementation of any interface methods.

25.6 Accessing the Current Database Transaction

Since the ADF Business Components components abstract all of the lower-level database programming details for you, you typically won't need *direct* access to the JDBC `Connection` object. Unless you use the reserved release mode described in Section 28.3.1, "Supported Release Levels", there is no guarantee at runtime that your application will use the exact same application module instance or JDBC `Connection` instance across different web page requests. Since inadvertently holding a reference to the JDBC Connection object in this type of pooled services environment can cause unpredictable behavior at runtime, by design, the ADF Business Components layer has no direct API to obtain the JDBC `Connection`. This is an intentional attempt to discourage its direct use and inadvertent abuse.

However, on occasion it may come in handy when you're trying to integrate third-party code with ADF Business Components, so you can use a helper method like the one shown in Example 25–10 to access the connection.

Example 25–10 Helper Method to Access the Current JDBC Connection

```
/**
 * Put this method in your XXXXImpl.java class where you need
 * to access the current JDBC connection
 */
private Connection getCurrentConnection() throws SQLException {
 /* Note that we never execute this statement, so no commit really happens */
 PreparedStatement st = getDBTransaction().createPreparedStatement("commit",1);
 Connection conn = st.getConnection();
 st.close();
 return conn;
}
```

Caution: Oracle recommends that you never cache the JDBC connection obtained using the helper method above in your own code anywhere. Instead, call the helper method each time you need it to avoid inadvertently holding a reference to a JDBC Connection that might be used in another request by another user at a later time do to the pooled services nature of the ADF runtime environment.

25.7 Working with Libraries of Reusable Business Components

As with other Java components, you can create a JAR file containing one of more packages of reusable ADF components. Then, in other projects you can import one or more packages of components from this component library to reference those in a new application.

Note: The examples in this section refer to the `ReusableComponents`, `ProjectImportingReusableComponents`, and `OtherProjectWithComponents` projects in the `AdvancedExamples` workspace. See the note at the beginning of this chapter for download instructions.

25.7.1 How To Create a Reusable Library of Business Components

Use the **Create Business Components Archive Profile** dialog to create a JAR file containing the Java classes and XML component definitions that comprise your business components library. This is available in the **New Gallery** in the **General > Deployment Files** category.

Note: If you don't see the **Deployment Profiles** category in the **New Gallery**, set the **Filter By** dropdown list at the top of the dialog to the **All Technologies** choice to make it visible.

Give the deployment profile a name like `ReusableComponents.bcdeploy` and click **OK**. As shown in Figure 25–8, the `ReusableComponents.bcdeploy` business components deployment archive profile contains two nested JAR deployment profiles:

- `ReusableComponentsMiddleTier.deploy`
- `ReusableComponentsCommon.deploy`

These two nested profiles are standard JAR deployment profiles that are pre-configured to bundle:

- All of the business components custom java classes and XML component definitions into a `ReusableComponentsCSMT.jar` archive
- All of the client interfaces, message bundle classes, and custom domains into a `ReusableComponentsCSCommon.jar`

They are partitioned this way in order to simplify deployment of ADF Business Components-based applications. The `*CSMT.jar` is an archive of components designed to be deployed only on the middle tier application server. The `*CSCommon.jar` is common both to the application server and to the remote client tier in the deployment scenario when the client interacting with the application module is running in a different physical server from the application module with which it is working.

Figure 25–8 Business Components Archive Deployment Profile Contains Nested Profiles

To create the JAR files, select the `ReusableComponents.bcdeploy` node in the Application Navigator under the **Resources** folder, and choose **Deploy** on the context menu. A **Deployment** tab appears in the JDeveloper **Log window** that should display feedback like:

```
----  Deployment started.  ----    Apr 28, 2006 7:04:02 PM
Running dependency analysis...
Wrote JAR file to ...\ReuseableComponents\deploy\ReuseableComponentsCSMT.jar
Running dependency analysis...
Wrote JAR file to ...\ReuseableComponents\deploy\ReuseableComponentsCSCommon.jar
Elapsed time for deployment:  less than one second
----  Deployment finished.  ----    Apr 28, 2006 7:04:02 PM
```

25.7.2 How To Import a Package of Reusable Components from a Library

Once you have created a reusable library of business components, you can import one or more packages of components from that library in other projects to reference them. When you import a package of business components from a library, the components in that package are available in the various **Available** lists of the ADF Business Components component wizards and editor, however they do not display in the Application Navigator. The only components that appear in the Application Navigator are the ones in the source path for the current project.

To import a package of business components from a library, do the following:

1. Define a library for your JAR file on the **Libraries** tab of the **Project Properties** dialog of the importing project.

   You can define the library as a project-level library or a user-level library. Be sure to include both the `*CSMT.jar` and the `*CSCommon.jar` in the class path of the library definition.

2. Include the new library in your importing project's library list.

3. With the importing project selected in the Application Navigator, choose File | Import from the JDeveloper main menu.

4. In the **Import** dialog that appears, select **Business Components** from the list.

5. Use the file open dialog to navigate *into* your library's `*CSMT.jar` file — as if it were a directory — and select the XML component definition file from any components in the package whose components you want to import.
6. Acknowledge the alert that confirms the successful importing of the package.
7. Repeat steps 3-6 again for each package of components you want to import.

Assuming that there was an entity object like `Product` in the package(s) of components you imported, you could then create a new view object in the importing project using the imported `Product` component as its entity usage. This is just one example. You can reference any of the imported components as if they were in the source path of your project. The only difference is that you cannot *edit* the imported components. In fact, the reusable component library JAR file might only contain the XML component definition files and the Java `*.class` files for the components without any source code.

25.7.3 What Happens When You Import a Package of Reusable Components from a Library

When you import a package of components into a project named *`YourImportingProjectName`*, JDeveloper adds a reference to that package in the *`YourImportingProjectName.jpx`* file in the root directory of your importing project's source path. As part of this entry, it includes a design time project named `_LocationURL` whose value points to the JAR file in which the imported components reside.

25.7.4 What You May Need to Know

25.7.4.1 Adding Other Directories of Business Components to Project Source Path

The Application Navigator displays all business components in the source path of your project. If you want to add additional business components from a directory that is not currently part of your project's source path, then open the **Project Content** page of the **Project Properties** dialog and add the parent directory for these other components as one of the directories in the **Java Content** list. In contrast to imported packages of components, these additional components added to your project's source path will be fully editable and will appear in the Application Navigator.

25.7.4.2 Have to Close/Reopen to See Changes from a JAR

If you make changes to your imported components and update the JAR file that contains them, you need to close and reopen any importing projects in order to pickup the changes. This does not require exiting out of JDeveloper. You can select your importing project in the Application Navigator, choose **File | Close** from the main menu, and then re-expand the project's nodes to close and reopen the project. When you do this, JDeveloper will reread the components from the updated version of the imported JAR file.

25.7.4.3 How to Remove an Imported Package from a Project

If you mistakenly import a package of components, or wish to remove an imported package of components that you are not using, at this time, JDeveloper provides no interactive way to do this. To unimport a package, you need to follow these steps:

1. Remove the workspace in question from the Application Navigator.
2. Use a text editor to edit the `YourImportingProjectName.jpx` file.
3. Remove the <Containee> element in that file the represents the imported package you want to remove.

   This means removing every line in the file between (and including) the appropriate <Containee> tag for that package and its matching </Containee> tag.
4. Reopen the workspace in JDeveloper.

Caution: Do not remove an imported package if your project still has components that reference it. If you do, JDeveloper will throw exceptions when the project is opened, or your application may have unpredictable behavior. Ensure that there are no references to any of the components in the imported package before manually removing the package entry from the `*.jpx` file.

25.8 Customizing Business Components Error Messages

Note: The examples in this section refer to the `CustomizedErrorMessages` project in the `AdvancedExamples` workspace. See the note at the beginning of this chapter for download instructions.

25.8.1 How to Customize Base ADF Business Components Error Messages

You can customize any of the built-in ADF Business Components error messages by providing an alternative message string for the error code in a custom message bundle. Assume you do not like the built-in error message:

```
JBO-27014: Attribute Name is Product is required
```

If you have requested the Oracle ADF source code from Oracle Worldwide Support, you can look in the `CSMessageBundle.java` file in the `oracle.jbo` package to see that this error message is related to the combination of the following lines in that message bundle file:

```
public class CSMessageBundle extends CheckedListResourceBundle {
  // etc.
  public static final String EXC_VAL_ATTR_MANDATORY            = "27014";
  // etc.
  private static final Object[][] sMessageStrings = {
    // etc.
    {EXC_VAL_ATTR_MANDATORY, "Attribute {2} in {1} is required"},
    // etc.
  }
}
```

The numbered tokens {2} and {1} are error message placeholders. In this example the {1} is replaced at runtime with the name of the entity object and the {2} with the name of the attribute.

To create a custom message bundle file, do the following:

1. Open the **Business Components > Options** page in the **Project Properties** dialog for the project containing your business components.

   Notice the **Custom Message Bundles to use in this Project** list at the bottom of the dialog.

2. Click **New**.

3. Enter a name and package for the custom message bundle in the **Create MessageBundle class** dialog and click **OK**.

   > **Note:** If the fully-qualified name of your custom message bundle file does not appear in the **Custom Message Bundles to use in this Project** list, click the **Remove** button, then click the **Add** button to add the new message bundle file created. When the custom message bundle file is correctly registered, it's fully-qualified class name should appear in the list.

4. Click **OK** to dismiss the **Project Properties** dialog and open the new custom message bundle class in the source editor.

5. Edit the two-dimensional String array in the custom message bundle class to contain any customized messages you'd like to use.

   Example 25–11 illustrates a custom message bundle class that overrides the error message string for the JBO-27014 error considered above.

Example 25–11 Custom ADF Business Components Message Bundle

```
package devguide.advanced.customerrs;
import java.util.ListResourceBundle;
public class CustomMessageBundle extends ListResourceBundle {
  private static final Object[][] sMessageStrings
    = new String[][] {
       {"27014","You must provide a value for {2}"}
     };
  protected Object[][] getContents() {
    return sMessageStrings;
  }
}
```

25.8.2 What Happens When You Customize Base ADF Business Components Error Messages

After adding this message to your custom message bundle file, if you test the application using the Business Components Browser and try to blank out the value of a mandatory attribute, you'll now see your custom error message instead of the default one:

```
JBO-27014: You must provide a value for Name
```

You can add as many messages to the message bundle as you want. Any message whose error code key matches one of the built-in error message codes will be used at runtime instead of the default one in the `oracle.jbo.CSMessageBundle` message bundle.

25.8.3 How to Customize Error Messages for Database Constraint Violations

If you enforce constraints in the database, you might want to provide a custom error message in your ADF application to display to the end user when one of those constraints is violated. For example, imagine that you added a constraint called `NAME_CANNOT_BEGIN_WITH_X` to the SRDemo application's `PRODUCTS` table using the following DDL statement:

```
alter table products add (
  constraint name_cannot_begin_with_x
      check (upper(substr(name,1,1)) != 'X')
);
```

To define a custom error message in your application, just add a message to a custom message bundle with the constraint name as the message key. For example, assuming that you use the same `CustomMessageBundle.java` class created in the previous section, Example 25–12 shows what it would look like to define a message with the key `NAME_CANNOT_BEGIN_WITH_X` which matches the name of the database constraint name defined above.

Example 25–12 Customizing Error Message for Database Constraint Violation

```
package devguide.advanced.customerrs;
import java.util.ListResourceBundle;
public class CustomMessageBundle extends ListResourceBundle {
  private static final Object[][] sMessageStrings
    = new String[][] {
       {"27014","You must provide a value for {2}"},
       {"NAME_CANNOT_BEGIN_WITH_X",
        "The name cannot begin with the letter x!"}
     };
  protected Object[][] getContents() {
    return sMessageStrings;
  }
}
```

25.8.4 How to Implement a Custom Constraint Error Handling Routine

If the default facility for assigning a custom message to a database constraint violation does not meet your needs, you can implement your own custom constraint error handling routine. Doing this requires creating a custom framework extension class for the ADF transaction class, which you then configure your application module to use at runtime.

25.8.4.1 Creating a Custom Database Transaction Framework Extension Class

To write a custom framework extension class for the ADF transaction, create a class like the `CustomDBTransactionImpl` shown in Example 25–13. This example overrides the transaction object's `postChanges()` method to wrap the call to `super.postChanges()` with a `try/catch` block in order to perform custom processing on any `DMLConstraintException` errors that might be thrown. In this simple example, the only custom processing being performed is a call to `ex.setExceptions(null)` to clear out any nested detail exceptions that the

DMLConstraintException might have. Instead of this, you could perform any other kind of custom exception processing required by your application, including throwing a *custom* exception, provided your custom exception extends JboException directly or indirectly.

Example 25–13 Custom Database Transaction Framework Extension Class

```
package devguide.advanced.customerrs;
import oracle.jbo.DMLConstraintException;
import oracle.jbo.JboException;
import oracle.jbo.common.StringManager;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.TransactionEvent;
public class CustomDBTransactionImpl  extends DBTransactionImpl2 {
  public void postChanges(TransactionEvent te) {
    try {
      super.postChanges(te);
    }
    /*
     * Catch the DML constraint exception
     * and perform custom error handling here
     */
    catch (DMLConstraintException ex) {
      ex.setExceptions(null);
      throw ex;
    }
  }
}
```

25.8.4.2 Configuring an Application Module to Use a Custom Database Transaction Class

In order for your application module to use a custom database transaction class at runtime, you must:

1. Provide a custom implementation of the DatatabaseTransactionFactory class that overrides the create() method to return an instance of the customized transaction class.
2. Configure the value of the TransactionFactory property to be the fully-qualified name of this custom transaction factory class.

Example 25–14 shows a custom database transaction factory class that does this. It returns a new instance of the CustomDBTransactionImpl class when the framework calls the create() method on the database transaction factory.

Example 25–14 Custom Database Transaction Factory Class

```
package devguide.advanced.customerrs;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.DatabaseTransactionFactory;
public class CustomDatabaseTransactionFactory
       extends DatabaseTransactionFactory {
  public CustomDatabaseTransactionFactory() {
  }
  /**
   * Return an instance of our custom ToyStoreDBTransactionImpl class
   * instead of the default implementation.
   *
   * @return instance of custom CustomDBTransactionImpl implementation.
   */
  public DBTransactionImpl2 create() {
    return new CustomDBTransactionImpl();
  }
}
```

To complete the job, use the **Properties** tab of the Configuration Editor to assign the value `devguide.advanced.customerrs.CustomDatabaseTransactionFactory` to the `TransactionFactory` property. When you run the application using this configuration, your custom transaction class will be used.

25.9 Creating Extended Components Using Inheritance

Whenever you create a new business component, if necessary, you can extend an existing one to create a customized version of the original. For example, in the SRDemo application, as shown in Figure 25–9, the `ServiceRequestsByStatus` view object extends the `ServiceRequests` view object to add a named bind variable named `TheStatus` and to customize the `WHERE` clause to reference that bind variable.

Figure 25–9 ADF Business Components Can Extend Another Component

While the figure shows a view object example, this component inheritance facility is available for all component types. When one component extends another, the extended component inherits all of the metadata and behavior from the parent it extends. In the extended component, you can add new features or customize existing features of its parent component both through metadata and Java code.

Note: The examples in this section refer to the `BaseProject` project in the `AdvancedExamples` workspace. See the note at the beginning of this chapter for download instructions.

25.9.1 How To Create a Component That Extends Another

To create an extended component, use the component wizard in the **New Gallery** for the type of component you want to create. For example, to create an extended view object, you use the Create View Object wizard. On the **Name** page of the wizard — in addition to specifying a name and a package for the new component — provide the fully-qualified name of the component that you want to extend in the **Extends** field. To pick the component name from a list, use the **Browse** button next to the **Extends** field. Then, continue to create the extended component in the normal way using the remaining panels of the wizard.

25.9.2 What Happens When You Create a Component That Extends Another

As you've learned, the ADF business components you create are comprised of an XML component definition and an optional Java class. When you create a component that extends another, JDeveloper reflects this component inheritance in both the XML component definition and in any generated Java code for the extended component.

25.9.2.1 Understanding an Extended Component's XML Descriptor

JDeveloper notes the name of the parent component in the new component's XML component definition by adding an `Extends` attribute to the root component element. Any new declarative features you add or any aspects of the parent component's definition you've overridden appear in the extended component's XML component definition. In contrast, metadata that is purely inherited from the parent component is not repeated for the extended component.

Example 25–15 shows what the `ServiceRequstsByStatus.xml` XML component definition for the `ServiceRequstsByStatus` view object looks like. Notice the `Extends` attribute on the <ViewObject> element, <Variable> element related to the additional bind variable added in the extended view object, and the overridden value of the `Where` attribute for the WHERE clause that was modified to reference the `StatusCode` bind variable.

Example 25–15 Extended Component Reflects Parent in Its XML Descriptor

```
<ViewObject
   Name="ServiceRequestsByStatus"
   Extends="oracle.srdemo.model.queries.ServiceRequests"
   Where="((ServiceRequest.CREATED_BY = CreatedByUser.USER_ID)
          AND (ServiceRequest.ASSIGNED_TO = AssignedToUser.USER_ID(+)))
          AND (ServiceRequest.PROD_ID = Product.PROD_ID)
          AND STATUS LIKE NVL(:StatusCode,&#39;%&#39;)"
   OrderBy="REQUEST_DATE DESC"
   BindingStyle="OracleName"
   CustomQuery="false"
   ComponentClass="oracle.srdemo.model.queries.ServiceRequestsByStatusImpl"
   FetchMode="FETCH_AS_NEEDED"
   UseGlueCode="false" >
```

```
   <Variable
      Name="StatusCode"
      Kind="where"
      Type="java.lang.String"
      DefaultValue="%" >
   </Variable>
</ViewObject>
```

25.9.2.2 Understanding Java Code Generation for an Extended Component

If you enable custom Java code for an extended component, JDeveloper automatically generates the Java classes to extend the respective Java classes of its parent component. In this way, the extended component can override any aspect of the parent component's programmatic behavior as necessary. If the parent component is an XML-only component with no custom Java class of its own, the extended component's Java class extends whatever base Java class the parent would use at runtime. This could be the default ADF Business Components framework class in the `oracle.jbo.server` package, or could be your own framework extension class if you have specified that in the **Extends** dialog of the parent component.

In addition, if the extended component is an application module or view object and you enable client interfaces on it, JDeveloper automatically generates the extended component's client interfaces to extend the respective client interfaces of the parent component. If the respective client interface of the parent component does not exist, then the extended component's client interface directly extends the appropriate base ADF Business Components interface in the `oracle.jbo` package.

25.9.3 What You May Need to Know

25.9.3.1 You Can Use Parent Classes and Interfaces to Work with Extended Components

Since an extended component is a customized version of its parent, code you write that works with the *parent* component's Java classes or its client interfaces works without incident for either the parent component *or* any customized version of that parent component.

For example, assume you have a base `Products` view object with custom Java classes and client interfaces like:

- class `ProductsImpl`
- row class `ProductsRowImpl`
- interface `Products`
- row interface `ProductsRow`

If you create a `ProductsByName` view object that extends `Products`, then you can use the base component's classes and interface to work both with `Products` and `ProductsByName`.

Example 25–16 illustrates a test client program that works with the `Products`, `ProductsRow`, `ProductsByName`, and `ProductsByNameRow` client interfaces. A few interesting things to note about the example are the following:

1. You can use parent `Products` interface for working with the `ProductsByName` view object that extends it.
2. Alternatively, you can cast an instance of the `ProductsByName` view object to its own more specific `ProductsByName` client interface.
3. You can test if row `ProductsRow` is actually an instance of the more specific `ProductsByNameRow` before casting it and invoking a method specific to the `ProductsByNameRow` interface.

Example 25–16 Working with Parent and Extended Components

```
package devguide.advanced.extsub;
/* imports omitted */
public class TestClient {
  public static void main(String[] args) {
    String        amDef = "devguide.advanced.extsub.ProductModule";
    String        config = "ProductModuleLocal";
    ApplicationModule am =
    Configuration.createRootApplicationModule(amDef,config);
    Products products = (Products)am.findViewObject("Products");
    products.executeQuery();
    ProductsRow product = (ProductsRow)products.first();
    printAllAttributes(products,product);
    testSomethingOnProductsRow(product);
    // 1. You can use parent Products interface for ProductsByName
    products = (Products)am.findViewObject("ProductsById");
    // 2. Or cast it to its more specific ProductsByName interface
    ProductsByName productsById = (ProductsByName)products;
    productsById.setProductName("Ice");
    productsById.executeQuery();
    product = (ProductsRow)productsById.first();
    printAllAttributes(productsById,product);
    testSomethingOnProductsRow(product);
    am.getTransaction().rollback();
    Configuration.releaseRootApplicationModule(am,true);
  }
  private static void testSomethingOnProductsRow(ProductsRow product) {
    try {
      // 3. Test if row is a ProductsByNameRow before casting
      if (product instanceof ProductsByNameRow) {
        ProductsByNameRow productByName = (ProductsByNameRow)product;
        productByName.someExtraFeature("Test");
      }
      product.setName("Q");
      System.out.println("Setting the Name attribute to 'Q' succeeded.");
    }
    catch (ValidationException v) {
      System.out.println(v.getLocalizedMessage());
    }
  }
  private static void printAllAttributes(ViewObject vo, Row r) {
    String viewObjName = vo.getName();
    System.out.println("Printing attribute for a row in VO '"+
                       viewObjName+"'");
    StructureDef def = r.getStructureDef();
    StringBuilder sb = new StringBuilder();
```

```
    int numAttrs = def.getAttributeCount();
    AttributeDef[] attrDefs = def.getAttributeDefs();
    for (int z = 0; z < numAttrs; z++) {
      Object value = r.getAttribute(z);
      sb.append(z > 0 ? "  " : "")
        .append(attrDefs[z].getName())
        .append("=")
        .append(value == null ? "<null>" : value)
        .append(z < numAttrs - 1 ? "\n" : "");
    }
    System.out.println(sb.toString());
  }
}
```

Running the test client above produces the following results:

```
Printing attribute for a row in VO 'Products'
ProdId=100
  Name=Washing Machine W001
  Checksum=I am the Product Class
Setting the Name attribute to 'Q' succeeded.
Printing attribute for a row in VO 'ProductsById'
ProdId=119
  Name=Ice Maker I012
  Checksum=I am the Product Class
  SomeExtraAttr=SomeExtraAttrValue
## Called someExtraFeature of ProductsByNameRowImpl
Setting the Name attribute to 'Q' succeeded.
```

Note: In this example, Products is an entity-based view object based on the `Product` entity object. The `Product` entity object includes a transient `Checksum` attribute that returns the string "I am the Product class". You'll learn more about why this was included in the example in Section 25.10, "Substituting Extended Components In a Delivered Application".

25.9.3.2 Class Extends is Disabled for Extended Components

When you create an extended component, the **Class Extends** button on the **Java** page of the extended component is disabled. This is due to the fact that JDeveloper automatically extends the appropriate class of its parent component, so it does not make sense to allow you to select a different class.

25.9.3.3 Interesting Aspects You Can Extend for Key Component Types

Entity Objects

When you create an extended entity object, you can introduce new attributes, new associations, new validators, and new custom code. You can override certain declarative aspects of existing attributes as well as overriding any method from the parent component's class.

View Objects

When you create an extended view object, you can introduce new attributes, new view links, new bind variables, and new custom code. You can override certain declarative aspects of existing attributes as well as overriding any method from the parent component's class.

Application Modules

When you create an extended application module, you can introduce new view object instances or new nested application module instance and new custom code. You can also override any method from the parent component's class.

25.9.3.4 Extended Components Have Attribute Indices Relative to Parent

If you add new attributes in an extended entity object or view object, the attribute index numbers are computed relative to the parent component. For example, consider the `Products` view object mentioned above. If you enable a custom view row class, it might have attribute index constants defined in the `ProductsRowImpl.java` class like this:

```
public class ProductsRowImpl extends ViewRowImpl
                             implements ProductsRow {
  public static final int PRODID = 0;
  public static final int NAME = 1;
  public static final int CHECKSUM = 2;
  //etc.
}
```

When you create an extended view object like `ProductsByName`, if that view object adds an addition attribute like `SomeExtraAttr` and has a custom view row class enabled, then its attribute constants will be computed relative to the maximum value of the attribute constants in the parent component:

```
public class ProductsByNameRowImpl extends ProductsRowImpl
                                   implements ProductsByNameRow {
  public static final int MAXATTRCONST =
    ViewDefImpl.getMaxAttrConst("devguide.advanced.extsub.Products");
  public static final int SOMEEXTRAATTR = MAXATTRCONST;
```

Additional attributes would have index values of `MAXATTRCONST+1`, `MAXATTRCONST+2`, etc.

25.9.3.5 Design Time Limitations for Changing Extends After Creation

After defining an extended component, JDeveloper allows you to change the parent component from which an extended component inherits. You can accomplish by:

- Selecting the extended component in the Application Navigator
- Using the Property Inspector to change the `Extends` property

However, you cannot currently use this technique to change the extended component to not inherit from any parent. The one exception to this limitation is the entity object, whose component editor offers an **Extends** field on the **Name** page that you can blank out if necessary. For all other extended components, to make them no longer extend from a parent component you need to delete and recreate them to accomplish this.

25.10 Substituting Extended Components In a Delivered Application

If you deliver packaged applications that can require on-site customization for each potential client of your solution, ADF Business Components offers a useful feature to simplify that task.

Note: The examples in this section refer to the `BaseProject` and `ExtendAndSubstitute` projects in the `AdvancedExamples` workspace. See the note at the beginning of this chapter for download instructions.

25.10.1 Extending and Substituting Components Is Superior to Modifying Code

All too often, on-site application customization is performed by making direct changes to the source code of the delivered application. This approach demonstrates its weaknesses whenever you deliver patches or new feature releases of your original application to your clients. Any customizations they had been applied to the base application's source code need to be painstakingly re-applied to the patched or updated version of the base application. Not only does this render the application customization a costly, ongoing maintenance expense, it can introduce subtle bugs due to human errors that occur when reapplying previous customzations to new releases.

ADF Business Components offers a superior, component-based approach to support application customization that doesn't require changing — or even having access to — the base application's source code. To customize your delivered application, your customers can:

1. Import one or more packages of components from the base application into a new project.
2. Create new components to effect the application customization, extending appropriate parent components from the base application as necessary.
3. Define a list of global component substitutions, naming their customized components to substitute for your base application's appropriate parent components.

When the customer runs your delivered application with a global component substitution list defined, their customized application components are used by your delivered application without changing any of its code. When you deliver a patched or updated version of the original application, their component customizations apply to the updated version the next time they restart the application without needing to re-apply any customizations.

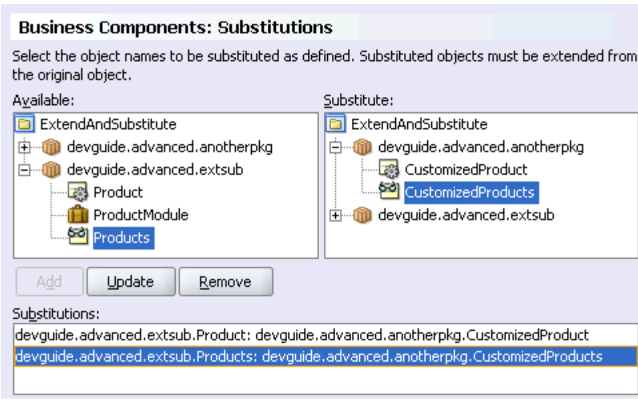
25.10.2 How To Substitute an Extended Component

To define global component substitutions, use the **Business Components > Substitutions** page of the **Project Properties** dialog in the project where you've created extended components based on the imported components from the base application. As shown in Figure 25–10, to define each component substitution:

1. Select the base application's component in the **Available** list.
2. Select the customized, extended component to substitute in the **Substitute** list.
3. Click **Add**.

Note: You can only substitute a component in the base application with an extended component that inherits directly or indirectly from the base one.

Figure 25–10 Defining Business Components Substitutions



25.10.3 What Happens When You Substitute

When you define a list of global component substitutions in a project named *YourExtendsAndSubstitutesProject*, the substitution list is saved in the *YourExtendsAndSubstitutesProject*.jpx in the root directory of the source path.

The file will contain <Substitute> elements as shown in Example 25–17, one for each component to be substituted.

Example 25–17 Component Substitution List Saved in the Project's JPX File

```
<JboProject
   Name="ExtendAndSubstitute"
   SeparateXMLFiles="true"
   PackageName="" >
   <Containee
      Name="anotherpkg"
      FullName="devguide.advanced.anotherpkg.anotherpkg"
      ObjectType="JboPackage" >
   </Containee>
   <Containee
      Name="extsub"
      FullName="devguide.advanced.extsub.extsub"
      ObjectType="JboPackage" >
      <DesignTime>
         <Attr Name="_LocationURL"
               Value="../../BaseProject/deploy/BaseProjectCSMT.jar" />
      </DesignTime>
   </Containee>
   <Substitutes>
      <Substitute OldName="devguide.advanced.extsub.Product"
                  NewName="devguide.advanced.anotherpkg.CustomizedProduct" />
      <Substitute OldName="devguide.advanced.extsub.Products"
                  NewName="devguide.advanced.anotherpkg.CustomizedProducts" />
   </Substitutes>
</JboProject>
```

25.10.4 Enabling the Substituted Components in the Base Application

To have the original application use the set of substituted components, define the Java system property `Factory-Substitution-List` and set its value to the name of the project whose `*.jpx` file contains the substitution list. The value should be just the project name without any `*.jpr` or `*.jpx` extension.

Consider a simple example that customizes the `Product` entity object and the `Products` view object described in Section 25.9.3.1, "You Can Use Parent Classes and Interfaces to Work with Extended Components". To perform the customization, assume you create new project named `ExtendsAndSubstitutes` that:

- Defines a library for the JAR file containing the base components
- Imports the package containing `Product` and `Products`
- Creates new extended components in a distinct package name called `CustomizedProduct` and `CustomizedProducts`
- Defines a component substitution list to use the extended components.

When creating the extended components, assume that you:

- Added an extra view attribute named `ExtraViewAttribute` to the CustomizedProducts view object.
- Added a new validation rule to the `CustomizedProduct` entity object to enforce that the product name cannot be the letter "`Q`".
- Overrode the `getChecksum()` method in the `CustomizedProduct.java` class to return "**I am the CustomizedProduct Class**".

If you define the `Factory-Substitution-List` Java system property set to the value `ExtendsAndSubstitutes`, then when you run the exact same test client class shown above in Example 25–16 the output of the sample will change to reflect the use of the substituted components:

```
Printing attribute for a row in VO 'Products'
ProdId=100
  Name=Washing Machine W001
  Checksum=I am the CustomizedProduct Class
  ExtraViewAttribute=Extra Attr Value
The name cannot be Q!
Printing attribute for a row in VO 'ProductsById'
ProdId=119
  Name=Ice Maker I012
  Checksum=I am the CustomizedProduct Class
  SomeExtraAttr=SomeExtraAttrValue
## Called someExtraFeature of ProductsByNameRowImpl
The name cannot be Q!
```

Compared to the output from Example 25–16, notice that in the presence of the factory substitution list, the `Products` view object in the original test program now has the additional `ExtraViewAttribute`, now reports a `Checksum` attribute value of "I am the CustomizedProduct Class", and now disallows the assignment of the product name to have the value "Q". These component behavior changes were performed without needing to modify the original Java or XML source code of the delivered components.

26

Advanced Entity Object Techniques

This chapter describes advanced techniques for use in the entity objects in your business domain layer.

This chapter includes the following sections:

- Section 26.1, "Creating Custom, Validated Data Types Using Domains"
- Section 26.2, "Updating a Deleted Flag Instead of Deleting Rows"
- Section 26.3, "Advanced Entity Association Techniques"
- Section 26.4, "Basing an Entity Object on a PL/SQL Package API"
- Section 26.5, "Basing an Entity Object on a Join View or Remote DBLink"
- Section 26.6, "Using Inheritance in Your Business Domain Layer"
- Section 26.7, "Controlling Entity Posting Order to Avoid Constraint Violations"
- Section 26.8, "Implementing Automatic Attribute Recalculation"
- Section 26.9, "Implementing Custom Validation Rules"

Note: To experiment with a working version of the examples in this chapter, download the `AdvancedEntityExamples` workspace from the *Example Downloads* page at `http://otn.oracle.com/documentation/jdev/b25947_01/`.

26.1 Creating Custom, Validated Data Types Using Domains

When you find yourself repeating the same sanity-checking validations on the values of similar attributes across multiple entity objects, you can save yourself time and effort by creating your own data types that encapsulate this validation. For example, imagine that across your business domain layer there are numerous entity object attributes that store strings that represent email addresses. One technique you could use to ensure that end-users always enter a valid email address everywhere one appears in your business domain layer is to:

- Use a basic `String` data type for each of these attributes
- Add an attribute-level method validator with Java code that ensures that the String value has the format of a valid email address for each attribute

However, these approaches could get tedious quickly in a large application. Luckily, ADF Business Components offers an alternative that allows you to create your own `EmailAddress` data type that represents email addresses. After centralizing all of the sanity-checking regarding email address values into this new custom data type, you can use the `EmailAddress` as the type of every attribute in your application that represents an email address. By doing this, you make the intention of the attribute values more clear to other developers and simplify application maintenance by putting the validation in a single place. ADF Business Components calls these developer-created data types domains.

> **Note:** The examples in this section refer to the `SimpleDomains` project in the `AdvancedEntityExamples` workspace. See the note at the beginning of this chapter for download instructions. Run the `CreateObjectType.sql` script in the **Resources** folder against the `SRDemo` connection to set up the additional database objects required for the project.

26.1.1 What Are Domains?

Domains are Java classes that extend the basic data types like `String`, `Number`, and `Date` to add constructor-time validation to insure the candidate value passes relevant sanity checks. They offer you a way to define custom data types with cross-cutting behavior such as basic data type validation, formatting, and custom metadata properties in a way that are inherited by any entity objects or view objects that use the domain as the Java type of any of their attributes.

26.1.2 How To Create a Domain

To create a domain, use the Create Domain wizard. This is available in the **New Gallery** in the **ADF Business Components** category.

In step 1, on the **Name** panel specify a name for the domain and a package in which it will reside. To create a domain based on a simple Java type, leave the **Domain for an Oracle Object Type** unchecked.

In step 2, on the **Settings** panel, indicate the base type for the domain and the database column type to which it will map. For example, if you were creating a domain called `ShortEmailAddress` to hold eight-character short email addresses, you would set the base type to `String` and the **Database Column Type** to `VARCHAR2(8)`. You can set other common attribute settings on this panel as well.

Then, click **Finish** to create your domain.

26.1.3 What Happens When You Create a Domain

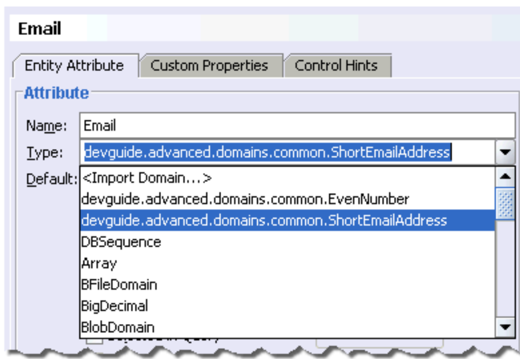
When you create a domain, JDeveloper creates its XML component definition in the subdirectory of your project's source path that corresponds to the package name you chose. For example, if you created the `ShortEmailAddress` domain in the `devguide.advanced.domains` package, JDeveloper would create the `ShortEmailAddress.xml` file in the `./devguide/advanced/domains` subdirectory. A domain always has a corresponding Java class, which JDeveloper creates in the `common` subpackage of the package where the domain resides. This means it would create the `ShortEmailAddress.java` class in the `devguide.advanced.domains.common` package. The domain's Java class is automatically generated with the appropriate code to behave in a way that is identical to one of the built-in data types.

26.1.4 What You May Need to Know

26.1.4.1 Using Domains for Entity and View Object Attributes

Once you've created a domain in a project, it automatically appears among the list of available data types in the **Attribute Type** dropdown list in the entity object and view object wizards and editors as shown in Figure 26–1. To use the domain as the type of a given attribute, just pick it from the list.

Figure 26–1 Custom Domain Types in the Attribute Type Dropdown List



Note: The entity-mapped attributes in an entity-based view object inherit their data type from their corresponding underlying entity object attribute, so if the entity attribute uses a domain type, so will the matching view object attribute. For transient or SQL-derived view object attributes, you can directly set the type to use a domain since it is not inherited from any underlying entity.

26.1.4.2 Validate Method Should Throw DataCreationException If Sanity Checks Fail

Typically, the only coding task you need to do for a domain is to write custom code inside the generated `validate()` method. Your implementation of the `validate()` method should perform your sanity checks on the candidate value being constructed, and throw a `DataCreationException` in the `oracle.jbo` package if the validation fails.

In order to throw an exception message that is translatable, you can create a message bundle class similar to the one shown in Example 26–1. Create it in the same `common` package as your domain classes themselves. The message bundle returns an array of {*MessageKeyString*, *TranslatableMessageString*} pairs.

Example 26–1 Custom Message Bundle Class For Domain Exception Messages

```
package devguide.advanced.domains.common;
import java.util.ListResourceBundle;
public class ErrorMessages extends ListResourceBundle {
  public static final String INVALID_SHORTEMAIL = "30002";
  public static final String INVALID_EVENNUMBER = "30003";
  private static final Object[][] sMessageStrings = new String[][] {
      { INVALID_SHORTEMAIL,
        "A valid short email address has no @-sign or dot."},
      { INVALID_EVENNUMBER,
```

```
        "Number must be even."}
    };

  /**
   * Return String Identifiers and corresponding Messages
   * in a two-dimensional array.
   */
  protected Object[][] getContents() {
    return sMessageStrings;
  }
}
```

26.1.4.3 String Domains Aggregate a String Value

Since `String` is a base JDK type, a domain based on a `String` aggregates a `private mData String` member field to hold the value that the domain represents. Then, the class implements the `DomainInterface` expected by the ADF runtime, as well as the `Serializable` interface, so the domain can be used in method arguments or returns types of ADF components custom client interfaces.

Example 26–2 shows the `validate()` method for a simple `ShortEmailAddress` domain class. It tests to make sure that the `mData` value does not contains an at-sign or a dot, and if it does, then the method throws `DataCreationException` referencing an appropriate message bundle and message key for the translatable error message.

Example 26–2 Simple ShortEmailAddress String-Based Domain Type with Custom Validation

```
public class ShortEmailAddress implements DomainInterface, Serializable {
  private String mData;
  // etc.
  /**Implements domain validation logic and throws a JboException on error. */
  protected void validate() {
    int atpos = mData.indexOf('@');
    int dotpos = mData.lastIndexOf('.');
    if (atpos > -1 || dotpos > -1) {
      throw new DataCreationException(ErrorMessages.class,
      ErrorMessages.INVALID_SHORTEMAIL,null,null);
    }
  }
  // etc.
}
```

26.1.4.4 Other Domains Extend Existing Domain Type

Other simple domains based on a built-in type in the `oracle.jbo.domain` package extend the base type as shown in Example 26–3. It illustrates the `validate()` method for a simple Number-based domain called `EvenNumber` that represents even numbers.

Example 26–3 Simple EvenNumber Number-Based Domain Type with Custom Validation

```
public class EvenNumber extends Number {
  // etc.
  /**
   * Validates that value is an even number, or else
   * throws a DataCreationException with a custom
   * error message.
   */
  protected void validate() {
    if (getValue() % 2 == 1) {
      throw new DataCreationException(ErrorMessages.class,
      ErrorMessages.INVALID_EVENNUMBER,null,null);
    }
  }
  // etc.
}
```

26.1.4.5 Simple Domains are Immutable Java Classes

When you create a simple domain based on one of the basic data types, it is an *immutable* class. This just means that once you've constructed a new instance of it like this:

```
ShortEmailAddress email = new ShortEmailAddress("smuench");
```

You cannot change its value. If you want to reference a different short email address, you just construct another one:

```
ShortEmailAddress email = new ShortEmailAddress("bribet");
```

This is not a new concept since it's the same way that `String`, `Number`, and `Date` classes behave, among others.

26.1.4.6 Creating Domains for Oracle Object Types When Useful

The Oracle database supports the ability to create user-defined types in the database. For example, you could create a type called `POINT_TYPE` using the following DDL statement:

```
create type point_type as object (
   x_coord number,
   y_coord number
);
```

If you use user-defined types like `POINT_TYPE`, you can create domains base on them, or you can reverse-engineer tables containing columns of object type to have JDeveloper create the domain for you.

Manually Creating Object Type Domains

To create a domain yourself, do the following in the Create Domain wizard:

- In step 1 of the Create Domain wizard on the **Name** panel, check the **Domain for an Oracle Object Type** checkbox, then select the object type for which you want to create a domain from the **Available Types** list.
- In step 2 on the **Settings** panel, use the **Attribute** dropdown list to switch between the multiple domain properties to adjust the settings as appropriate.
- Click **Finish**

Reverse-Engineering Object Type Domains

In addition to manually creating object type domains, when you use the Business Components from Tables wizard and select a table containing columns of an Oracle object type, JDeveloper automatically creates domains for those object types as part of the reverse-engineering process. For example, imagine you created a table like this with a column of type POINT_TYPE:

```
create table interesting_points(
  id number primary key,
  coordinates point_type,
  description varchar2(20)
);
```

If you create an entity object for the INTERESTING_POINTS table in the Business Components from Tables wizard, then you will get both an InterestingPoints entity object and a PointType domain. The latter will have been automatically created, based on the POINT_TYPE object type, since it was required as the data type of the Coordinates attribute of the InterestingPoints entity object.

Unlike simple domains, object type domains are mutable. JDeveloper generates getter and setter methods into the domain class for each of the elements in the object type's structure. After changing any domain properties, when you set that domain as the value of a view object or entity object attribute, it is treated as a single unit. ADF does not track which domain properties have changed, only that a domain-valued attribute value has changed.

Note: Domains based on Oracle object types are useful for working programmatically with data whose underlying type is an oracle object type. They also can simplify passing and receiving structure information to stored procedures. However, support for working with object type domains in the ADF binding layer is complete, so it's not straightforward to use object domain-valued attributes in declaratively-databound user interfaces.

26.1.4.7 Quickly Navigating to the Domain Class

After selecting a domain in the Application Navigator, you can quickly navigate to its implementation class by:

- Choosing **Go to Domain Class** on the right-mouse context menu, or
- Double-clicking on the domain class in the Structure Window

26.1.4.8 Domains Get Packaged in the Common JAR

When you create a business components archive, as described in Section 25.7, "Working with Libraries of Reusable Business Components", the domain classes and message bundle files in the *.common subdirectories of your project's source path get packaged into the *CSCommon.jar. They are classes that are common to both the middle-tier application server and to an eventual remote-client you might need to support.

26.1.4.9 Entity and View Object Attributes Inherit Custom Domain Properties

You can define custom metadata properties on a domain. Any entity object or view object attribute based on that domain inherits those custom properties as if they had been defined on the attribute itself. If the entity object or view object attribute defines the same custom property, its setting takes precedence over the value inherited from the domain.

26.1.4.10 Domain Settings Cannot Be Less Restrictive at Entity or View Level

JDeveloper will enforce declarative settings you impose at the domain definition level cannot be made *less* restrictive in the Entity Object editor or View Object editor for an attribute based on the domain type. For example, if you define a domain to have its **Updatable** property set to **While New**, then when you use your domain as the Java type of an entity object attribute, you can set **Updatable** to be **Never** (more restrictive) but you cannot set it to be **Always**. Similarly, if you define a domain to be **Persistent**, you cannot make it transient later. When sensible for your application, set declarative properties for a domain to be as lenient as possible so you can later make them more restrictive as needed.

26.2 Updating a Deleted Flag Instead of Deleting Rows

For auditing purposes, once a row is added to a table, sometimes your requirements may demand that rows are never physically deleted from the table. Instead, when the end-user deletes the row in the user interface, the value of a `DELETED` column should be updated from "`N`" to "`Y`" to mark it as deleted. This section explains the two method overrides required to alter an entity object's default behavior to achieve this effect. The following sections assume you want to change the `Product` entity from the SRDemo application to behave in this way. They presume that you've altered the PRODUCTS table to have an additional `DELETED` column, and synchronized the `Product` entity with the database to add the corresponding `Deleted` attribute.

26.2.1 How to Update a Deleted Flag When a Row is Removed

To update a deleted flag when a row is removed, enable a custom Java class for your entity object and override the `remove()` method to set the deleted flag before calling the `super.remove()` method. Example 26–4 shows what this would look like in the `ProductImpl` class of the SRDemo application's `Product` entity object. It is important to set the attribute *before* calling `super.remove()` since an attempt to set the attribute of a deleted row will encounter the `DeadEntityAccessException`.

Example 26–4 Updating a Deleted Flag When a Product Entity Row is Removed

```
// In ProductImpl.java
public void remove() {
  setDeleted("Y");
  super.remove();
}
```

The row will still be removed from the row set, but it will have the value of its Deleted flag modified to "Y" in the entity cache. The second part of implementing this behavior involves forcing the entity to perform an `UPDATE` instead of an `INSERT` when it is asked to perform its DML operation. You need to implement both parts for a complete solution.

26.2.2 Forcing an Update DML Operation Instead of a Delete

To force an entity object to be updated instead of deleted, override the `doDML()` method and write code that conditionally changes the `operation` flag. When the operation flag equals `DML_DELETE`, your code will change it to `DML_UPDATE` instead. Example 26–5 shows what this would look like in the `ProductImpl` class of the SRDemo application's Product entity object.

Example 26–5 Forcing an Update DML Operation Instead of a Delete

```
// In ProductImpl.java
protected void doDML(int operation, TransactionEvent e) {
   if (operation == DML_DELETE) {
     operation = DML_UPDATE;
   }
   super.doDML(operation, e);
  }
```

With this overridden `doDML()` method in place to complement the overridden `remove()` method described in the previous section, any attempt to remove a `Product` entity through any view object with a `Product` entity usage will update the `DELETED` column instead of physically deleting the row. Of course, in order to prevent "deleted" products from appearing in your view object query results, you will need to appropriately modify their WHERE clauses to include only products `WHERE DELETED = 'N'`.

26.3 Advanced Entity Association Techniques

This section describes several advanced techniques for working with associations between entity objects.

26.3.1 Modifying Association SQL Clause to Implement Complex Associations

When you need to represent a more complex relationship between entities than one based only on the equality of matching attributes, you can modify the association's SQL clause to include more complex criteria. For example, sometimes the relationship between two entities depends on effectivity dates. A `ServiceRequest` may be related to a `Product`, however if the name of the product changes over time, each row in the PRODUCTS table might include additional `EFFECTIVE_FROM` and `EFFECTIVE_UNTIL` columns that track the range of dates in which that product row is (or was) in use. The relationship between a `ServiceRequest` and the `Product` with which it is associated might then be described by a combination of the matching `ProdId` attributes and a condition that the service request's `RequestDate` lie between the product's `EffectiveFrom` and `EffectiveUntil` dates.

You can setup this more complex relationship in the Association Editor. First add any additional necessary attribute pairs on the **Entity Objects** page, which in this example would include one (`EffectiveFrom`, `RequestDate`) pair and one (`EffectiveUntil`, `RequestDate`) pair. Then on the **Association SQL** page you can edit the **Where** field to change the WHERE clause to be:

```
(:Bind_ProdId = ServiceRequest.PROD_ID) AND
(ServiceRequest.REQUEST_DATE BETWEEN :Bind_EffectiveFrom
                                 AND :Bind_EffectiveUntil)
```

26.3.2 Exposing View Link Accessor Attributes at the Entity Level

When you create a view link between two entity-based view objects, on the **View Link Properties** page, you have the option to expose view link accessor attributes both at the view object level as well as at the entity object level. By default, a view link accessor is only exposed at the view object level of the destination view object. By checking the appropriate **In Entity Object: *SourceEntityName*** or **In Entity Object:*DestinationEntityName*** checkbox, you can opt to have JDeveloper include a view link attribute in either or both of the source or destination entity objects. This can provide a handy way for an entity object to access a related set of related view rows, especially when the query to produce the rows only depends on attributes of the current row.

26.3.3 Optimizing Entity Accessor Access By Retaining the Row Set

Each time you retrieve an entity association accessor row set, by default the entity object creates a new `RowSet` object to allow you to work with the rows. This does *not* imply re-*executing* the query to produce the results each time, only creating a new instance of a `RowSet` object with its default iterator reset to the "slot" before the first row. To force the row set to refresh its rows from the database, you can call its `executeQuery()` method.

Since there is a small amount of overhead associated with creating the row set, if your code makes numerous calls to the same association accessor attributes, you can consider enabling the association accessor row set retention for the source entity object in the association. To use the association accessor retention feature, first enable a custom Java *entity collection* class for your entity object. As with other custom entity Java classes you've seen, you do this on the **Java** panel of the Entity Object editor by selecting the **Entity Collection Class** checkbox. Then, in the *`YourEntityColl`*`Impl` class that JDeveloper creates for you, override the `init()` method, and add a line after `super.init()` that calls the `setAssociationAccessorRetained()` method passing `true` as the parameter. It affects all association accessor attributes for that entity object.

When this feature is enabled for an entity object, since the association accessor row set it not recreated each time, the current row of its default row set iterator is also retained as a side-effect. This means that your code will need to explicitly call the `reset()` method on the row set you retrieve from the association accessor to reset the current row in its default row set iterator back to the "slot" before the first row.

Note, however, that with accessor retention enabled, your failure to call `reset()` each time before you iterate through the rows in the accessor row set can result in a subtle, hard-to-detect error in your application. For example, if you iterate over the rows in an association accessor row set like this, for example, to calculate some aggregate total:

```
// In ProductImpl.java
RowSet rs = (RowSet)getServiceRequests();
while (rs.hasNext()) {
  ServiceRequestImpl r = (ServiceRequestImpl)rs.next();
  // Do something important with attributes in each row
}
```

The first time you work with the accessor row set, the code will work. However, since the row set (and its default row set iterator) are retained, the second and subsequent times you access the row set the current row will already be at the end of the row set and the while loop will be skipped since `rs.hasNext()` will be `false`. Instead, with this feature enabled, write your accessor iteration code like this:

```
// In ProductImpl.java
RowSet rs = (RowSet)getServiceRequests();
rs.reset(); // Reset default row set iterator to slot before first row!
while (rs.hasNext()) {
  ServiceRequestImpl r = (ServiceRequestImpl)rs.next();
  // Do something important with attributes in each row
}
```

26.4 Basing an Entity Object on a PL/SQL Package API

If you have a PL/SQL package that encapsulates insert, update, and delete access to an underlying table, you can override the default DML processing event for the entity object that represents that table to invoke the procedures in your PL/SQL API instead. Often, such PL/SQL packages are used in combination with a companion database view. Client programs read data from the underlying table using the database view, and "write" data back to the table using the procedures in the PL/SQL package. This section considers the code necessary to create a `Product` entity object based on such a combination of a view and a package.

Given the `PRODUCTS` table in the SRDemo schema, consider a database view named `PRODUCTS_V`, created using the following DDL statement:

```
create or replace view products_v
as select prod_id,name,image,description from products;
```

In addition, consider the simple `PRODUCTS_API` package shown in Example 26–6 that encapsulates insert, update, and delete access to the underlying `PRODUCTS` table.

Example 26–6 Simple PL/SQL Package API for the PRODUCTS Table

```
create or replace package products_api is
  procedure insert_product(p_prod_id number,
                           p_name varchar2,
                           p_image varchar2,
                           p_description varchar2);
  procedure update_product(p_prod_id number,
                           p_name varchar2,
                           p_image varchar2,
                           p_description varchar2);
  procedure delete_product(p_prod_id number);
end products_api;
```

The following sections explain how to create an entity object based on the above combination of view and package.

Note: The examples in this section refer to the `EntityWrappingPLSQLPackage` project in the `AdvancedEntityExamples` workspace. See the note at the beginning of this chapter for download instructions. Run the `CreateAll.sql` script in the **Resources** folder against the `SRDemo` connection to setup the additional database objects required for the project.

26.4.1 How to Create an Entity Object Based on a View

To create an entity object based on a view, use the Create Entity Object wizard and perform the following steps:

- In step 1 on the **Name** panel, give the entity a name like `Product` and check the **Views** checkbox at the bottom of the **Database Objects** section.

  This enables the display of the available database views in the current schema in the **Schema Object** list.

- Select the desired database view in the **Schema Object** list.
- In step 3 on the **Attribute Settings** panel, use the **Select Attribute** dropdown list to choose the attribute that will act as the primary key, then enable the **Primary Key** setting for that property.

> **Note:** When defining the entity based on a view, JDeveloper cannot automatically determine the primary key attribute since database views do not have related constraints in the database data dictionary.

- Then click **Finish**.

26.4.2 What Happens When You Create an Entity Object Based on a View

By default, an entity object based on a view performs all of the following directly against the underlying database view:

- `SELECT` statement (for `findByPrimaryKey()`)
- `SELECT FOR UPDATE` statement (for `lock()`), and
- `INSERT`, `UPDATE`, `DELETE` statements (for `doDML()`)

The following sections first illustrate how to override the `doDML()` operations, then explain how to extend that when necessary to override the `lock()` and `findByPrimaryKey()` handling in a second step.

26.4.3 Centralizing Details for PL/SQL-Based Entities into a Base Class

If you plan to have more than one entity object based on a PL/SQL API, it's a smart idea to abstract the generic details into a base framework extension class. In doing this, you'll be using several of the concepts you learned in Chapter 25, "Advanced Business Components Techniques". Start by creating a `PLSQLEntityImpl` class that extends the base `EntityImpl` class that each one of your PL/SQL-based entities can use as their base class. As shown in Example 26–7, you'll override the `doDML()` method of the base class to invoke a different helper method based on the operation.

Example 26–7 Overriding doDML() to Call Different Procedures Based on the Operation

```
// In PLSQLEntityImpl.java
protected void doDML(int operation, TransactionEvent e) {
  // super.doDML(operation, e);
  if (operation == DML_INSERT)
    callInsertProcedure(e);
  else if (operation == DML_UPDATE)
    callUpdateProcedure(e);
  else if (operation == DML_DELETE)
    callDeleteProcedure(e);
}
```

In the `PLSQLEntityImpl.java` base class, you can write the helper methods so that they perform the default processing like this:

```
// In PLSQLEntityImpl.java
/* Override in a subclass to perform non-default processing */
protected void callInsertProcedure(TransactionEvent e) {
  super.doDML(DML_INSERT, e);
}
/* Override in a subclass to perform non-default processing */
protected void callUpdateProcedure(TransactionEvent e) {
  super.doDML(DML_UPDATE, e);
}
/* Override in a subclass to perform non-default processing */
protected void callDeleteProcedure(TransactionEvent e) {
  super.doDML(DML_DELETE, e);
}
```

After putting this infrastructure in place, when you base an entity object on the `PLSQLEntityImpl` class, you can use the **Source | Override Methods** menu item to override the `callInsertProcedure()`, `callUpdateProcedure()`, and `callDeleteProcedure()` helper methods and perform the appropriate stored procedure calls for that particular entity. To simplify the task of implementing these calls, you could add the `callStoredProcedure()` helper method you learned about in Chapter 25.5, "Invoking Stored Procedures and Functions" to the `PLSQLEntityImpl` class as well. This way, any PL/SQL-based entity objects that extend this class can leverage the helper method.

26.4.4 Implementing the Stored Procedure Calls for DML Operations

To implement the stored procedure calls for DML operations, do the following:

- Use the **Class Extends** button on the **Java** panel of the Entity Object Editor to set your `Product` entity object to have the `PLSQLEntityImpl` class as its base class.
- Enable a custom Java class for the `Product` entity object.
- Use the **Source | Override Methods** menu item and select the `callInsertProcedure()`, `callUpdateProcedure()`, and `callDeleteProcedure()` methods.

Example 26–8 shows the code you would write in these overridden helper methods.

Example 26–8 Leveraging a Helper Method to Invoke Insert, Update, and Delete Procedures

```
// In ProductImpl.java
protected void callInsertProcedure(TransactionEvent e) {
  callStoredProcedure("products_api.insert_product(?,?,?,?)",
                      new Object[] { getProdId(), getName(), getImage(),
                                     getDescription() });
}
protected void callUpdateProcedure(TransactionEvent e) {
  callStoredProcedure("products_api.update_product(?,?,?,?)",
                      new Object[] { getProdId(), getName(), getImage(),
                                     getDescription() });
}
protected void callDeleteProcedure(TransactionEvent e) {
  callStoredProcedure("products_api.delete_product(?)",
                      new Object[] { getProdId() });
}
```

At this point, if you create a default entity-based view object called Products for the Product entity object and add an instance of it to a ProductModule application module you can quickly test inserting, updating, and deleting rows from the Products view object instance in the Business Components Browser.

Often, overriding just the insert, update, and delete operations will be enough. The default behavior that performs the SELECT statement for findByPrimaryKey() and the SELECT FOR UPDATE statement for the lock() against the database view works for most basic kinds of views.

However, if the view is complex and does not support SELECT FOR UPDATE or if you need to perform the findByPrimaryKey() and lock() functionality using additional stored procedures API's, then you can follow the steps in the next section.

26.4.5 Adding Select and Lock Handling

You can also handle the lock and findByPrimaryKey() functionality of an entity object by invoking stored procedures if necessary. Imagine that the PRODUCTS_API package were updated to contain the two additional procedures shown in Example 26–9. Both the lock_product and select_product procedures accept a primary key attribute as an IN parameter and return values for the remaining attributes using OUT parameters.

Example 26–9 Additional Locking and Select Procedures for the PRODUCTS Table

```
/* Added to PRODUCTS_API package */
  procedure lock_product(p_prod_id number,
                         p_name OUT varchar2,
                         p_image OUT varchar2,
                         p_description OUT varchar2);
  procedure select_product(p_prod_id number,
                           p_name OUT varchar2,
                           p_image OUT varchar2,
                           p_description OUT varchar2);
```

26.4.5.1 Updating PLSQLEntityImpl Base Class to Handle Lock and Select

You can extend the PLSQLEntityImpl base class to handle the lock() and findByPrimaryKey() overrides using helper methods similar to the ones you added for insert, update, delete. At runtime, both the lock() and findByPrimaryKey() operations end up invoking the lower-level entity object method called doSelect(boolean lock). The lock() operation calls doSelect() with a true value for the parameter, while the findByPrimaryKey() operation calls it passing false instead.

Example 26–10 shows the overridden doSelect() method in PLSQLEntityImpl to delegate as appropriate to two helper methods that subclasses can override as necessary.

Example 26–10 Overriding doSelect() to Call Different Procedures Based on the Lock Parameter

```
// In PLSQLEntityImpl.java
protected void doSelect(boolean lock) {
  if (lock) {
    callLockProcedureAndCheckForRowInconsistency();
  } else {
    callSelectProcedure();
  }
}
```

The two helper methods are written to just perform the default functionality in the base `PLSQLEntityImpl` class:

```
// In PLSQLEntityImpl.java
/* Override in a subclass to perform non-default processing */
protected void callLockProcedureAndCheckForRowInconsistency() {
  super.doSelect(true);
}
/* Override in a subclass to perform non-default processing */
protected void callSelectProcedure() {
  super.doSelect(false);
}
```

Notice that the helper method that performs locking has the name `callLockProcedureAndCheckForRowInconsistency()`. This reminds developers that it is their responsibility to perform a check to detect at the time of locking the row whether the newly-selected row values are the same as the ones the entity object in the entity cache believes are the current database values.

To assist subclasses in performing this old-value versus new-value attribute comparison, you can add one final helper method to the `PLSQLEntityImpl` class like this:

```
// In PLSQLEntityImpl
protected void compareOldAttrTo(int attrIndex, Object newVal) {
  if ((getPostedAttribute(attrIndex) == null && newVal != null) ||
      (getPostedAttribute(attrIndex) != null && newVal == null) ||
      (getPostedAttribute(attrIndex) != null && newVal != null &&
       !getPostedAttribute(attrIndex).equals(newVal))) {
    throw new RowInconsistentException(getKey());
  }
}
```

26.4.5.2 Implementing Lock and Select for the Product Entity

With the additional infrastructure in place in the base `PLSQLEntityImpl` class, you can override the `callSelectProcedure()` and `callLockProcedureAndCheckForRowInconsistency()` helper methods in the `Product` entity object's `ProductImpl` class. Since the `select_product` and `lock_product` procedures have `OUT` arguments, as you learned in Section 25.5.4, "Calling Other Types of Stored Procedures", you need to use a JDBC `CallableStatement` object to perform these invocations.

Example 26–11 shows the code required to invoke the `select_product` procedure. It's performing the following basic steps:

1. Creating a `CallableStatement` for the PLSQL block to invoke.
2. Registering the `OUT` parameters and types, by one-based bind variable position.
3. Setting the `IN` parameter value.
4. Executing the statement.
5. Retrieving the possibly updated column values.
6. Populating the possibly updated attribute values in the row.
7. Closing the statement.

Example 26–11 Invoking the Stored Procedure to Select a Row by Primary Key

```
// In ProductImpl.java
protected void callSelectProcedure() {
  String stmt = "begin products_api.select_product(?,?,?,?);end;";
  // 1. Create a CallableStatement for the PLSQL block to invoke
  CallableStatement st = getDBTransaction().createCallableStatement(stmt, 0);
  try {
    // 2. Register the OUT parameters and types
    st.registerOutParameter(2, VARCHAR2);
    st.registerOutParameter(3, VARCHAR2);
    st.registerOutParameter(4, VARCHAR2);
    // 3. Set the IN parameter value
    st.setObject(1,getProdId());
    // 4. Execute the statement
    st.executeUpdate();
    // 5. Retrieve the possibly updated column values
    String possiblyUpdatedName = st.getString(2);
    String possiblyUpdatedImage = st.getString(3);
    String possiblyUpdatedDesc  = st.getString(4);
    // 6. Populate the possibly updated attribute values in the row
    populateAttribute(NAME,possiblyUpdatedName,true,false,false);
    populateAttribute(IMAGE,possiblyUpdatedImage,true,false,false);
    populateAttribute(DESCRIPTION,possiblyUpdatedDesc,true,false,false);
  } catch (SQLException e) {
      throw new JboException(e);
  } finally {
    if (st != null) {
      try {
        // 7. Closing the statement
        st.close();
      } catch (SQLException e) {
      }
    }
  }
}
```

Example 26–12 shows the code to invoke the `lock_product` procedure. It's doing basically the same steps as above, with just the following two interesting differences:

- After retrieving the possibly updated column values from the `OUT` parameters, it uses the `compareOldAttrTo()` helper method inherited from the `PLSQLEntityImpl` to detect whether or not a `RowInconsistentException` should be thrown as a result of the row lock attempt.

- In the `catch (SQLException e)` block, it is testing to see whether the database has thrown the error:

  ```
  ORA-00054: resource busy and acquire with NOWAIT specified
  ```

  and if so, it again throws the ADF Business Components `AlreadyLockedException` just as the default entity object implementation of the `lock()` functionality would do in this situation.

Example 26–12 Invoking the Stored Procedure to Lock a Row by Primary Key

```
// In ProductImpl.java
  protected void callLockProcedureAndCheckForRowInconsistency() {
    String stmt = "begin products_api.lock_product(?,?,?,?);end;";
    CallableStatement st = getDBTransaction().createCallableStatement(stmt, 0);
    try {
      st.registerOutParameter(2, VARCHAR2);
      st.registerOutParameter(3, VARCHAR2);
      st.registerOutParameter(4, VARCHAR2);
      st.setObject(1,getProdId());
      st.executeUpdate();
      String possiblyUpdatedName = st.getString(2);
      String possiblyUpdatedImage = st.getString(3);
      String possiblyUpdatedDesc  = st.getString(4);
      compareOldAttrTo(NAME,possiblyUpdatedName);
      compareOldAttrTo(IMAGE,possiblyUpdatedImage);
      compareOldAttrTo(DESCRIPTION,possiblyUpdatedDesc);
    } catch (SQLException e) {
      if (Math.abs(e.getErrorCode()) == 54) {
        throw new AlreadyLockedException(e);
      } else {
        throw new JboException(e);
      }
    } finally {
      if (st != null) {
        try {
          st.close();
        } catch (SQLException e) {
        }
      }
    }
  }
```

With these methods in place, you have a `Product` entity object that wraps the `PRODUCTS_API` package for all of its database operations. Due to the clean separation of the data querying functionality of view objects and the data validation and saving functionality of entity objects, you can now leverage this `Product` entity object in any way you would use a normal entity object. You can build as many different view objects that use `Product` as their entity usage as necessary.

26.5 Basing an Entity Object on a Join View or Remote DBLink

If you need to create an entity object based on either of the following:

- Synonym that resolves to a remote table over a DBLINK
- View with `INSTEAD OF` triggers

Then you will encounter the following error if any of its attributes are marked as **Refresh on Insert** or **Refresh on Update**:

```
JBO-26041: Failed to post data to database during "Update"
## Detail 0 ##
ORA-22816: unsupported feature with RETURNING clause
```

The error says it all. These types of schema objects to not support the `RETURNING` clause, which by default the entity object uses to more efficiently return the refreshed values in the same database roundtrip in which the `INSERT` or `UPDATE` operation was executed.

To disable the use of the `RETURNING` clause for an entity object of this type, do the following:

1. Enable a custom entity definition class for the entity object.
2. In the custom entity definition class, override the `createDef()` method to call:

   ```
   setUseReturningClause(false)
   ```

3. If the **Refresh on Insert** attribute is the primary key of the entity object, you must identity some other attribute in the entity as an alternate unique key by setting the **Unique Key** property on it.

At runtime, when you have disabled the use of the RETURNING clause in this way, the entity object implements the **Refresh on Insert** and **Refresh on Update** behavior using a separate SELECT statement to retrieve the values to refresh after insert or update as appropriate.

26.6 Using Inheritance in Your Business Domain Layer

Inheritance is a powerful feature of object-oriented development that can simplify development and maintenance when used appropriately. As you've seen in Section 25.9, "Creating Extended Components Using Inheritance", ADF Business Components supports using inheritance to create new components that extend existing ones in order to add additional properties or behavior or modify the behavior of the parent component. This section helps you understand when inheritance can be useful in modeling the different kinds of entities in your reusable business domain layer.

Note: The examples in this section refer to the `InheritanceAndPolymorphicQueries` project in the `AdvancedEntityExamples` workspace. See the note at the beginning of this chapter for download instructions. Run the `AlterUsersTable.sql` script in the **Resources** folder against the `SRDemo` connection to setup the additional database objects required for the project.

26.6.1 Understanding When Inheritance Can be Useful

Your application's database schema might contain tables where different logical kinds of business information are stored in rows of the same table. These tables will typically have one column whose value determines the kind of information stored in each row. For example, the SRDemo application's `USERS` table stores information about end-users, technicians, and managers in the same table. It contains a `USER_ROLE` column whose value — `user`, `technician`, or `manager` — determines what kind of user the row represents.

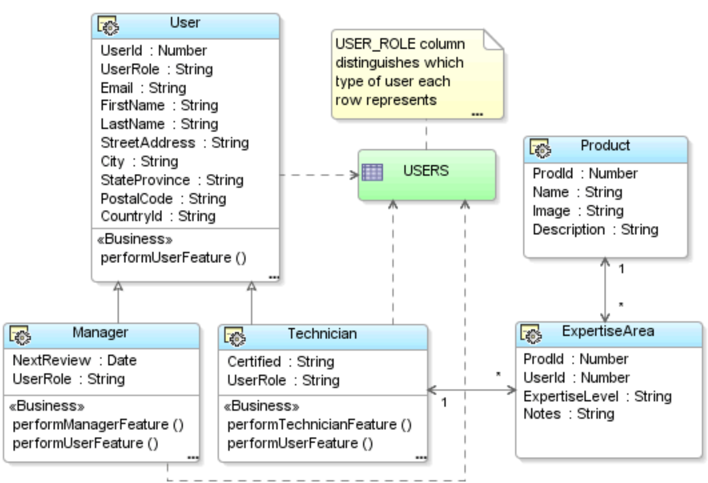
While the simple SRDemo application implementation doesn't yet contain this differentiation in this release, it's reasonable to assume that a future release of the application might require:

- Managing additional database-backed attributes that are specific to managers or specific to technicians
- Implementing common behavior for all users that is different for managers or technicians
- Implementing new functionality that is specific to only managers or only technicians

Figure 26–2 shows what the business domain layer would look like if you created distinct `User`, `Manager`, and `Technician` entity objects to allow distinguishing the different kinds of business information in a more formal way inside your application. Since technicians and managers are special *kinds* of users, their corresponding entity objects would extend the base `User` entity object. This base `User` entity object contains all of the attributes and methods that are common to all types of users. The `performUserFeature()` method in the figure represents one of these common methods.

Then, for the `Manager` and `Technician` entity objects you can add specific additional attributes and methods that are unique to that kind of user. For example, in the figure, `Manager` has an additional `NextReview` attribute of type `Date` to track when the manager must next review his employees. There is also a `performManagerFeature()` method that is specific to managers. Similarly, the `Technician` entity object has an additional `Certified` attribute to track whether the technician has completed training certification. The `performTechnicianFeature()` is a method that is specific to technicians. Finally, also note that since expertise areas only are relevant for technicians, the association between "users" and expertise levels is defined between `Technician` and `ExpertiseArea`.

Figure 26–2 Distinguishing Users, Managers, and Technicians Using Inheritance



By modeling these different kinds of users as distinct entity objects in an inheritance hierarchy in your domain business layer, you can simplify having them share common data and behavior and implement the aspects of the application that make them distinct.

26.6.2 How To Create Entity Objects in an Inheritance Hierarchy

To create entity objects in an inheritance hierarchy, you use the Create Entity Object wizard to create each entity following the steps outlined in the sections below. The example described here assumes that you've altered the SRDemo application's `USERS` table by executing the following DDL statement to add two new columns to it:

```
alter table users add (
  certified varchar2(1),
  next_review date
);
```

26.6.2.1 Start By Identifying the Discriminator Column and Distinct Values

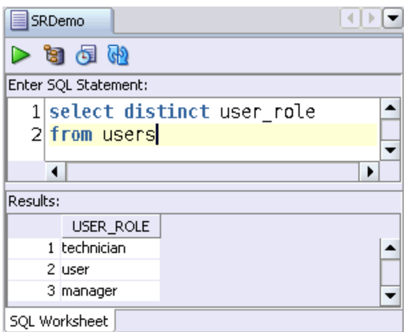
Before creating entity objects in an inheritance hierarchy based on table containing different kinds of information, you should first identify which column in the table is used to distinguish the kind of row it is. In the SRDemo application's `USERS` table, this is the `USER_ROLE` column. Since it helps partition or "discriminate" the rows in the table into separate groups, this column is known as the discriminator column.

Next, determine the valid values that the descriminator column takes on in your table. You might know this off the top of your head, or you could execute a simple SQL statement in the JDeveloper **SQL Worksheet** to determine the answer. To access the worksheet:

- Choose **View | Connection Navigator**.
- Expand the **Database** folder and select the `SRDemo` connection.
- Choose **SQL Worksheet** from the right-mouse context menu.

Figure 26–3 shows the results of performing a `SELECT DISTINCT` query in the **SQL Worksheet** on the `USER_ROLE` column in the `USERS` table. It confirms that the rows are partitioned into three groups based on the `USER_ROLE` discriminator values: `user`, `technician`, and `manager`.

Figure 26–3 Using the SQL Worksheet to Find Distinct Discriminator Column Values



26.6.2.2 Identify the Subset of Attributes Relevant to Each Kind of Entity

Once you know how many different kinds of business entities are stored in the table, you will also know how many entity objects to create to model these distinct items. You'll typically create one entity object per kind of item. Next, in order to help determine which entity should act as the base of the hierarchy, you need to determine which subset of attributes is relevant to each kind of item.

Using the example above, assume you determine that all of the attributes except `Certified` and `NextReview` are relevant to all users, that `Certified` is specific to technicians, and that `NextReview` is specific to managers. This information leads you to determine that the `Users` entity object should be the base of the hierarchy, with `Manager` and `Technician` entity object each extending `Users` to add their specific attributes.

26.6.2.3 Creating the Base Entity Object in an Inheritance Hierarchy

To create the base entity object in an inheritance hierarchy, use the Create Entity Object wizard and following these steps:

- In step 1 on the **Name** panel, provide a name and package for the entity, and select the schema object on which the entity will be based.

  For example, name the entity object `User` and base it on the `USERS` table.

- In step 2 on the **Attributes** panel, select the attributes in the **Entity Attributes** list that are not relevant to the base entity object (if any) and click **Remove** to remove them.

  For example, remove the `Certified` and `NextReview` attributes from the list.

- In step 3 on the **Attribute Settings** panel, use the **Select Attribute** dropdown list to choose the attribute that will act as the discriminator for the family of inherited entity objects and check the **Discriminator** checkbox to identify it as such. Importantly, you must also supply a **Default Value** for this discriminator attribute to identify rows of this base entity type.

  For example, select the `UserRole` attribute, mark it as a discriminator attribute, and set its **Default Value** to the value "`user`".

> **Note:** Leaving the **Default Value** blank for a discriminator attribute is legal. A blank default value means that a row whose discriminator column value `IS NULL` will be treated as this base entity type.

Then click **Finish** to create the entity object.

26.6.2.4 Creating a Subtype Entity Object in an Inheritance Hierarchy

To create a subtype entity object in an inheritance hierarchy, first do the following:

- Determine the entity object that will be the parent entity object from which your new entity object will extend.

  For example, the parent entity for a new `Manager` entity object will be the `User` entity created above.

- Ensure that the parent entity has a discriminator attribute already identified.

  The base type must already have the discriminator attribute identified as described in the section above. If it does not, use the Entity Object editor to set the **Discriminator** property on the appropriate attribute of the parent entity before creating the inherited child.

Then, use the Create Entity Object wizard and follow these steps to create the new subtype entity object in the hierarchy:

- In step 1 on the **Name** panel, provide a name and package for the entity, and click the **Browse** button next to the **Extends** field to select the parent entity from which the entity being created will extend.

  For example, name the new entity `Manager` and select the `User` entity object in the **Extends** field.

- In step 2 on the **Attributes** panel, use the **New from Table** button to add the attributes corresponding to the underlying table columns that are specific to this new entity subtype.

  For example, select the `NEXT_REVIEW` column to add a corresponding `NextReview` attribute to the `Manager` entity object

- Still on step 2, use the **Override** button to override the discriminator attribute so that you can customize the attribute metadata to supply a distinct **Default Value** for the `Manager` subtype.

  For example, override the `UserRole` `NextReview` attribute.

- In step 3 on the **Attribute Settings** panel, use the **Select Attribute** dropdown list to choose the discriminator attribute. Importantly, you must change the **Default Value** field to supply a distinct default value for the discriminator attribute that defines the entity subtype being created.

  For example, select the `UserRole` attribute and change its **Default Value** to the value "`manager`".

Then click **Finish** to create the subtype entity object.

> **Note:** You can repeat the same steps to define the `Technician` entity that extends `User` to add the additional `Certified` attribute and overrides the **Default Value** of the `UserRole` discriminator attribute to have the value "`technician`".

26.6.3 How to Add Methods to Entity Objects in an Inheritance Hierarchy

To add methods to entity objects in an inheritance hierarchy, enable the custom Java class for the entity in question and visit the code editor to add the method.

26.6.3.1 Adding Methods Common to All Entity Objects in the Hierarchy

To add a method that is common to all entity objects in the hierarchy, enable a custom Java class for the base entity object in the hierarchy and add the method in the code editor. For example, if you add the following method to the `UserImpl` class for the base `User` entity object, it will be inherited by all entity objects in the hierarchy:

```
// In UserImpl.java
public void performUserFeature() {
  System.out.println("## performUserFeature as User");
}
```

26.6.3.2 Overriding Common Methods in a Subtype Entity

To override a method in a subtype entity that is common to all entity objects in the hierarchy, enable a custom Java class for the subtype entity and choose **Source | Override Methods** to launch the **Override Methods** dialog. Select the method you want to override, and click **OK**. Then, customize the overridden method's implementation in the code editor. For example, imagine overriding the `performUserFeature()` method in the `ManagerImpl` class for the `Manager` subtype entity object and change the implementation to look like this:

```
// In ManagerImpl.java
public void performUserFeature() {
  System.out.println("## performUserFeature as Manager");
}
```

When working with instances of entity objects in a subtype hierarchy, sometimes you will process instances of multiple different subtypes. Since `Manager` and `Technician` entities are special kinds of `User`, you can write code that works with all of them using the more generic `UserImpl` type that they all have in common. When doing this generic kind of processing of classes that might be one of a *family* of subtypes in a hierarchy, Java will always invoke the most specific override of a method available.

This means that invoking the `performUserFeature()` method on an instance of `UserImpl` that happens to really be the more specific `ManagerImpl` subtype, will the result in printing out the following:

```
## performUserFeature as Manager
```

instead of the default result that regular `UserImpl` instances would get:

```
## performUserFeature as User
```

26.6.3.3 Adding Methods Specific to a Subtype Entity

To add a method that is specific to a subtype entity object in the hierarchy, enable a custom Java class for that entity and add the method in the code editor. For example, you could add the following method to the `ManagerImpl` class for the `Manager` subtype entity object:

```
// In ManagerImpl.java
public void performManagerFeature() {
  System.out.println("## performManagerFeature called");
}
```

26.6.4 What You May Need to Know

26.6.4.1 Sometimes You Need to Introduce a New Base Entity

In the example above, the `User` entity object corresponded to a concrete kind of row in the `USERS` table and it also played the role of the base entity in the hierarchy. In other words, all of its attributes were common to all entity objects in the hierarchy. You might wonder what would happen, however, if the `User` entity required a property that was specific to users, but not common to managers or technicians. Imagine that end-users can participate in customer satisfaction surveys, but that managers and technicians do not. The `User` entity would require a `LastSurveyDate` attribute to handle this requirement, but it wouldn't make sense to have `Manager` and `Technician` entity objects inherit it.

In this case, you can introduce a new entity object called `BaseUser` to act as the base entity in the hierarchy. It would have all of the attributes common to all `User`, `Manager`, and `Technician` entity objects. Then each of the three entities the correspond to concrete rows that appear in the table could have some attributes that are inherited from `BaseUser` and some that are specific to its subtype. In the `BaseUser` type, so long as you mark the `UserRole` attribute as a discriminator attribute, you can just leave the **Default Value** blank (or some other value that does *not* occur in the `USER_ROLE` column in the table). Because at runtime you'll never be using instances of the `BaseUser` entity, it doesn't really matter what its discriminator default value is.

26.6.4.2 Finding Subtype Entities by Primary Key

When you use the `findByPrimaryKey()` method on an entity definition, it only searches the entity cache for the entity object type on which you call it. In the example above, this means that if you call `UserImpl.getDefinitionObject()` to access the entity definition for the `User` entity object when you call `findByPrimaryKey()` on it, you will only find entities in the cache that happen to be users. Sometimes this is exactly the behavior you want. However, if you want to find an entity object by primary key allowing the possibility that it might be a subtype in an inheritance hierarchy, then you can use the `EntityDefImpl` class' `findByPKExtended()` method instead. In the `User` example described here, this alternative finder method would find an entity object by primary key whether it is a `User`, `Manager`, or `Technician`. You can then use the Java `instanceof` operator to test which type you found, and then cast the `UserImpl` object to the more specific entity object type in order to work with features specific to that subtype.

26.6.4.3 You Can Create View Objects with Polymorphic Entity Usages

When you create an entity-based view object with an entity usage corresponding to a base entity object in an inheritance hierarchy, you can configure the view object to query rows corresponding to multiple different subtypes in the base entity's subtype hierarchy. Each row in the view object will use the appropriate subtype entity object as the entity row part, based on matching the value of the discriminator attribute. See Section 27.6.2, "How To Create a View Object with a Polymorphic Entity Usage" for specific instructions on setting up and using these view objects.

26.7 Controlling Entity Posting Order to Avoid Constraint Violations

Due to database constraints, when you perform DML operations to save changes to a number of related entity objects in the same transaction, the order in which the operations are performed can be significant. If you try to insert a new row containing foreign key references *before* inserting the row being referenced, the database can complain with a constraint violation. This section helps you understand the default order for processing of entity objects during commit time and how to programmatically influence that order when necessary.

> **Note:** The examples in this section refer to the `ControllingPostingOrder` project in the `AdvancedEntityExamples` workspace. See the note at the beginning of this chapter for download instructions.

26.7.1 Understanding the Default Post Processing Order

By default, when you commit the transaction the entity objects in the pending changes list are processed in chronological order, in other words, the order in which the entities were added to the list. This means that, for example, if you create a new `ServiceRequest` and then a new `Product` related to that service request, the new `ServiceRequest` will be inserted first and the new `Product` second.

26.7.2 How Compositions Change the Default Processing Ordering

When two entity objects are related by a *composition*, the strict chronological ordering is modified automatically to ensure that composed parent and child entity rows are saved in an order that avoids violating any constraints. This means, for example, that a new parent entity row is inserted before any new composed children entity rows.

26.7.3 Overriding postChanges() to Control Post Order

If your related entities are associated but *not* composed, then you need to write a bit of code to ensure that the related entities get saved in the appropriate order.

26.7.3.1 Observing the Post Ordering Problem First Hand

Consider the `newServiceRequestForNewProduct()` custom method from an `ExampleModule` application module in Example 26–13. It accepts a set of parameters and:

1. Creates a new `ServiceRequest`.
2. Creates a new `Product`.
3. Sets the product id to which the server request pertains.
4. Commits the transaction.
5. Constructs a `Result` Java bean to hold new product ID and service request ID.
6. Results the result.

> **Note:** The code makes the assumption that both `ServiceRequest.SvrId` and `Product.ProdId` have been set to have DBSequence data type to populate their primary keys based on a sequence.

Example 26–13 Creating a New ServiceRequest Then a New Product and Returning the New Ids

```
// In ExampleModuleImpl.java
public Result newServiceRequestForNewProduct(String prodName,
                                             String prodDesc,
                                             String problemDesc,
                                             Number customerId) {
  // 1. Create a new ServiceRequest
  ServiceRequestImpl newSR = createNewServiceRequest();
  // 2. Create a new Product
  ProductImpl newProd = createNewProduct();
  newProd.setName(prodName);
  newProd.setDescription(prodDesc);
  // 3. Set the product id to which service request pertains
  newSR.setProdId(newProd.getProdId().getSequenceNumber());
  newSR.setProblemDescription(problemDesc);
  newSR.setCreatedBy(customerId);
  // 4. Commit the transaction
  getDBTransaction().commit();
  // 5. Construct a bean to hold new product id and SR id
  Result result = new Result();
  result.setSvrId(newSR.getSvrId().getSequenceNumber());
  result.setProdId(newProd.getProdId().getSequenceNumber());
  // 6. Return the result
  return result;
}
private ServiceRequestImpl createNewServiceRequest() {
  EntityDefImpl srDef   = ServiceRequestImpl.getDefinitionObject();
  return (ServiceRequestImpl)srDef.createInstance2(getDBTransaction(),null);
}
private ProductImpl createNewProduct() {
  EntityDefImpl srDef   = ProductImpl.getDefinitionObject();
  return (ProductImpl)srDef.createInstance2(getDBTransaction(),null);
}
```

If you add this method to the application module's client interface and test it from a test client program, you get an error:

```
oracle.jbo.DMLConstraintException:
JBO-26048: Constraint "SVR_PRD_FK" violated during post operation:
"Insert" using SQL Statement
"BEGIN
  INSERT INTO SERVICE_REQUESTS(
    SVR_ID,STATUS,REQUEST_DATE,
    PROBLEM_DESCRIPTION,PROD_ID,CREATED_BY)
   VALUES (?,?,?,?,?,?)
   RETURNING SVR_ID INTO ?;
END;".
## Detail 0 ##
java.sql.SQLException:
ORA-02291: integrity constraint (SRDEMO.SVR_PRD_FK) violated
           - parent key not found
```

The database complains when the SERVICE_REQUESTS row is inserted that the value of its PROD_ID foreign key doesn't correspond to any row in the PRODUCTS table. This occurred because:

- The code created the ServiceRequest before the Product
- ServiceRequest and Product entity objects are associated but not composed
- The DML operations to save the new entity rows is done in chronological order, so the new ServiceRequest gets inserted before the new Product.

26.7.3.2 Forcing the Product to Post Before the ServiceRequest

To remedy the problem, you could reorder the lines of code in the example to create the Product first, then the ServiceRequest. While this would address the immediate problem, it still leaves the chance that another application developer could creating things in an incorrect order.

The better solution is to make the entity objects *themselves* handle the posting order so it will work correctly regardless of the order of creation. To do this you need to override the postChanges() method in the entity that contains the foreign key attribute referencing the associated entity object and write code as shown in Example 26–14. In this example, since it is the ServiceRequst that contains the foreign key to the Product entity, you need to update the ServiceRequest to conditionally force a related, new Product to post before the service request posts itself.

The code tests whether the entity being posted is in the STATUS_NEW or STATUS_MODIFIED state. If it is, it retrieves the related product using the getProduct() association accessor. If the related Product also has a post-state of STATUS_NEW, then *first* it calls postChanges() on the related parent row before calling super.postChanges() to perform its own DML.

Example 26–14 Overriding postChanges() in ServiceRequestImpl to Post Product First

```
// In ServiceRequestImpl.java
public void postChanges(TransactionEvent e) {
  /* If current entity is new or modified */
  if (getPostState() == STATUS_NEW ||
      getPostState() == STATUS_MODIFIED) {
    /* Get the associated product for the service request */
    ProductImpl product = getProduct();
    /* If there is an associated product */
    if (product != null) {
      /* And if it's post-status is NEW */
      if (product.getPostState() == STATUS_NEW) {
        /*
         * Post the product first, before posting this
         * entity by calling super below
         */
        product.postChanges(e);
      }
    }
  }
  super.postChanges(e);
}
```

If you were to re-run the example now, you would see that without changing the creation order in the `newServiceRequestForNewProduct()` method's code, entities now post in the correct order — first new `Product`, then new `ServiceRequest`. Yet, there is still a problem. The constraint violation still appears, but now for a different reason!

If the primary key for the `Product` entity object were user-assigned, then the code in Example 26–14 would be all that is required to address the constraint violation by correcting the post ordering.

Note: An alternative to the programmatic technique discussed above, which solves the problem at the J2EE application layer, is the use of deferrable constraints at the database layer. If you have control over your database schema, consider defining (or altering) your foreign key constraints to be `DEFERRABLE INITIALLY DEFERRED`. This causes the database to defer checking the constraint until transaction commit time. This allows the application to perform DML operations in any order provided that by `COMMIT` time all appropriate related rows have been saved and would alleviate the parent/child ordering described above. However, you would still need to write the code described in the following sections to cascade-update the foreign key values if the parent's primary key is assigned from a sequence.

In this example, however, the `Product.ProdId` is assigned from a database sequence, and not user-assigned in this example. So when a new `Product` entity row gets posted its `ProdId` attribute is refreshed to reflect the database-assigned sequence value. The foreign key value in the `ServiceRequest.ProdId` attribute referencing the new product is "orphaned" by this refreshing of the product's ID value. When the service request row is saved, its `PROD_ID` value still doesn't match a row in the `PRODUCTS` table, and the constraint violation occurs again. The next two sections discuss the solution to address this "orphaning" problem.

26.7.3.3 Understanding Associations Based on DBSequence-Valued Primary Keys

Recall from Section 6.6.3.8, "Trigger-Assigned Primary Key Values from a Database Sequence" that when an entity object's primary key attribute is of `DBSequence` type, during the transaction in which it is created, its numerical value is a unique, temporary negative number. If you create a number of associated entities in the same transaction, the relationships between them are based on this temporary negative key value. When the entity objects with `DBSequence`-value primary keys are posted, their primary key is refreshed to reflect the correct database-assigned sequence number, leaving the associated entities that are still holding onto the temporary negative foreign key value "orphaned".

For entity objects based on a *composition*, when the parent entity object's `DBSequence`-valued primary key is refreshed, the composed children entity rows automatically have their temporary negative foreign key value updated to reflect the owning parent's refreshed, database-assigned primary key. This means that for composed entities, the "orphaning" problem does not occur.

However, when entity objects are related by an association that is not a composition, you need to write a little code to insure that related entity rows referencing the temporary negative number get updated to have the refreshed, database-assigned primary key value. The next section outlines the code required.

26.7.3.4 Refreshing References to DBSequence-Assigned Foreign Keys

When an entity like `Product` in this example has a `DBSequence`-valued primary key, and it is referenced as a foreign key by other entities that are associated with (but not composed by) it, you need to override the `postChanges()` method as shown in Example 26–15 to save a reference to the row set of entity rows that might be referencing this new `Product` row. If the status of the current `Product` row is New, then the code assigns the `RowSet`-valued return of the `getServiceRequest()` association accessor to the newServiceRequestsBeforePost member field before calling `super.postChanges()`.

Example 26–15 Saving Reference to Entity Rows Referencing this New Product

```
// In ProductImpl.java
RowSet newServiceRequestsBeforePost = null;
public void postChanges(TransactionEvent TransactionEvent) {
  /* Only bother to update references if Product is a NEW one */
  if (getPostState() == STATUS_NEW) {
    /*
     * Get a rowset of service requests related
     * to this new product before calling super
     */
    newServiceRequestsBeforePost = (RowSet)getServiceRequest();
  }
  super.postChanges(TransactionEvent);
}
```

This saved `RowSet` is then used by the overridden `refreshFKInNewContainees()` method shown in Example 26–16. It gets called to allow a new entity row to cascade update its refreshed primary key value to any other entity rows that were referencing it before the call to `postChanges()`. It iterates over the `ServiceRequestImpl` rows in the newServiceRequestsBeforePost row set (if non-null) and sets the new product ID value of each one to the new sequence-assigned product value of the newly-posted `Product` entity.

Example 26–16 Cascade Updating Referencing Entity Rows with New ProdId Value

```
// In ProductImpl.java
protected void refreshFKInNewContainees() {
  if (newServiceRequestsBeforePost != null) {
    Number newProdId = getProdId().getSequenceNumber();
    /*
     * Process the rowset of service requests that referenced
     * the new product prior to posting, and update their
     * ProdId attribute to reflect the refreshed ProdId value
     * that was assigned by a database sequence during posting.
     */
    while (newServiceRequestsBeforePost.hasNext()) {
      ServiceRequestImpl svrReq =
        (ServiceRequestImpl)newServiceRequestsBeforePost.next();
      svrReq.setProdId(newProdId);
    }
    closeNewServiceRequestRowSet();
  }
}
```

After implementing this change, the code in Example 26–13 runs without encountering any database constraint violations.

26.8 Implementing Automatic Attribute Recalculation

Section 6.10, "Adding Transient and Calculated Attributes to an Entity Object" explained how to add calculated attributes to an entity object. Often the formula for the calculated value will depend on other attribute values in the entity. For example, consider a `LineItem` entity object representing the line item of an order. The `LineItem` might have attributes like `Price` and `Quantity`. You might introduce a calculated attributed named `ExtendedTotal` which you calculate by multiplying the price times the quantity. When either the `Price` or `Quantity` attributes is modified, you might expect the calculated attribute `ExtendedTotal` to be updated to reflect the new extended total, but this does not happen automatically. Unlike a spreadsheet, the entity object does not have any built-in expression evaluation engine that understands what attributes your formula depends on.

To address this limitation, you can write code in a framework extension class for entity objects that add a recalculation facility. The `SREntityImpl` framework extension class in the SRDemo application contains the code shown in Example 26–17 that does this. It does not try to implement a sophisticated expression evaluator. Instead, it leverages the custom properties mechanism to allow a developer to supply a declarative "hint" about which attributes (e.g. `X`, `Y`, and `Z`) should be recalculated when another attribute like `A` gets changed.

To leverage the generic facility, the developer of an entity object:

- Bases his entity on the framework extension class containing this additional code,
- Defines one or more entity-level custom properties that follow a particular naming pattern. These indicate to the generic code which attributes should get recalculated when a particular other attribute changes.

  To indicate that "when attribute `A` changes, recalculate attributes `X`, `Y`, and `Z`" he would add a custom property named `Recalc_A` with the comma-separated value "`X,Y,Z`" to indicate that.

To implement the functionality the `SREntityImpl` class overrides the `notifyAttributesChanged()` method. This method gets invoked whenever the value of entity object attributes change. As arguments, the method receives two arrays:

- `int[]` of attribute index numbers whose values have changed
- `Object[]` containing the new values for those attributes

The code does the following basic steps:

1. Iterates over the set of custom entity properties
2. If property name starts with "`Recalc_`" it gets the substring following this prefix to know the name of the attribute whose change should trigger recalculation of others.
3. Determines the index of the recalc-triggering attribute.
4. If the array of changed attribute indexes includes the index of the recalc-triggering attribute, then tokenize the comma-separated value of the property to find the names of the attributes to recalculate.
5. If there were any attributes to recalculate, add their attribute indexes to a new `int[]` of attributes whose values have changed.

   The new array is created by copying the existing array elements in the attrIndices array to a new array, then adding in the additional attribute index numbers.
6. Call the `super` with the possibly updated array of changed attributes.

Example 26–17 Entity Framework Extension Code to Automatically Recalculate Derived Attributes

```
// In SREntityImpl.java
protected void notifyAttributesChanged(int[] attrIndices, Object[] values) {
  int attrIndexCount = attrIndices.length;
  EntityDefImpl def = getEntityDef();
  HashMap eoProps = def.getPropertiesMap();
  if (eoProps != null && eoProps.size() > 0) {
    Iterator iter = eoProps.keySet().iterator();
    ArrayList otherAttrIndices = null;
    // 1. Iterate over the set of custom entity properties
    while (iter.hasNext()) {
      String curPropName = (String)iter.next();
      if (curPropName.startsWith(RECALC_PREFIX)) {
        // 2. If property name starts with "Recalc_" get follow attr name
        String changingAttrNameToCheck = curPropName.substring(PREFIX_LENGTH);
        // 3. Get the index of the recalc-triggering attribute
        int changingAttrIndexToCheck =
            def.findAttributeDef(changingAttrNameToCheck).getIndex();
        if (isAttrIndexInList(changingAttrIndexToCheck,attrIndices)) {
          // 4. If list of changed attrs includes recalc-triggering attr,
          //    then tokenize the comma-separated value of the property
          //    to find the names of the attributes to recalculate
          String curPropValue = (String)eoProps.get(curPropName);
          StringTokenizer st = new StringTokenizer(curPropValue,",");
          if (otherAttrIndices == null) {
            otherAttrIndices = new ArrayList();
          }
          while (st.hasMoreTokens()) {
            String attrName = st.nextToken();
            int attrIndex = def.findAttributeDef(attrName).getIndex();
            if (!isAttrIndexInList(attrIndex,attrIndices)) {
              Integer intAttr = new Integer(attrIndex);
              if (!otherAttrIndices.contains(intAttr)) {
                otherAttrIndices.add(intAttr);
              }
            }
          }
        }
      }
    }
    if (otherAttrIndices != null && otherAttrIndices.size() > 0) {
      // 5. If there were any attributes to recalculate, add their attribute
      //    indexes to the int[] of attributes whose values have changed
      int extraAttrsToAdd = otherAttrIndices.size();
      int[] newAttrIndices = new int[attrIndexCount + extraAttrsToAdd];
      Object[] newValues = new Object[attrIndexCount + extraAttrsToAdd];
      System.arraycopy(attrIndices,0,newAttrIndices,0,attrIndexCount);
      System.arraycopy(values,0,newValues,0,attrIndexCount);
      for (int z = 0; z < extraAttrsToAdd; z++) {
        newAttrIndices[attrIndexCount+z] =
         ((Integer)otherAttrIndices.get(z)).intValue();
        newValues[attrIndexCount+z] =
         getAttribute((Integer)otherAttrIndices.get(z));
      }
      attrIndices = newAttrIndices;
      values = newValues;
    }
  }
```

```
  // 6. Call the super with the possibly updated array of changed attributes
  super.notifyAttributesChanged(attrIndices, values);
}
```

The `ServiceHistory` entity object in the SRDemo application uses this feature by setting a custom entity property named `Recalc_SvhType` with the value of `Hidden`. This way, anytime the value of the SvhType attribute is changed, the value of the calculated `Hidden` attribute is recalculated.

26.9 Implementing Custom Validation Rules

ADF Business Components comes with a base set of built-in declarative validation rules that you can use. However, the most powerful feature of the validator architecture for entity objects is that you can create your own custom validation rules. When you notice that you or your team are writing the same kind of validation code over and over, you can build a custom validation rule class that captures this common validation "pattern" in a parameterized way. Once you've defined a custom validation rule class, you can register it in JDeveloper so that it is as simple to use as any of the built-in rules. In fact, as you see in the following sections, you can even bundle your custom validation rule with a custom UI panel that JDeveloper will leverage automatically to facilitate developers' using and configuring the parameters your validation rule might require.

26.9.1 How To Create a Custom Validation Rule

To write a custom validation rule for entity objects, create a Java class that implements the `JboValidatorInterface` in the `oracle.jbo.rules` package. As shown in Example 26–18, this interface contains one main `validate()` method, and a getter and setter method for a `Description` property.

Example 26–18 All Validation Rules Must Implement the JboValidatorInterface

```
package oracle.jbo.rules;
public interface JboValidatorInterface {
void validate(JboValidatorContext valCtx) { }
java.lang.String getDescription() { }
void setDescription(String description) { }
}
```

If the behavior of your validation rule will be parameterized to make it more flexible, then add additional bean properties to your validator class for each parameter. For example, the SRDemo application contains a custom validation rule called `DateMustComeAfterRule` which validates that one date attribute must come after another date attribute. To allow developer's using the rule to configure the names of the date attributes to use as the initial and later dates for validation, this class defines two properties `initialDateAttrName` and `laterDateAttrName`.

Example 26–19 shows the code that implements the custom validation rule. It extends the `AbstractValidator` to inherit support for working automatically with the entity object's custom message bundle, where JDeveloper will automatically save the validation error message when a developer uses the rule on one of their entity objects.

The `validate()` method of the validation rule gets invoked at runtime whenever the rule class should perform its functionality. The code performs the following basic steps:

1. Ensures validator is correctly attached at the entity level.
2. Gets the entity row being validated.
3. Gets the values of the initial and later date attributes.
4. Validate sthat initial date is before later date.
5. Throws an exception if the validation fails.

Example 26–19 Custom DateMustComeAfterRule in the SRDemo Application

```
package oracle.srdemo.model.frameworkExt.rules;
// NOTE: Imports omitted
public class DateMustComeAfterRule extends AbstractValidator
       implements JboValidatorInterface {
  /**
   * This method is invoked by the framework when the
   * validator should do its job.
   */
  public void validate(JboValidatorContext valCtx) {
    // 1. If validator is correctly attached at the entity level...
    if (validatorAttachedAtEntityLevel(valCtx)) {
      // 2. Get the entity row being validated
      EntityImpl eo = (EntityImpl)valCtx.getSource();
      // 3. Get the values of the initial and later date attributes
      Date initialDate = (Date) eo.getAttribute(getInitialDateAttrName());
      Date laterDate = (Date) eo.getAttribute(getLaterDateAttrName());
      // 4. Validate that initial date is before later date
      if (!validateValue(initialDate,laterDate)) {
        // 5. Throw the validation exception
        RulesBeanUtils.raiseException(getErrorMessageClass(),
                                      getErrorMsgId(),
                                      valCtx.getSource(),
                                      valCtx.getSourceType(),
                                      valCtx.getSourceFullName(),
                                      valCtx.getAttributeDef(),
                                      valCtx.getNewValue(),
                                      null, null);
      }
    }
    else {
      throw new RuntimeException("Rule must be at entity level");
    }
  }
  /**
   * Validate that the initialDate comes before the laterDate.
   */
  private boolean validateValue(Date initialDate, Date laterDate) {
    return (initialDate == null) || (laterDate == null) ||
    (initialDate.compareTo(laterDate) < 0);
  }
  /**
   * Return true if validator is attached to entity object
   * level at runtime.
   */
```

```
  private boolean validatorAttachedAtEntityLevel(JboValidatorContext ctx) {
    return ctx.getOldValue() instanceof EntityImpl;
  }
  // NOTE: Getter/Setter Methods omitted
  private String description;
  private String initialDateAttrName;
  private String laterDateAttrName;
}
```

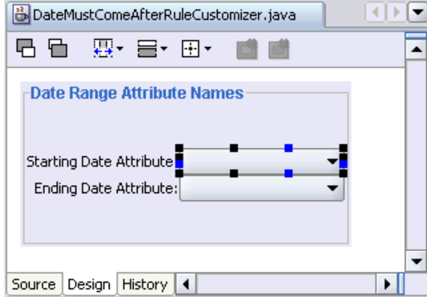
For easier reuse of your custom validation rules, you would typically package them into a JAR file for reference by applications that make use of the rules. In the SRDemo application, the `FrameworkExtensions` project contains a DateMustComeAfterRule.deploy deployment profile that packages the rule class into a JAR file named `DateMustComeAfterRule.jar` for use at runtime and design time.

26.9.2 Adding a Design Time Bean Customizer for Your Rule

Since a validation rule class is a bean, you can implement a standard JavaBean customizer class to improve the design time experience of setting the bean properties. In the example of the `DateMustComeAfter` rule in the previous section, the two properties developers will need to configure are the `initialDateAttrName` and `laterDateAttrName` properties.

Figure 26–4 illustrates using JDeveloper's visual designer for Swing to create a `DateMustComeAfterRuleCustomizer` using a `JPanel` with a titled border containing two JLabel prompts and two `JComboBox` controls for the dropdown lists. The code in the class populates the dropdown lists with the names of the Date-valued attributes of the current entity object being edited in the IDE. This will allow a developer who adds a `DateMustComeAfterRule` validation to their entity object to easily pick which date attributes should be used for the starting and ending dates for validation.

Figure 26–4 Using JDeveloper's Swing Visual Designer to Create Validation Rule Customizer



To associate a customizer with your `DateMustComeAfterRule` Java Bean, you follow the standard practice of creating a `BeanInfo` class. As shown in Example 26–20, the `DateMustComeAfterRuleBeanInfo` returns a BeanDescriptor that associates the customizer class with the `DateMustComeAfter` bean class.

You would typically package your customizer class and this bean info in a separate JAR file for design-time-only use. The `FrameworkExtensions` project in the SRDemo application contains a deployment profile that packages these classes in a `DateMustComeAfterRuleDT.jar`.

Example 26–20 BeanInfo to Associate a Customizer with a Custom Validation Rule

```
package oracle.srdemo.model.frameworkExt.rules;
import java.beans.BeanDescriptor;
import java.beans.SimpleBeanInfo;
public class DateMustComeAfterRuleBeanInfo extends SimpleBeanInfo {
  public BeanDescriptor getBeanDescriptor() {
    return new BeanDescriptor(DateMustComeAfterRule.class,
                              DateMustComeAfterRuleCustomizer.class);
  }
}
```

26.9.3 Registering and Using a Custom Rule in JDeveloper

To use a custom validation rule in a project containing entity objects, follow these steps:

1. Define a project-level library for the rule JAR files.
2. Add that library to your project's library list.
3. Use the **Business Components > Registered Rules** panel of the **Project Properties** dialog to add a one or more validation rules.

   When adding a validation rule, provide the fully-qualified name of the validation rule class, and supply a validation rule name that will appear in JDeveloper's list of available validators.

Figure 26–5 shows the **Validation** panel of the Entity Object editor for the SRDemo application's `ServiceRequest` entity object. When you edit the DateMustComeAfter rule, you can see the custom editing panel is automatically discovered from the rule class' BeanInfo and used at design time to show the developer the starting and ending attribute names. JDeveloper provides the support for capturing the translatable error message that will be shown to the end-user if the validation rule fails at runtime.

Figure 26–5 Custom Validation Rule with Custom Editor Panel in JDeveloper

27

Advanced View Object Techniques

This chapter describes advanced techniques you can use while designing and working with your view objects.

This chapter includes the following sections:

- Section 27.1, "Advanced View Object Concepts and Features"
- Section 27.2, "Tuning Your View Objects for Best Performance"
- Section 27.3, "Using Expert Mode for Full Control Over SQL Query"
- Section 27.4, "Working with Multiple Named View Criteria"
- Section 27.5, "Performing In-Memory Sorting and Filtering of Row Sets"
- Section 27.6, "Using View Objects to Work with Multiple Row Types"
- Section 27.7, "Reading and Writing XML"
- Section 27.8, "Using Programmatic View Objects for Alternative Data Sources"
- Section 27.9, "Creating a View Object with Multiple Updatable Entities"
- Section 27.10, "Declaratively Preventing Insert, Update, and Delete"

Note: To experiment with a working version of the examples in this chapter, download the `AdvancedViewObjectsExamples` workspace from *Example Downloads* page at `http://otn.oracle.com/documentation/jdev/b25947_01/`.

27.1 Advanced View Object Concepts and Features

This section describes a number of interesting view object concepts and features that have not been discussed in previous chapters.

27.1.1 Using a Max Fetch Size to Only Fetch the First N Rows

The default maximum fetch size of a view object is minus one (-1), which indicates that there is no artificial limit to the number of rows that can be fetched. Keep in mind that by default, rows are fetched as needed, so this default does not imply a view object will necessary fetch all the rows. It simply means that if you attempt to iterate through all the rows in the query result, you will get them all.

However, you might want to put an upper bound on the maximum number of rows that a view object will retrieve. If you write a query containing an `ORDER BY` clause and only want to return the first `N` rows to display the "Top-N" entries in a page, you can call the `setMaxFetchSize()` method on your view object to set the maximum fetch size to `N`. The view object will stop fetching rows when it hits the maximum fetch size. Often you will combine this technique with specifying a **Query Optimizer Hint** of `FIRST_ROWS` on the **Tuning** panel of the View Object Editor. This gives a hint to the database that you want to retrieve the first rows as quickly as possible, rather than trying to optimize the retrieval of all rows.

27.1.2 Consistently Displaying New Rows in View Objects Based on the Same Entity

When multiple instances of entity-based view objects in an application module are based on the same underlying entity object, a new row created in one of them can be automatically added (without having to re-query) to the row sets of the others to keep your user interface consistent or simply to consistently reflect new rows in different application pages for a pending transaction. Consider the SRDemo application's `SRList` page that displays an end-user's list of service requests. If the end-user goes to create a new service request, this task is performed through a different view object and handled by a custom application module method. Using this view object new row consistency feature, the newly created service request automatically appears in the end-user's list of open service requests on the `SRList` page without having to re-query the database.

For historical reasons, this capability is known as the *view link consistency* feature because in prior releases of Oracle ADF the addition of new rows to other relevant row sets only was supported for detail view object instances in a view link based on an association. Now this view link consistency feature works for any view objects for which it is enabled, regardless of whether they are involved in a view link or not.

27.1.2.1 How View Link Consistency Mode Works

Consider two entity-based view objects `ServiceRequestSummary` and `ServiceRequests` both based on the same underlying `ServiceRequest` entity object. When a new row is created in a row set for one of these view objects (like `ServiceRequests`) and the row's primary key is set, any of the *other* row sets for view objects based on the same `ServiceRequest` entity object (like `ServiceRequestSummary`) receive an event indicating a new row has been created. If their view link consistency flag is enabled, then a copy of the new row is inserted into their row set as well.

27.1.2.2 Understanding the Default View Link Consistency Setting and How to Change It

You can control the default setting for the view link consistency feature using the `jbo.viewlink.consistent` configuration parameter. The default setting for this parameter is the word "`DEFAULT`" which has the following meaning. If your view object has:

- A single entity usage, view link consistency is enabled
- Multiple entity usages, and:
  - If all secondary entity usages are marked as contributing reference information, then view link consistency is enabled
  - If any secondary entity usage marked as **not** being a reference view link consistency is disabled.

You can globally disable this feature by setting the `jbo.viewlink.consistent` to the value `false` in your configuration. Conversely, you could globally enable this feature by setting `jbo.viewlink.consistent` to the value `true`, but Oracle does **not** recommend doing this. Doing so would force view link consistency to be set on for view objects with secondary entity usages that are not marked as a reference which presently do not support the view link consistency feature well.

To set the feature programmatically, use the `setAssociationConsistent()` API on any `RowSet`. When you call this method on a view object, it affects its default row set.

27.1.2.3 Using a RowMatch to Qualify Which New, Unposted Rows Get Added to a Row Set

If a view object has view link consistency enabled, any new row created by another view object based on the same entity object is added to its row set. By default the mechanism adds new rows in an unqualified way. If your view object has a design-time `WHERE` clause that queries only a certain subset of rows, you can apply a `RowMatch` object to your view object to perform the same filtering in-memory. The filtering expression of the `RowMatch` object you specify prevents new rows from being added that wouldn't make sense to appear in that view object.

For example, the `ServiceRequestsByStatus` view object in the SRDemo application includes a design time `WHERE` clause like this:

```
WHERE /* ... */ AND STATUS LIKE NVL(:StatusCode,'%')
```

Its custom Java class overrides the `create()` method as shown in Example 27–1 to force view link consistency to be enabled. It also applies a `RowMatch` object whose filtering expression matches rows whose `Status` attribute matches the value of the `:StatusCode` named bind variable (or matches any row if `:StatusCode` = '`%`'). This `RowMatch` filter is used by the view link consistency mechanism to qualify the row that is a candidate to add to the row set. If the row qualifies by the `RowMatch`, it is added. Otherwise, it is not.

Example 27–1 Providing a Custom RowMatch to Control Which New Rows are Added

```
// In ServiceRequestsByStatusImpl.java
protected void create() {
  super.create();
  setAssociationConsistent(true);
  setRowMatch(new RowMatch("Status = :StatusCode or :StatusCode = '%'"));
}
```

See Section 27.5.4, "Performing In-Memory Filtering with RowMatch" for more information on creating and using a `RowMatch` object.

> **Note:** If the `RowMatch` facility does not provide enough control, you can override the view object's `rowQualifies()` method to implement a custom filtering solution. Your code can determine whether a new row qualifies to be added by the view link consistency mechanism or not.

27.1.2.4 Setting a Dynamic Where Clause Disables View Link Consistency

If you call `setWhereClause()` on a view object to set a dynamic where clause, the view link consistency feature is disabled on that view object. If you have provided an appropriate custom `RowMatch` object to qualify new rows for adding to the row set, you can call `setAssociationConsistent(true)` after `setWhereClause()` to re-enable view link consistency.

27.1.2.5 New Row from Other View Objects Added at the Bottom

If a row set has view link consistency enabled, then new rows added due to creation by other row sets are added to the bottom of the row set.

27.1.2.6 New, Unposted Rows Added to Top of RowSet when Re-Executed

If a row set has view link consistency enabled, then when you call the `executeQuery()` method, any qualifying new, unposted rows are added to the top of the row set before the queried rows from the database are added.

27.1.3 Understanding View Link Accessors Versus Data Model View Link Instances

View objects support two different styles of master-detail coordination:

- View link *instances* for active data model master/detail coordination
- View link *accessor* attributes for programmatically accessing detail row sets on demand

27.1.3.1 Enabling a Dynamic Detail Row Set with Active Master/Detail Coordination

When you add a view link instance to your application module's data model, you connect two specific view object instances and indicate that you want active master/detail coordination between the two. At runtime the view link instance in the data model facilitates the eventing that enables this coordination. Whenever the current row is changed on the master view object instance, an event causes the detail view object to be refreshed by automatically invoking `executeQuery()` with a new set of bind parameters for the new current row in the master view object.

A key feature of this active data model master/detail is that the master and detail view object instances are stable objects to which client user interfaces can establish bindings. When the current row changes in the master — instead of producing a *new* detail view object instance — the existing detail view object instance updates its default row set to contain a the set of rows related to the new current master row. In addition, the user interface binding objects receive events that allow the display to update to show the detail view object's refreshed row set.

Another key feature that is exclusive to active data model master/detail is that a detail view object instance can have multiple master view object instances. For example, an `ExpertiseAreas` view object instance may be a detail of *both* a `Products` and a `Technicians` view object instances. Whenever the current row in *either* the `Products` or `Technicians` view object instance changes, the default row set of the detail `ExpertiseAreas` view object instance is refreshed to include the row of expertise area information for the current technician and the current product. See Section 27.1.6, "Setting Up a Data Model with Multiple Masters" for details on setting up a detail view object instance with multiple-masters.

27.1.3.2 Accessing a Stable Detail Row Set Using View Link Accessor Attributes

When you need to programmatically access the detail row set related to a view object row by virtue of a view link, you can use the view link accessor attribute. You control the name of the view link accessor attribute on the **View Link Properties** panel of the View Link Editor. Assuming you've named your accessor attribute *`AccessorAttrName`*, you can access the detail row set using the generic `getAttribute()` API like:

```
RowSet detail = (RowSet)currentRow.getAttribute("AccessorAttrName");
```

If you've generated a custom view row class for the master view object and exposed the getter method for the view link accessor attribute on the client view row interface, you can write strongly-typed code to access the detail row set like this:

```
RowSet detail = (RowSet)currentRow.getAccessorAttrName();
```

Unlike the active data model master/detail, programmatic access of view link accessor attributes does not require a detail view object instance in the application module's data model. Each time you invoke the view link accessor attribute, it returns a `RowSet` containing the set of detail rows related to the master row on which you invoke it.

Using the view link accessor attribute, the detail data rows are stable. As long as the attribute value(s) involved in the view link definition in the master row remain unchanged, the detail data rows will not change. Changing of the current row in the master does not affect the detail row set which is "attached" to a given master row. For this reason, in addition to being useful for general programmatic access of detail rows, view link accessor attributes are appropriate for UI object like the tree control, where data for each master node in a tree needs to retain its distinct set of detail rows.

27.1.3.3 Accessor Attributes Create Distinct Row Sets Based on an Internal View Object

When you *combine* the use of active data model master/detail with programmatic access of detail row sets using view link accessor, it is even more important to understand that they are distinct mechanisms. For example, imagine that you have:

- Defined `ServiceRequests` and `ServiceHistories` view objects
- Defined a view link between them, naming the view link accessor `HistoriesForRequest`
- Added instances of them to an application module's data model named `master` (of type `ServiceRequests`) and `detail` (of type `ServiceHistories`) coordinated actively by a view link instance.

If you find a service request in the `master` view object instance, the `detail` view object instance updates as expected to show the corresponding service request histories. At this point, if you invoke a custom method that programmatically accesses the `HistoriesForRequest` view link accessor attribute of the current `ServiceRequests` row, you get a `RowSet` containing the set of `ServiceHistory` rows. You might reasonably expect this programmatically access `RowSet` to have come from the `detail` view object instance in the data model, but this is not the case.

The `RowSet` returned by a view link accessor always originates from an *internally created* view object instance, not one you that added to the data model. This internal view object instance is created as needed and added with a system-defined name to the root application module.

The principal reason a distinct, internally-created view object instance is used is to guarantee that it remains unaffected by developer-related changes to their own view objects instances in the data model. For example, if the view row were to use the detail view object in the data model for view link accessor `RowSet`, the resulting row set could be inadvertently affected when the developer dynamically:

1. Adds a `WHERE` clause with new named bind parameters

   If such a view object instance were used for the view link accessor result, unexpected results or an error could ensue because the dynamically-added `WHERE` clause bind parameter values have not been supplied for the view link accessor's `RowSet`: they were only supplied for the default row set of the detail view object instance in the data model.

2. Adds an additional master view object instance for the detail view object instance in the data model.

   In this scenario, the semantics of the accessor would be changed. Instead of the accessor returning `ServiceHistory` rows for the current `ServiceRequest` row, it could all of a sudden start returning *only* the `ServiceHistory` rows for the current `ServiceRequest` that were created by a current technician, for example.

3. Removes the detail view object instance or its containing application module instance.

   In this scenario, all rows in the programmatically-accessed detail RowSet would become invalid.

Furthermore, Oracle ADF needs to distinguish between the active data model master/detail and view link accessor row sets for certain operations. For example, when you create a new row in a detail view object, the framework automatically populates the attributes involved in the view link with corresponding values of the master. In the active data model master/detail case, it gets these values from the current row(s) of the possibly multiple master view object instances in the data model. In the case of creating a new row in a `RowSet` returned by a view link accessor, it populates these values from the master row on which the accessor was called.

27.1.4 Presenting and Scrolling Data a Page at a Time Using the Range

To present and scroll through data a page at a time, you can configure a view object to manage for you an appropriately-sized range of rows. The range facility allows a client to easily display and update a subset of the rows in a row set, as well as easily scroll to subsequent pages *N* rows as a time. You call `setRangeSize()` to define how many rows of data should appear on each page. The default range size is one (`1`) row. A range size of minus one (`-1`) indicates the range should include all rows in the row set.

Note: When using the ADF Model layer's declarative data binding, the iterator binding in the page definition has a `RangeSize` property. At runtime, the iterator binding invokes the `setRangeSize()` method on its corresponding row set iterator, passing the value of this `RangeSize` property. The ADF design time by default sets this `RangeSize` property to `10` rows for most iterator bindings. An exception is the range size specified for a List binding to supply the set of valid values for a UI component like a dropdown list. In this case, the default range size is minus one (`-1`) to allow the range to include all rows in the row set.

When you set a range size greater than one, you control the row set paging behavior using the iterator mode. The two iterator mode flags you can pass to the `setIterMode()` method are:

- `RowIterator.ITER_MODE_LAST_PAGE_PARTIAL`

  In this mode, the last page of rows may contain fewer rows than the range size. For example, if you set the range size to 10 and your row set contains 23 rows, the third page of rows will contain only three rows. This is the style that works best for web applications.

- `RowIterator.ITER_MODE_LAST_PAGE_FULL`

  In this mode, the last page of rows is kept full, possibly including rows at the top of the page that had appeared at the bottom of the previous page. For example, if you set the range size to 10 and your row set contains 23 rows, the third page of rows will contain 10 rows, the first seven of which appeared as the last seven rows of page two. This is the style that works best for desktop-fidelity applications using Swing.

27.1.5 Efficiently Scrolling Through Large Result Sets Using Range Paging

As a general rule, for highest performance, Oracle recommends building your application in a way that avoids giving the end-user the opportunity to scroll through very large query results. To enforce this recommendation, call the `getEstimatedRowCount()` method on a view object to determine how many rows would be returned the user's query *before* actually executing the query and allowing the user to proceed. If the estimated row count is unreasonably large, your application can demand that the end-user provide additional search criteria.

However, when you *must* work with very large result sets, you can use the view object's access mode called "range paging" to improve performance. The feature allows your applications to page back and forth through data, a range of rows at a time, in a way that is more efficient for large data sets than the default "scrollable" access mode.

27.1.5.1 Understanding How to Oracle Supports "TOP-N" Queries

The Oracle database supports a feature called a "Top-N" query to efficiently return the first *N* ordered rows in a query. For example, if you have a query like:

```
SELECT EMPNO, ENAME,SAL FROM EMP ORDER BY SAL DESC
```

If you want to retrieve the top 5 employees by salary, you can write a query like:

```
SELECT * FROM (
   SELECT X.*,ROWNUM AS RN FROM (
      SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY SAL DESC
   ) X
) WHERE RN <= 5
```

which gives you results like:

```
     EMPNO ENAME       SAL   RN
---------- -------- ------ ----
      7839 KING       5000    1
      7788 SCOTT      3000    2
      7902 FORD       3000    3
      7566 JONES      2975    4
      7698 BLAKE      2850    5
```

The feature is not only limited to retrieving the first *N* rows in order. By adjusting the criteria in the outmost `WHERE` clause you can efficiently retrieve any range of rows in the query's sorted order. For example, to retrieve rows `6` through `10` you could alter the query this way:

```
SELECT * FROM (
  SELECT X.*,ROWNUM AS RN FROM  (
     SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY SAL DESC
  ) X
) WHERE RN BETWEEN 6 AND 10
```

Generalizing this idea, if you want to see page number *P* of the query results, where each page contains *R* rows, then you would write a query like:

```
SELECT * FROM (
  SELECT X.*,ROWNUM AS RN FROM (
    SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY SAL DESC
  ) X
) WHERE RN BETWEEN ((:P - 1) * :R) + 1 AND (:P) * :R
```

As the result set you consider grows larger and larger, it becomes more and more efficient to use this technique to page through the rows. Rather than retrieving hundreds or thousands of rows over the network from the database, only to display ten of them on the page, instead you can produce a clever query to retrieve only the `R` rows on page number `P` from the database. No more than a handful of rows at a time needs to be returned over the network when you adopt this strategy.

To implement this database-centric paging strategy in your application, you could handcraft the clever query yourself and write code to manage the appropriate values of the `:R` and `:P` bind variables. Alternatively, you can use the view object's range paging access mode, which implements it automatically for you.

27.1.5.2 How to Enable Range Paging for a View Object

To enable range paging for your view object, first call `setRangeSize()` to define the number of rows per page, then call the following method:

```
yourViewObject.setAccessMode(RowSet.RANGE_PAGING);
```

27.1.5.3 What Happens When You Enable Range Paging

When a view object's access mode is set to `RANGE_PAGING`, the view object takes its default query like:

```
SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
```

and automatically "wraps" it to produce a Top-N query.

For best performance, the statement uses a combination of greater than and less than conditions instead of the `BETWEEN` operator, but the logical outcome is the same as the Top-N wrapping query you saw above. The actual query produced to wrap a base query of:

```
SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
```

looks like this:

```
SELECT * FROM (
  SELECT /*+ FIRST_ROWS */ IQ.*, ROWNUM AS Z_R_N FROM (
    SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
  ) IQ  WHERE ROWNUM < :0)
WHERE Z_R_N > :1
```

The two bind variables are bound as follows:

- `:1` index of the first row in the current page
- `:0` is bound to the last row in the current page

27.1.5.4 How are View Rows Cached When Using Range Paging?

When a view object operates in `RANGE_PAGING` access mode, it only keeps the current range (or "page") of rows in memory in the view row cache at a time. That is, if you are paging through results ten at a time, then on the first page, you'll have rows 1 through 10 in the view row cache. When you navigate to page two, you'll have rows 11 through 20 in the cache. This also can help make sure for large row sets that you don't end up with tons of rows cached just because you want to preserve the ability to scroll backwards and forwards.

27.1.5.5 How to Scroll to a Given Page Number Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, to scroll to page number *`N`* call its `scrollToRangePage()` method, passing *`N`* as the parameter value.

27.1.5.6 Estimating the Number of Pages in the Row Set Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, you can access an estimate of the total number of pages the entire query result would produce using the `getEstimatedRangePageCount()` method.

27.1.5.7 Accommodating Inserts and Deletes Using Auto Posting

The range paging access mode is typically used for paging through read-only row sets, and often is used with read-only view objects. You allow the user to find the row they are looking for by paging through a large row set with range paging access mode, then you use the `Key` of that row to find the selected row in a different view object for editing.

Additionally, the view object supports a `RANGE_PAGING_AUTO_POST` access mode to accommodate the inserting and deleting of rows from the row set. This mode behaves like the `RANGE_PAGING` mode, except that it eagerly calls `postChanges()` on the database transaction whenever any changes are made to the row set. This communicates the pending changes to the database via appropriate `INSERT`, `UPDATE`, or `DELETE` statements so that the changes are preserved when scrolling backward or forward.

27.1.5.8 Understanding the Tradeoffs of Using Range Paging Mode

You might ask yourself, "Why wouldn't I *always* want to use `RANGE_PAGING` mode?" The answer is that using range paging potentially causes more overall queries to be executed as you are navigating forward and backward through your view object rows. You would want to avoid using `RANGE_PAGING` mode in these situations:

- You plan to read all the rows in the row set immediately (for example, to populate a dropdown list).

  In this case your range size would be set to `-1` and there really is only a single "page" of all rows, so range paging does not add value.

- You need to page back and forth through a small-sized row set.

  If you have 100 rows or fewer, and are paging through them 10 at a time, with `RANGE_PAGING` mode you will execute a query each time you go forward and

backward to a new page. In normal mode, you will cache the view object rows as you read them in, and paging backwards through the previous pages will not re-execute queries to show those already-seen rows.

In the case of a very large (or unpredictably large) row set, the trade off of potentially doing a few more queries — each of which only returns up to the `RangeSize` number of rows from the database — is more efficient then trying to cache all of the previously-viewed rows. This is especially true if you allow the user to jump to an arbitrary page in the list of results. Doing so in normal, scrollable mode requires fetching and caching all of the rows between the current page and the page the users jumps to. In `RANGE_PAGING` mode, it will ask the database just for the rows on that page. Then, if the user jumps back to a page of rows that they have already visited, in `RANGE_PAGING` mode, those rows get re-queried again since only the current page of rows is held in memory in this mode.

27.1.6 Setting Up a Data Model with Multiple Masters

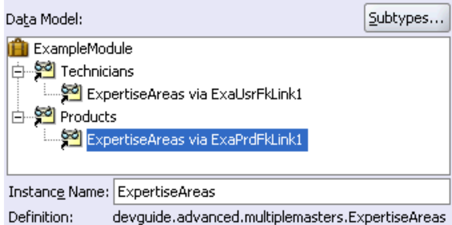
When useful, you can set up your data model to have multiple master view object instances for the same detail view object instance. Consider view objects named `Technicians`, `Products`, and `ExpertiseAreas` with view links defined between:

- `Products` and `ExpertiseAreas`
- `Technicians` and `ExpertiseAreas`

> **Note:** The example in this section refer to the `MultipleMasters` project in the `AdvancedViewObjectExamples` workspace. See the note at the beginning of this chapter for download instructions.

Figure 27–1 shows what the data model panel looks like when you've configured both `Technicians` and `Products` view object instances to be masters of the same `ExpertiseAreas` view object instance.

Figure 27–1 Multiple Master View Object Instances for the Same Detail



To set up the data model as shown in Figure 27–1 open the Application Module Editor and follow these steps on the **Data Model** panel:

1. Add an instance of the `Technicians` view object to the data model.

   Assume you name it `Technicians`.

2. Add an instance of the `Products` view object to the data model

   Assume you name it `Products`.

3. Select the `Technicians` view object instance in the **Data Model** list

4. In the **Available View Objects** list, select the `ExpertiseAreas` view object indented beneath the `Technicians` view object, enter the view object instance name of `ExpertiseAreas` in the **Name** field, and click **>** to shuttle it into data model as a detail of the existing `Technicians` view object instance.
5. Select the `Products` view object instance in the **Data Model** list
6. In the **Available View Objects** list, select the `ExpertiseAreas` view object indented beneath the `Products` view object, enter the same view object instance name of `ExpertiseAreas` in the **Name** field, and click **>** to shuttle it into data model as a detail of the existing `Products` view object instance.

   An alert will appear: **An instance of a View Object with the name ExpertiseAreas has already been used in the data model. Would you like to use the same instance?**
7. Click **Yes** to confirm you want the `ExpertiseAreas` view object instance to *also* be the detail of the `Products` view object instance.

27.1.7 Understanding When You Can Use Partial Keys with findByKey()

View objects based on multiple entity usages support the ability to find view rows by specifying a partially populated key. A partial key is a multi-attribute `Key` object with some of its attributes set to `null`. However, there are strict rules about what kinds of partial keys can be used to perform the `findByKey()`.

If a view object is based on *N* entity usages, where *N* > 1, then the view row key is by default comprised of all of the primary key attributes from *all* of the participating entity usages. Only the ones from the *first* entity object are *required* to participate in the view row key, but by default all of them do.

If you allow the key attributes from some *secondary* entity usages to remain as key attributes at the view row level, then you should leave *all* of the attributes that form the primary key for that entity object as part of the view row key. Assuming you have left the one or more key attributes in the view row for *M* of the *N* entity usages, where (*M* <= *N*), then you can use `findByKey()` to find rows based on any subset of these *M* entity usages. Each entity usage for which you provide values in the `Key` object, requires that you must provide non-null values for *all* of the attributes in that entity's primary key.

You have to follow this rule because when a view object is based on at least one or more entity usages, its `findByKey()` method finds rows by delegating to the `findByPrimaryKey()` method on the entity definition corresponding to the first entity usage whose attributes in the view row key are non-null. The entity definition's `findByPrimaryKey()` method requires all key attributes for any given entity object to be non-null in order to find the entity row in the cache.

As a concrete example, imagine that you have a `ServiceRequests` view object with an `ServiceRequest` entity object as its primary entity usage, and a `Product` entity as secondary reference entity usage. Furthermore, assume that you leave the **Key Attribute** property of *both* of the following view row attributes set to `true`:

- `SvrId` — primary key for the `ServiceRequest` entity
- `ProdId1` — primary key for the `Product` entity

The view row key will therefore be the (`SvrId`, `ProdId1`) combination. When you do a `findByKey()`, you can provide a `Key` object that provides:

- A completely specified key for the underlying `ServiceRequest` entity

```
Key k = new Key(new Object[]{new Number(200), null});
```

- A completely specified key for the underlying `Product` entity

```
Key k = new Key(new Object[]{null, new Number(118)});
```

- A completely specified key for both entities

```
Key k = new Key(new Object[]{new Number(200), new Number(118)});
```

When a valid partial key is specified, the `findByKey()` method can return multiple rows as a result, treating the missing entity usage attributes in the Key object as a wildcard.

27.1.8 Creating Dynamic Attributes to Store UI State

You can add one or more dynamic attributes to a view object at runtime using the `addDynamicAttribute()` method. Dynamic attributes can hold any object as their value. Typically, you will consider using dynamic attributes when writing generic framework extension code that requires storing some additional per-row transient state to implement a feature you want to add to the framework in a global, generic way.

27.1.9 Working with Multiple Row Sets and Row Set Iterators

While you typically work with a view object's *default* row set, you can call the `createRowSet()` method on the `ViewObject` interface to create secondary, named row sets based on the same view object's query. One situation where this could make sense is when your view object's SQL query contains named bind variables. Since each `RowSet` object stores its own copy of bind variable values, you could use a single view object to produce and process multiple row sets based on different combinations of bind variables values. You can find a named row set you've created using the `findRowSet()` method. When you're done using a secondary row set, call its `closeRowSet()` method.

For any `RowSet`, while you typically work with its *default* row set iterator, you can call the `createRowSetIterator()` method of the `RowSet` interface to create secondary, named row set iterators. You can use find a named row set iterator you've created using the `findRowSetIterator()` method. When you're done using a secondary row set iterator, call its `closeRowSetIterator()` method.

Note: Through the ADF Model declarative data binding layer, user interface pages or panels in your application work with the default row set iterator of the default row set of view objects in the application module's data model. Due to this fact, the most typical scenario for creating secondary row set iterators is to write business logic that iterates over a view object's default row set without disturbing the current row of the *default* row set iterator used by the user interface layer.

27.1.10 Optimizing View Link Accessor Access By Retaining the Row Set

Each time you retrieve a view link accessor row set, by default the view object creates a new `RowSet` object to allow you to work with the rows. This does *not* imply re-*executing* the query to produce the results each time, only creating a new instance of a `RowSet` object with its default iterator reset to the "slot" before the first row. To force the row set to refresh its rows from the database, you can call its `executeQuery()` method.

Since there is a small amount of overhead associated with creating the row set, if your code makes numerous calls to the same view link accessor attributes you can consider enabling view link accessor row set retention for the source view object in the view link. To use the view link accessor retention feature, enable a custom Java class for your view object, override the `create()` method, and add a line after `super.create()` that calls the `setViewLinkAccessorRetained()` method passing `true` as the parameter. It affects all view link accessor attributes for that view object.

When this feature is enabled for a view object, since the view link accessor row set it not recreated each time, the current row of its default row set iterator is also retained as a side-effect. This means that your code will need to explicitly call the `reset()` method on the row set you retrieve from the view link accessor to reset the current row in its default row set iterator back to the "slot" before the first row.

Note, however, that with accessor retention enabled, your failure to call `reset()` each time before you iterate through the rows in the accessor row set can result in a subtle, hard-to-detect error in your application. For example, if you iterate over the rows in a view link accessor row set like this, for example to calculate some aggregate total:

```
RowSet rs = (RowSet)row.getAttribute("ServiceRequestsForProduct");
while (rs.hasNext()) {
  Row r = rs.next();
  // Do something important with attributes in each row
}
```

The first time you work with the accessor row set the code will work. However, since the row set (and its default row set iterator) are retained, the second and subsequent times you access the row set the current row will already be at the end of the row set and the while loop will be skipped since `rs.hasNext()` will be `false`. Instead, with this feature enabled, write your accessor iteration code like this:

```
RowSet rs = (RowSet)row.getAttribute("ServiceRequestsForProduct");
rs.reset(); // Reset default row set iterator to slot before first row!
while (rs.hasNext()) {
  Row r = rs.next();
  // Do something important with attributes in each row
}
```

Recall that if view link consistency is on, when the accessor is retained the new unposted rows will show up at the end of the row set. This is slightly different from when the accessor is not retained (the default), where new unposted rows will appear at the beginning of the accessor row set.

27.2 Tuning Your View Objects for Best Performance

You can use view objects to read rows of data, create and store rows of transient data, as well as automatically coordinate inserts, updates, and deletes made by end users with your underlying business objects. How you design and use your view objects can definitely affect their performance at runtime. This section provides guidance on configuring your view objects to get the best possible performance.

27.2.1 Use Bind Variables for Parameterized Queries

Whenever the `WHERE` clause of your query includes values that might change from execution to execution, you should use named bind variables. Their use also protects your application against abuse through SQL injection attacks by malicious end-users.

27.2.1.1 Use Bind Variables to Avoid Re-parsing of Queries

Bind variables are place holders in the SQL string whose value you can easily change at runtime without altering the text of the SQL string itself. Since the query text doesn't change from execution to execution, the database can efficiently reuse the same parsed statement each time. Avoiding re-parsing of your statement alleviates the database from having to continually re-determine its query optimization plan and eliminates contention by multiple end-users on other costly database resources used during this parsing operation. This savings leads to higher runtime performance of your application. See Section 5.9, "Using Named Bind Variables" for details on how to use named bind variables.

27.2.1.2 Use Bind Variables to Prevent SQL-Injection Attacks

Using bind variables for parameterized `WHERE` clause values is especially important if their values will be supplied by *end-users* of your application. Consider the example shown in Example 27–2. It adds a dynamic `WHERE` clause formed by concatenating a user-supplied parameter value into the statement.

Example 27–2 Using String Concatenation Instead of Bind Variables is Vulnerable to SQL-Injection Attacks

```
// EXAMPLE OF BAD PRACTICE, Do not follow this approach!
String userSuppliedValue = ... ;
yourViewObject.setWhereClause("BANK_ACCOUNT_ID = "+userSuppliedValue);
```

A user with malicious intentions — if able to learn any details about your application's underlying database schema — could supply a carefully-constructed "bank account number" as a field value or URL parameter like:

```
BANK_ACCOUNT_ID
```

When the code in Example 27–2 concatenates this value into the dynamically-applied where clause, what the database sees is a query predicate like this:

```
WHERE (BANK_ACCOUNT_ID = BANK_ACCOUNT_ID)
```

This `WHERE` clause retrieves *all* bank accounts instead of just the current user's, perhaps allowing the hacker to view private information of another person's account. This technique of short-circuiting an application's `WHERE` clause by trying to supply a maliciously-constructed parameter value into a SQL statement is called a SQL injection attack. Using named bind variables instead for these situations as shown in Example 27–3 prevents the vulnerability.

Example 27–3 Use Named Bind Variables Instead of String Concatenation

```
// Best practice using named bind variables
String userSuppliedValue = ... ;
yourViewObject.setWhereClause("BANK_ACCOUNT_ID = :BankAcccountId");
yourViewObject.defineNamedWhereClauseParam("BankAcccountId", null, null);
yourViewObject.setNamedWhereClauseParam("BankAcccountId",userSuppliedValue);
```

If a malicious user supplies an illegal value in this case, they receive an error your application can handle instead of obtaining data they are not suppose to see.

27.2.2 Use Read-Only View Objects When Entity-Based Features Not Required

View objects can either be related to underlying entity objects or not. When a view object is related to one or more underlying entity objects you can create new rows, and modify or remove queried rows. The view object coordinates with underlying entity objects to enforce business rules and to permanently save the changes. In addition, entity-based view objects:

- Immediately reflect pending changes made to relevant entity object attributes made through other view objects in the same transaction
- Initialize attribute values in newly created rows to the values from the underlying entity object attributes
- Reflect updated reference information when foreign key attribute values are changed

On the other hand, view objects that are not related to any entity object are read-only, do not pickup entity-derived default values, do not reflect pending changes, and do not reflect updated reference information. You need to decide what kind of functionality your application requires and design the view object accordingly. Typically view objects used for SQL-based validation purposes, as well as for displaying the list of valid selections in a dropdown list, can be read-only.

There is a small amount of runtime overhead associated with the coordination between view object rows and entity object rows, so if you don't need any of the functionality offered by an entity-mapped view object, you can slightly increase performance by using a read-only view object with no related entity objects.

27.2.3 Use SQL Tracing to Identify Ill-Performing Queries

After deciding whether your view object should be mapped to entities or not, your attention should turn to the query itself. The **Explain Plan** button on the **Query** panel of the View Object Editor allows you to see the query plan that the database query optimizer will use. If you see that it is doing a full table scan, you should consider adding indexes or providing a value for the **Optimizer Hint** field on the **Tuning** panel to explicitly control which query plan will be used. These facilities provide some useful tools to the developer to evaluate the query plans for individual view object SQL statements. However, their use is not a substitute for tracing the SQL of the entire application to identify poorly performing queries in the presence of a production environment's amount of data and number of end users.

You can use the Oracle database's SQL Tracing facilities to produce a complete log of all SQL statements your application performs. The approach that works in all versions of the Oracle database is to issue the command:

```
ALTER SESSION SET SQL_TRACE TRUE
```

This command enables tracing of the current database session and logs all SQL statements to a server-side trace file until you either enter `ALTER SESSION SET SQL_TRACE FALSE` or close the connection. To simplify enabling this option to trace your ADF applications, override the `afterConnect()` method of your application module (or custom application module framework extension class) to conditionally perform the `ALTER SESSION` command to enable SQL tracing based on the presence of a Java system property as shown in Example 27–4.

Example 27–4 Conditionally Enabling SQL Tracing in an Application Module

```
// In YourCustomApplicationModuleImpl.java
protected void afterConnect() {
  super.afterConnect();
  if (System.getProperty("enableTrace") != null) {
    getDBTransaction().executeCommand("ALTER SESSION SET SQL_TRACE TRUE");
  }
}
```

After producing a trace file, you use the `tkprof` utility supplied with the database to format the information and to better understand information about each query executed like:

- The number of times it was (re)parsed
- The number of times it was executed
- How many round-trips were made between application server and the database
- Various quantitative measurements of query execution time

Using these techniques, you can decide which additional indexes might be required to speed up particular queries your application performs, or which queries could be changed to improve their query optimization plan.

> **Note:** The Oracle 10g database provides the new `DBMS_MONITOR` package that further simplifies SQL tracing and integrates it with Oracle Enterprise Manager for visually monitoring the most frequently performed query statements your applications perform.

27.2.4 Consider the Appropriate Tuning Settings for Every View Object

The **Tuning** panel of the View Object Editor lets you set various options that can dramatically effect your query's performance.

27.2.4.1 Set the Database Retrieval Options Appropriately

The **Retrieve from the Database** section, controls how the view object retrieves rows from the database server. The options for the fetch mode are **All Rows**, **At Most One Row**, and **No Rows**. Most view objects will stick with the default **All Rows** option, which will be retrieved **As Needed** or **All at Once** depending on which option you choose. The "as needed" option ensures that an `executeQuery()` operation on the view object initially retrieves only as many rows as necessary to fill the first page of a display, whose number of rows is set based on the view object's range size.

For view objects whose `WHERE` clause expects to retrieve a *single* row, set the option to **At Most One Row** for best performance. This way, the view object knows you don't expect any more rows and will skip its normal test for that situation. Finally, if you use the view object only for creating new rows, set the option to **No Rows** so no query will ever be performed.

27.2.4.2 Consider Whether Fetching One Row at a Time is Appropriate

The fetch size controls how many rows will be returned in each round trip to the database. By default, the framework will fetch rows in batches of one row at a time. If you are fetching any more than one row, you will gain efficiency by setting this **in Batches of** value.

However the higher the number, the larger the client-side buffer required, so avoid setting this number arbitrarily high. If you are displaying results `N` rows at a time in the user interface, it's good to set the fetch size to at least `N+1` so that each page of results can be retrieved in a single round trip to the database.

Caution: Unless your query *really* fetches just one row, leaving the *default* fetch size of one (`1`) in the **in Batches of** field on the **Tuning** panel is a recipe for bad performance due to many unnecessary round trips between the application server and the database. Oracle strongly recommends considering the appropriate value for each view object's fetch size.

27.2.4.3 Specify a Query Optimizer Hint if Necessary

The **Query Optimizer Hint** field allows you to specify an optional hint to the Oracle query optimizer to influence what execution plan it will use. At runtime, the hint you provide is added immediately after the `SELECT` keyword in the query, wrapped by the special comment syntax `/*+ YOUR_HINT */`. Two common optimizer hints are:

- `FIRST_ROWS` — to hint that you want the first rows as quickly as possible
- `ALL_ROWS` — to hint that you want all rows as quickly as possible

There are many other optimizer hints that are beyond the scope of this manual to document. Reference the Oracle 10g database reference manuals for more information on available hints.

27.2.5 Creating View Objects at Design Time

It's important to understand the overhead associated with creating view objects at runtime. Avoid the temptation to do this without a compelling business requirement. For example, if your application issues a query against a table whose name you know at design time and if the list of columns to retrieve is also fixed, then create a view object at design time. When you do this, your SQL statements are neatly encapsulated, can be easily explained and tuned during development, and incur no runtime overhead to discover the structure and data types of the resulting rows.

In contrast, when you use the `createViewObjectFromQueryStmt()` API on the `ApplicationModule` interface at runtime, your query is buried in code, it's more complicated to proactively tune your SQL, and you pay a performance penalty each time the view object is created. Since the SQL query statement for a dynamically-created view object could theoretically be different on each execution, an extra database round trip is required to discover the "shape" of the query results on-the-fly. Only create queries dynamically if you *cannot* know the name of the table to query until runtime. Most other needs can be addressed using a design-time created view object in combination with runtime API's to set bind variables in a fixed where clause, or to add an additional `WHERE` clause (with optional bind variables) at runtime.

27.2.6 Use Forward Only Mode to Avoid Caching View Rows

Often you will use write code that programmatically iterates through the results of a view object. A typical situation will be custom validation code that must process multiple rows of query results to determine whether an attribute or an entity is valid or not. In these cases, if you intend to read each row in the row set a single time and never require scrolling backward or re-iterating the row set a subsequent time, then you can use "forward only" mode to avoid caching the retrieved rows. To enable forward only mode, call `setForwardOnly(true)` on the view object.

Note: Using a read-only view object (with no entity usages) in forward-only mode with an appropriately tuned fetch size is the most efficient way to programmatically read data.

You can also use forward-only mode to avoid caching rows when inserting, updating, or deleting data as long as you never scroll backward through the row set and never call `reset()` to set the iterator back to the first row. Forward only mode only works with a range size of one (`1`).

27.3 Using Expert Mode for Full Control Over SQL Query

When defining entity-based view objects, you can fully-specify the `WHERE` and `ORDER BY` clauses, whereas, by default, the `FROM` clause and `SELECT` list are automatically derived. The names of the tables related to the participating entity usages determine the `FROM` clause, while the `SELECT` list is based on the:

- Underlying column names of participating entity-mapped attributes
- SQL expressions of SQL-calculated attributes

When you require full control over the `SELECT` or `FROM` clause in a query, you can enable "Expert Mode".

27.3.1 How to Enable Expert Mode for Full SQL Control

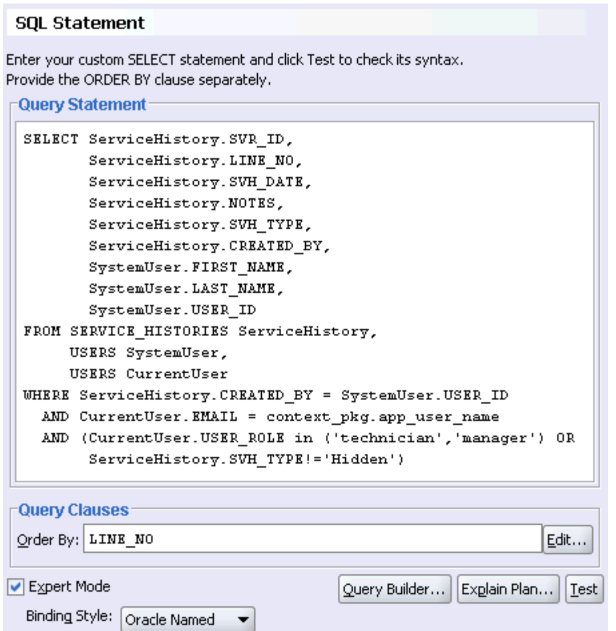
To enable expert mode, select **Expert Mode** on the **SQL Statement** panel of the Create View Object wizard or View Object Editor.

27.3.2 What Happens When You Enable Expert Mode

When you enable expert mode, the read-only **Generated Statement** section of the **SQL Statement** panel becomes a fully-editable **Query Statement** text box, displaying the full SQL statement. Using this text box, you can change every aspect of the SQL query.

For example, Example 27–2 shows the **SQL Statement** page of the View Object editor for the SRDemo application's `ServiceHistories` view object. It's an expert mode, entity-based view object that references a PL/SQL function `context_pkg.app_user_name` and joins the USERS table an additional time in the FROM clause to filter hidden service history notes from end-users who are not in the `technician` or `manager` roles.

Figure 27–2 ServiceHistories Expert Mode View Object in the SRDemo Application



27.3.3 What You May Need to Know

27.3.3.1 You May Need to Perform Manual Attribute Mapping

The automatic cooperation of a view object with its underlying entity objects depends on correct attribute-mapping metadata saved in the XML component definition. This information relates the view object attributes to corresponding attributes from participating entity usages. JDeveloper maintains this attribute mapping information in a fully-automatic way for normal entity-based view objects. However, when you decide to use expert mode with a view object, you need to pay attention to the changes you make to the `SELECT` list. That is the part of the SQL query that directly relates to the attribute mapping. Even in expert mode, JDeveloper continues to offer some assistance in maintaining the attribute mapping metadata when you do the following to the `SELECT` list:

- Reorder an expression without changing its column alias

  JDeveloper reorders the corresponding view object attribute and maintains the attribute mapping.

- Add a new expression
- JDeveloper adds a new SQL-calculated view object attribute with a corresponding Camel-Capped name based on the column alias of the new expression.
- Remove an expression

  JDeveloper converts the corresponding SQL-calculated or entity-mapped attribute related to that expression to a transient attribute.

However, if you rename a column alias in the `SELECT` list, JDeveloper has no way to detect this, so it is treated as if you removed the old column expression and added a new one of a different name.

After making any changes to the SELECT list of the query, visit the **Attribute Mappings** panel to ensure that the attribute mapping metadata is correct. The table on this panel, which is disabled for view objects in normal mode, becomes enabled for expert mode view objects. For each view object attribute, you will see its corresponding SQL column alias in the table. By clicking into a cell in the **View Attributes** column, you can use the dropdown list that appears to select the appropriate entity object attribute to which any entity-mapped view attributes should correspond.

Note: If the view attribute is SQL-calculated or transient, a corresponding attribute with a “SQL” icon appears in the **View Attributes** column to represent it. Since these attributes are not related to underlying entity objects, there is no entity attribute related information required for them.

27.3.3.2 Disabling Expert Mode Loses Any Custom Edits

When you disable expert mode for a view object it will return to having its SELECT and FROM clause be derived again. JDeveloper warns you that doing this might lose any of your custom edits to the SQL statement. If this is what you want, after acknowledging the alert, your view object's SQL query reverts back to the default.

27.3.3.3 Once In Expert Mode, Changes to SQL Expressions Are Ignored

Consider a Products view object with a SQL-calculated attribute named Shortens whose SQL expression you defined as SUBSTR(NAME,1,10). If you switch this view object to expert mode, the **Query Statement** box will show a SQL query like this:

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
       Products.DESCRIPTION,
       SUBSTR(NAME,1,10) AS SHORT_NAME
FROM PRODUCTS Products
```

If you go back to the attribute definition for the Shortens attribute and change the **SQL Expression** field from SUBSTR(NAME,1,10) to SUBSTR(NAME,1,15), then the change will be saved in the view object's XML component definition. Note, however, that the SQL query will remain as above. This occurs because JDeveloper never tries to *modify* the text of an expert mode query. In expert mode, the developer is in full control. JDeveloper attempts to adjust metadata as described above in function of some kinds of changes you make yourself to the expert mode SQL statement, but it does not perform the reverse. Therefore, if you change view object metadata, the expert mode SQL statement is not updated to reflect it.

To make the above change to the SQL calculated Shortens attribute, you need to update the expression in the expert mode SQL statement itself. To be 100% thorough, you should make the change *both* in the attribute metadata *and* in the expert mode SQL statement. This would ensure — if you (or another developer on your team) ever decides to toggle expert mode *off* at a later point in time — that the automatically derived SELECT list would contain the correct SQL-derived expression.

Note: If you find you had to make numerous changes to the view object metadata of an expert mode view object, you can consider the following technique to avoid having to manually translate any effects those changes might have implied to the SQL statement yourself. First, copy the text of your customized query to a temporary file. Then, disable expert mode for the view object and acknowledge the warning that you will lose your changes. At this point JDeveloper will re-derive the correct generated SQL statement based on all the new metadata changes you've made. Finally, you can enable expert mode once again and re-apply your SQL customizations.

27.3.3.4 Don't Map Incorrect Calculated Expressions to Entity Attributes

When changing the `SELECT` list expression that corresponds to *entity-mapped* attributes, don't introduce SQL calculations that change the value of the attribute when retrieving the data. To illustrate the problem that will occur if you do this, consider the following query for a simple entity-based view object named `Products`:

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
       Products.DESCRIPTION
FROM PRODUCTS Products
```

Imagine that you wanted to limit the name column to showing only the first ten characters of the name for some use case. The correct way to do that would be to introduce a new SQL-calculated field called `ShortName` with an expression like `SUBSTR(Products.NAME,1,10)`. However, one way you might have thought to accomplish this was to switch the view object to expert mode and change the `SELECT` list expression for the entity-mapped NAME column to the following:

```
SELECT Products.PROD_ID,
       SUBSTR(Products.NAME,1,10) AS NAME,
       Products.IMAGE,
       Products.DESCRIPTION
FROM PRODUCTS Products
```

This alternative strategy would initially appear to work. At runtime, you see the truncated value of the name as you are expecting. However, if you modify the row, when the underlying entity object attempts to lock the row it does the following:

- Issues a `SELECT FOR UPDATE` statement, retrieving all columns as it tries to lock the row.
- If the entity object successfully locks the row, it compares the original values of all the persistent attributes in the entity cache as they were last retrieved from the database with the values of those attributes just retrieved from the database during the lock operation.
- If any of the values differs, then the following error is thrown:

  ```
  (oracle.jbo.RowInconsistentException)
  JBO-25014: Another user has changed the row with primary key [...]
  ```

If you see an error like this at runtime even though you are the *only* user testing the system, it is most likely due to your inadvertently introducing a SQL function in your expert mode view object that changed the selected value of an entity-mapped attribute. In the example above, the `SUBSTR(Products.NAME,1,10)` function introduced causes the original selected value of the `Name` attribute to be truncated.

When the row-lock SQL statement selects the value of the NAME column, it will select the entire value. This will cause the comparison described above to fail, producing the "phantom" error that another user has changed the row.

The same thing would happen with NUMBER, or DATE valued attributes if you inadvertently apply SQL functions in expert mode to truncate or alter their retrieved values for entity-mapped attributes. If you need to present altered versions of entity-mapped attribute data, introduce a new SQL-calculated attribute with the appropriate expression to handle the job.

27.3.3.5 Expert Mode SQL Formatting is Retained

When you switch a view object to expert mode, its XML component definition switches from storing parts of the query in separate XML attributes, to saving the entire query in a single <SQLQuery> element. The query is wrapped in a XML CDATA section to preserve the line formatting you may have done to make a complex query be easier to understand.

27.3.3.6 Expert Mode Queries Are Wrapped as Inline Views

If your expert-mode view object:

- Contains a design-time ORDER BY clause specified in the **Order By** field of the **Query Clauses** panel, or
- Has a dynamic where clause or order by clause applied at runtime using setWhereClause() or setOrderByClause()

then its query gets nested into an inline view before applying these clauses. For example, suppose your expert-mode query was defined as:

```
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
union all
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_USERS
```

At runtime, when you set an additional WHERE clause like email = :TheUserEmail, the view object nests its original query into an inline view like this:

```
SELECT * FROM(
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
union all
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_USERS) QRSLT
```

and then adds the dynamic where clause predicate at the end, so that the final query the database sees is:

```
SELECT * FROM(
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from USERS
union all
select USER_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_USERS) QRSLT
WHERE email = :TheUserEmail
```

This query "wrapping" is necessary in general for expert mode queries since the original query could be arbitrarily complex, including SQL `UNION`, `INTERSECT`, `MINUS`, or other operators that combine multiple queries into a single result. In those cases, simply "gluing" the additional runtime `WHERE` clause onto the end of the query text could produce unexpected results since. For example, it might only apply to the *last* of several `UNION`'ed statements. By nesting the original query verbatim into an inline view, the view object guarantees that your additional `WHERE` clause is correctly used to filter the results of the original query, regardless of how complex it is.

27.3.3.7 Disabling the Use of Inline View Wrapping at Runtime

Due to the inline view wrapping of expert mode view objects, the dynamically-added `WHERE` clause can only refer to columns in the `SELECT` list of the original query. To avoid this limitation, when necessary you can disable the use of the inline view wrapping by calling `setNestedSelectForFullSql(false)`.

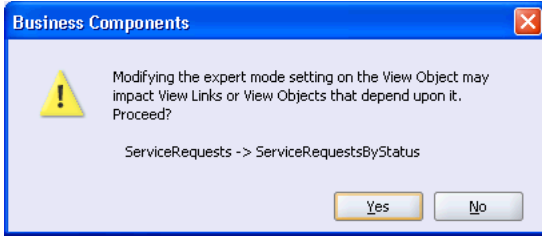
27.3.3.8 Enabling Expert Mode May Impact Dependent Objects

When you modify a query to be in expert mode after you have already created:

- View links involving it, or
- Other view objects that extend it

JDeveloper will warn you with the alert shown in Figure 27–3 to remind you that you should revisit these dependent components to ensure their SQL statements still reflect the correct query.

Figure 27–3 Proactive Reminder to Revisit Dependent Components



For example, if you were to modify the `ServiceRequests` view object in the SRDemo application to use expert mode, since the `ServiceRequestsByStatus` view object extends it, you need to revisit the extended component to ensure its query still logically reflects an extension of the modified parent component.

27.4 Working with Multiple Named View Criteria

You can define *multiple* named view criteria and then selectively apply any combination of them to your view object at runtime as needed.

Note: The examples in this section refer to the `MultipleViewCriterias` project in the `AdvancedViewObjectExamples` workspace. See the note at the beginning of this chapter for download instructions.

27.4.1 Defining Named View Criteria

To define named view criteria, you override the `create()` method in your view object's custom Java class and call the `putViewCriteria()` method to define one or more named `ViewCriteria` objects.

For example, given a `Users` view object based on the `USERS` table in the SRDemo schema, you could override the `create()` method as shown in Example 27–5 to define named view criteria called `CountryIsUS`, `CountryIsNotUS`, `IsStaff`, and `IsCustomer` by calling appropriate helper methods.

Example 27–5 Defining Multiple Named View Criteria in an Overridden create() Method

```
package devguide.advanced.multiplevc;
// Imports omitted
public class UsersImpl extends ViewObjectImpl implements Users {
  // etc.
  protected void create() {
    super.create();
    defineCountryIsUSCriteria();
    defineCountryIsNotUSCriteria();
    defineIsStaffCriteria();
    defineIsCustomerCriteria();
  }
  private void defineCountryIsUSCriteria() {
    ViewCriteria vc = createViewCriteria();
    ViewCriteriaRow vcr = vc.createViewCriteriaRow();
    vcr.setAttribute("CountryId","US");
    vc.add(vcr);
    putViewCriteria("CountryIsUS",vc);
  }
  private void defineCountryIsNotUSCriteria() {
    ViewCriteria vc = createViewCriteria();
    ViewCriteriaRow vcr = vc.createViewCriteriaRow();
    vcr.setAttribute("CountryId","US");
    vcr.setConjunction(ViewCriteriaRow.VCROW_CONJ_NOT);
    vc.add(vcr);
    putViewCriteria("CountryIsNotUS",vc);
  }
  private void defineIsStaffCriteria() {
    ViewCriteria vc = createViewCriteria();
    ViewCriteriaRow vcr = vc.createViewCriteriaRow();
    vcr.setAttribute("UserRole","IN ('technician','manager')");
    vc.add(vcr);
    putViewCriteria("IsStaff",vc);
  }
  private void defineIsCustomerCriteria() {
    ViewCriteria vc = createViewCriteria();
    ViewCriteriaRow vcr = vc.createViewCriteriaRow();
    vcr.setAttribute("UserRole","user");
    vc.add(vcr);
    putViewCriteria("IsCustomer",vc);
  }
  // etc.
}
```

27.4.2 Applying One or More Named View Criteria

To apply one or more named view criteria, use the `setApplyViewCriteriaNames()` method. This method accepts a String array of the names of the criteria you want to apply. If you apply more than one named criteria, they are `AND`-ed together in the WHERE clause produced at runtime. Then, you can expose custom methods on the client interface of the view object to encapsulate applying combinations of the named view criteria. For example, Example 27–6 shows custom methods `showStaffInUS()`, `showCustomersOutsideUS()`, and `showCustomersInUS()`, each of which uses the `setApplyViewCriteriaNames()` method to apply an appropriate combination of named view criteria. Once these methods are exposed on the view object's client interface, at runtime clients can invoke these methods as needed to change the information displayed by the view object.

Example 27–6 Exposing Client Methods to Enable Appropriate Named Criterias

```
// In UsersImpl.java
public void showStaffInUS() {
  setApplyViewCriteriaNames(new String[]{"CountryIsUS","IsStaff"});
  executeQuery();
}
public void showCustomersOutsideUS() {
  setApplyViewCriteriaNames(new String[]{"CountryIsNotUS","IsCustomer"});
  executeQuery();
}
public void showCustomersInUS() {
  setApplyViewCriteriaNames(new String[]{"CountryIsUS","IsCustomer"});
  executeQuery();
}
```

27.4.3 Removing All Applied Named View Criteria

To remove any currently applied named view criteria, use `setApplyViewCriteriaNames(null)`. For example, you could add the `showAll()` method in Example 27–7 to the `Users` view object and expose it on the client interface. This would allow clients to return to an unfiltered view of the data when needed.

Example 27–7 Removing All Applied Named View Criterias

```
// In UsersImpl.java
public void showAll() {
  setApplyViewCriteriaNames(null);
  executeQuery();
}
```

Note: The `setApplyViewCriterias(null)` removes all applied view criteria, but allows you to later reapply any combination of them. In contrast, the `clearViewCriterias()` method *deletes* all named view criteria. After calling `clearViewCriterias()` you would have to use `putViewCriteria()` again to define new named criteria before you could apply them.

27.4.4 Using the Named Criteria at Runtime

Example 27–8 shows the interesting lines of a `TestClient` class that works with the `Users` view object described above. It invokes different client methods on the `Users` view object interface to show different filtered sets of data. The `showRows()` method is a helper method that iterates over the rows in the view object to display some attributes.

Example 27–8 Test Client Code Working with Named View Criterias

```
// In TestClientMultipleViewCriterias.java
Users vo = (Users)am.findViewObject("Users");
vo.showCustomersOutsideUS();
showRows(vo,"After applying view criterias for customers outside US");
vo.showStaffInUS();
showRows(vo,"After applying view criterias for staff in US");
vo.showCustomersInUS();
showRows(vo,"After applying view criterias for customers in US");
vo.showAll();
showRows(vo,"After clearing all view criterias");
```

Running the `TestClient` program produces output as follows:

```
--- After applying view criterias for customers outside US ---
Hermann Baer [user, DE]
John Chen [user, TH]
:
--- After applying view criterias for staff in US ---
David Austin [technician, US]
Bruce Ernst [technician, US]
:
--- After applying view criterias for customers in US ---
Shelli Baida [user, US]
Emerson Clabe [user, US]
:
--- After clearing all view criterias ---
David Austin [technician, US]
Hermann Baer [user, DE]
:
```

27.5 Performing In-Memory Sorting and Filtering of Row Sets

By default a view object performs its query against the database to retrieve the rows in its resulting row set. However, you can also use view objects to perform in-memory searches and sorting to avoid unnecessary trips to the database.

Note: The examples in this section refer to the `InMemoryOperations` project in the `AdvancedViewObjectExamples` workspace. See the note at the beginning of this chapter for download instructions. The examples illustrate using the in-memory sorting and filtering functionality from the client side using methods on the interfaces in the `oracle.jbo` package. The same functionality can be, and typically *should* be, encapsulated inside custom methods of your application module or view object components, which you expose on their respective client interface.

27.5.1 Understanding the View Object's Query Mode

The view object's query mode controls the source used to retrieve rows to populate its row set. The `setQueryMode()` allows you to control which mode, or combination of modes, are used:

- `ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES`

  This is the default mode that retrieves results from the database.

- `ViewObject.QUERY_MODE_SCAN_VIEW_ROWS`

  This mode uses rows already in the row set as the source, allowing you to progressively refine the row set's contents through in-memory filtering.

- `ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS`

  This mode, valid only for entity-based view objects, uses the entity rows presently in the entity cache as the source to produce results based on the contents of the cache.

You can use the modes individually, or combine them using Java's logical `OR` operator (*X*| *Y*). For example, to create a view object that queries the entity cache for unposted new entity rows, as well as the database for existing rows, you could write code like:

```
setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES |
             ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS)
```

If you combine the query modes, the view object automatically handles skipping of duplicate rows. In addition, there is an implied order to the results that are found:

1. Scan view rows (if specified)
2. Scan entity cache (if specified)
3. Scan database tables (if specified) by issuing a SQL query

If you call the `setQueryMode()` method to change the query mode, your new setting takes effect the next time you call the `executeQuery()` method.

27.5.2 Sorting View Object Rows In Memory

To sort the rows in a view object at runtime, use the `setSortBy()` method. You pass a sort expression that looks like a SQL ORDER BY clause. However, instead of referencing the column names of the table, you use the view object's attribute names. For example, for a view object containing attributes named `DaysOpen` and `CreatedByUser`, you could sort the view object first by `DaysOpen` descending, then by `CreatedByUser` by calling:

```
setSortBy("DaysOpen desc, CreatedByUser");
```

Alternatively, you can use the *zero-based* attribute index position in the sorting clause like this:

```
setSortBy("2 desc, 3");
```

After calling the `setSortBy()` method, the rows will be sorted the next time you call the `executeQuery()` method. The view object translates this sorting clause into an appropriate format to use for ordering the rows depending on the query mode of the view object. If you use the default query mode, the `SortBy` clause is translated into an appropriate `ORDER BY` clause and used as part of the SQL statement sent to the database. If you use either of the in-memory query modes, then the `SortBy` by clause

is translated into one or more SortCriteria objects for use in performing the in-memory sort.

> **Note:** While SQL ORDER BY expressions treat column names in a case-insensitive way, the attribute names in a SortBy expression are case-*sensitive*.

27.5.2.1 Combining setSortBy and setQueryMode for In-Memory Sorting

Example 27–9 shows the interesting lines of code from the TestClientSetSortBy class that uses setSortBy() and setQueryMode() to perform an in-memory sort on the rows produced by a read-only view object ResolvedServiceRequests.

Example 27–9 Combining setSortBy and setQueryMode for In-Memory Sorting

```
// In TestClientSetSortBy.java
am.getTransaction().executeCommand("ALTER SESSION SET SQL_TRACE TRUE");
ViewObject vo = am.findViewObject("ResolvedServiceRequests");
vo.executeQuery();
showRows(vo,"Initial database results");
vo.setSortBy("DaysOpen desc");
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
vo.executeQuery();
showRows(vo,"After in-memory sorting by DaysOpen desc");
vo.setSortBy("DaysOpen desc, CreatedByUser");
vo.executeQuery();
showRows(vo,"After in-memory sorting by DaysOpen desc, CreatedByUser");
```

Running the example produces the results:

```
--- Initial database results ---
106,Ice machine not working,1,mhartste
103,Washing machine leaks,4,ngreenbe
105,Air in dryer not hot,4,jmurman
109,Freezer is not cold,4,jwhalen
:
--- After in-memory sorting by DaysOpen desc ---
100,I have noticed that every time I do a...,9,dfaviet
101,Agitator does not work,8,sbaida
103,Washing machine leaks,4,ngreenbe
105,Air in dryer not hot,4,jmurman
:
--- After in-memory sorting by DaysOpen desc, CreatedByUser ---
100,I have noticed that every time I do a...,9,dfaviet
101,Agitator does not work,8,sbaida
105,Air in dryer not hot,4,jmurman
109,Freezer is not cold,4,jwhalen
:
```

The first line in Example 27–9 containing the executeCommand() call issues the ALTER SESSION SET SQL TRACE command to enable SQL tracing for the current database session. This causes the Oracle database to log every SQL statement performed to a server-side trace file. It records information about the text of each SQL statement, including how many times the database parsed the statement and how many round-trips the client made to fetch batches of rows while retrieving the query result.

Note: You might need a DBA to grant permission to the SRDEMO account to perform the ALTER SESSION command to do the tracing of SQL output.

Once you've produced a trace file, you can use the tkprof utility that comes with the database to format the file:

```
tkprof xe_ora_3916.trc trace.prf
```

This will produces a trace.prf file containing the interesting information shown in Example 27–10 about the SQL statement performed by the ResolvedServiceRequests view object. You can see that after initially querying six rows of data in a single execute and fetch from the database, the two subsequent sorts of those results did not cause any further executions. Since the code set the query mode to ViewObject.QUERY_MODE_SCAN_VIEW_ROWS the setSortBy() followed by the executeQuery() performed the sort in memory.

Example 27–10 TKPROF Output of a Trace File Confirming Sort Was Done In Memory

```
*************************************************************
SELECT * FROM (select sr.svr_id,
       case
         when length(sr.problem_description) > 37 then
         rtrim(substr(sr.problem_description,1,37))||'...'
         else sr.problem_description
       end as problem_description,
       ceil(
         (select trunc(max(svh_date))
            from service_histories
           where svr_id = sr.svr_id)
         - trunc(request_date)
       ) as days_open,
       u.email as created_by_user
from service_requests sr, users u
 where sr.created_by = u.user_id
   and status = 'Closed') QRSLT  ORDER BY days_open

call    count     cpu  elapsed disk  query  current     rows
------- -----  ------ -------- ---- ------ --------  -------
Parse       1    0.00     0.00    0      0        0        0
Execute     1    0.00     0.00    0      0        0        0
Fetch       1    0.00     0.00    0     22        0        6
------- -----  ------ -------- ---- ------ --------  -------
total       3    0.00     0.00    0     22        0        6
*************************************************************
```

27.5.2.2 Extensibility Points for In-Memory Sorting

Should you need to customize the way that rows are sorted in memory, you have the following two extensibility points:

1. You can override the method:

   ```
   public void sortRows(Row[] rows)
   ```

   This method performs the actual in-memory sorting of rows. By overriding this method you can plug in an alternative sorting approach if needed.

2. You can override the method:

   ```
   public Comparator getRowComparator()
   ```

   The default implementation of this method returns an `oracle.jbo.RowComparator`. `RowComparator` invokes the `compareTo()` method to compare two data values. These methods/objects can be overridden to provide custom compare routines.

27.5.3 Performing In-Memory Filtering with View Criteria

To filter the contents of a row set using `ViewCriteria`, you can call:

- `applyViewCriteria()` or `setApplyViewCriteriaNames()` followed by `executeQuery()` to produce a new, filtered row set.
- `findByViewCriteria()` to retrieve a new row set to process programmatically without changing the contents of the original row set.

Both of these approaches can be used against the database or to perform in-memory filtering, or both, depending on the view criteria mode. You set the criteria mode using the `setCriteriaMode()` method on the `ViewCriteria` object, to which you can pass either of the following integer flags, or the logical `OR` of both:

- `ViewCriteria.CRITERIA_MODE_QUERY`
- `ViewCriteria.CRITERIA_MODE_CACHE`

When used for in-memory filtering, the operators supported are `ViewCriteria` are =, >, <, <=, >=, <>, and `LIKE`.

Example 27–11 shows the interesting lines from a `TestClientFindByViewCriteria` class that uses the two features described above both against the database and in-memory. It uses a `CustomerList` view object instance and performs the following basic steps:

1. Queries customers from the database with a last name starting with a 'C', producing the output:

   ```
   --- Initial database results with applied view criteria ---
   John Chen
   Emerson Clabe
   Karen Colmenares
   ```

2. Subsets the results from step 1 in memory to only those with a first name starting with 'J'. It does this by adding a second view criteria row to the view criteria and setting the conjunction to use "AND". This produces the output:

   ```
   --- After augmenting view criteria and applying in-memory ---
   John Chen
   ```

3. Sets the conjunction back to `OR` and re-applies the criteria to the database to query customers with last name like 'J%' or first name like 'C%'. This produces the output:

```
--- After changing view criteria and applying to database again ---
John Chen
Jose Manuel Urman
Emerson Clabe
Karen Colmenares
Jennifer Whalen
```

4. Defines a new criteria to find customers in-memory with first or last name that contain a letter 'o'
5. Uses `findByViewCriteria()` to produce new row set instead of subsetting, producing the output:

```
--- Rows returned from in-memory findByViewCriteria ---
John Chen
Jose Manuel Urman
Emerson Clabe
Karen Colmenares
```

6. Shows that original row set hasn't changed when `findByViewCriteria()` was used, producing the output:

```
--- Note findByViewCriteria didn't change rows in the view ---
John Chen
Jose Manuel Urman
Emerson Clabe
Karen Colmenares
Jennifer Whalen
```

Example 27–11 Performing Database and In-Memory Filtering with View Criteria

```
// In TestClientFindByViewCriteria.java
ViewObject vo = am.findViewObject("CustomerList");
// 1. Show customers with a last name starting with a 'C'
ViewCriteria vc = vo.createViewCriteria();
ViewCriteriaRow vcr1 = vc.createViewCriteriaRow();
vcr1.setAttribute("LastName","LIKE 'C%'");
vo.applyViewCriteria(vc);
vo.executeQuery();
vc.add(vcr1);
vo.executeQuery();
showRows(vo, "Initial database results with applied view criteria");
// 2. Subset results in memory to those with first name starting with 'J'
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
ViewCriteriaRow vcr2 = vc.createViewCriteriaRow();
vcr2.setAttribute("FirstName","LIKE 'J%'");
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_AND);
vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_CACHE);
vc.add(vcr2);
vo.executeQuery();
showRows(vo,"After augmenting view criteria and applying in-memory");
// 3. Set conjuction back to OR and re-apply to database query to find
// customers with last name like 'J%' or first name like 'C%'
vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_QUERY);
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES);
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_OR);
vo.executeQuery();
```

```
showRows(vo,"After changing view criteria and applying to database again");
// 4. Define new critera to find customers with first or last name like '%o%'
ViewCriteria nameContainsO = vo.createViewCriteria();
ViewCriteriaRow lastContainsO = nameContainsO.createViewCriteriaRow();
lastContainsO.setAttribute("LastName","LIKE '%o%'");
ViewCriteriaRow firstContainsO = nameContainsO.createViewCriteriaRow();
firstContainsO.setAttribute("FirstName","LIKE '%o%'");
nameContainsO.add(firstContainsO);
nameContainsO.add(lastContainsO);
// 5. Use findByViewCriteria() to produce new rowset instead of subsetting
nameContainsO.setCriteriaMode(ViewCriteria.CRITERIA_MODE_CACHE);
RowSet rs = (RowSet)vo.findByViewCriteria(nameContainsO,
                         -1,ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
showRows(rs,"Rows returned from in-memory findByViewCriteria");
// 6. Show that original rowset hasn't changed
showRows(vo,"Note findByViewCriteria didn't change rows in the view");
```

27.5.4 Performing In-Memory Filtering with RowMatch

The `RowMatch` object provides an even more convenient way to express in-memory filtering conditions. You create a `RowMatch` object by passing a query predicate expression to the constructor like this:

```
RowMatch rm =
 new RowMatch("LastName = 'Popp' or (FirstName like 'L%' and LastName like 'D%')")
;
```

As you do with the `SortBy` clause, you phrase the `RowMatch` expression in terms of the view object attribute names, using the supported operators like =, >, <, <=, >=, <>, and `LIKE`. You can group subexpressions with parenthesis and use the `and` and `or` operators between subexpressions.

> **Note:** While SQL query predicates treat column names in a case-insensitive way, the attribute names in a `RowMatch` expression are case-*sensitive*.

27.5.4.1 Applying a RowMatch to a View Object

To apply a `RowMatch` to your view object, call the `setRowMatch()` method. In contrast to a `ViewCriteria`, the `RowMatch` is only used for *in-memory* filtering, so there is no "match mode" to set. You can use a `RowMatch` on view objects in any supported query mode, and you will see the results of applying it the next time you call the `executeQuery()` method.

When you apply a `RowMatch` to a view object, the `RowMatch` expression can reference the view object's named bind variables using the same `:VarName` notation that you would use in a SQL statement. For example, if a view object had a named bind variable named `StatusCode`, you could apply a RowMatch to it with an expression like:

```
Status = :StatusCode or :StatusCode = '%'
```

Example 27–12 shows the interesting lines of a `TestClientRowMatch` class that illustrate the `RowMatch` in action. The CustomerList view object used in the example has a transient `Boolean` attribute named `Selected`. The code performs the following basic steps:

1. Queries the full customer list, producing the output:

```
--- Initial database results ---
Neena Kochhar [null]
Lex De Haan [null]
Nancy Greenberg [null]
:
```

2. Marks odd-numbered rows selected by setting the `Selected` attribute of odd rows to `Boolean.TRUE`, producing the output:

```
--- After marking odd rows selected ---
Neena Kochhar [null]
Lex De Haan [true]
Nancy Greenberg [null]
Daniel Faviet [true]
John Chen [null]
Ismael Sciarra [true]
:
```

3. Uses a `RowMatch` to subset the row set to contain only the select rows, that is, those with `Selected = true`. This produces the output:

```
--- After in-memory filtering on only selected rows ---
Lex De Haan [true]
Daniel Faviet [true]
Ismael Sciarra [true]
Luis Popp [true]
:
```

4. Further subsets the row set using a more complicated `RowMatch` expression, producing the output:

```
--- After in-memory filtering with more complex expression ---
Lex De Haan [true]
Luis Popp [true]
```

Example 27–12   Performing In-Memory Filtering with RowMatch

```
// In TestClientRowMatch.java
// 1. Query the full customer list
ViewObject vo = am.findViewObject("CustomerList");
vo.executeQuery();
showRows(vo,"Initial database results");
// 2. Mark odd-numbered rows selected by setting Selected = Boolean.TRUE
markOddRowsAsSelected(vo);
showRows(vo,"After marking odd rows selected");
// 3. Use a RowMatch to subset row set to only those with Selected = true
RowMatch rm = new RowMatch("Selected = true");
vo.setRowMatch(rm);
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
vo.executeQuery();
showRows(vo, "After in-memory filtering on only selected rows");
// 4. Further subset rowset using more complicated RowMatch expression
rm = new RowMatch("LastName = 'Popp' "+
                  "or (FirstName like 'L%' and LastName like 'D%')");
vo.setRowMatch(rm);
vo.executeQuery();
showRows(vo,"After in-memory filtering with more complex expression");
```

```
// 5. Remove RowMatch, set query mode back to database, requery to see full list
vo.setRowMatch(null);
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES);
vo.executeQuery();
showRows(vo,"After re-querying to see a full list again");
```

27.5.4.2 Using RowMatch to Test an Individual Row

In addition to using a RowMatch to filter a row set, you can also use its rowQualifies() method to test whether any individual row matches the criteria it encapsulates. For example:

```
RowMatch rowMatch = new RowMatch("CountryId = 'US'");
if (rowMatch.rowQualifies(row)) {
  System.out.println("Customer is from the United States ");
}
```

27.5.4.3 How a RowMatch Affects Rows Fetched from the Database

Once you apply a RowMatch, if the view object's query mode is set to retrieve rows from the database, when you call executeQuery() the RowMatch is applied to rows as they are fetched. If a fetched row does not qualify, it is not added to the rowset.

Unlike a SQL WHERE clause, a RowMatch can evaluate expressions involving transient view object attributes and not-yet-posted attribute values. This can be useful to filter queried rows based on RowMatch expressions involving transient view row attributes whose values are calculated in Java. This interesting aspect should be used with care, however, if your application needs to process a large rowset. Oracle recommends using database-level filtering to retrieve the smallest-possible rowset first, and then using RowMatch as appropriate to subset that list in memory.

27.6 Using View Objects to Work with Multiple Row Types

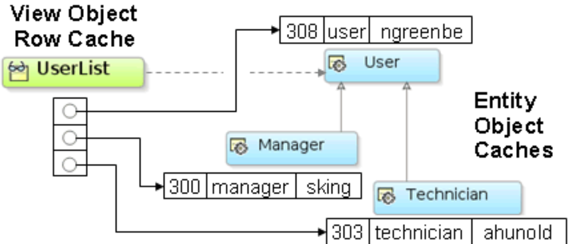
In Section 26.6, "Using Inheritance in Your Business Domain Layer" you saw how to create an inheritance hierarchy of User, Technician, and Manager entity objects. Sometimes you will create a view object to work with entity rows of a single type like Technician, which perhaps includes Technician-specific attributes. At other times you may want to query and update rows for users, technicians, *and* managers in the same row set, working with attributes that they all share in common.

Note: To experiment with the example described in this section, use the same InheritanceAndPolymorphicQueries project in the AdvancedEntityExamples workspace used in Section 26.6, "Using Inheritance in Your Business Domain Layer".

27.6.1 What is a Polymorphic Entity Usage?

A *polymorphic* entity usage is one that references a base entity object in an inheritance hierarchy and is configured to handle *subtypes* of that entity as well. Figure 27–4 shows the results of using a view object with a polymorphic entity usage. The entity-based UserList view object has the User entity object as its primary entity usage. The view object partitions each row retrieved from the database into an entity row part of the appropriate entity object *subtype* of User. It creates the appropriate entity row subtype based on consulting the value of the discriminator attribute. For example, if the UserList query retrieves one row for user ngreenbe, one row for manager sking, and one row for technician ahunold, the underlying entity row parts would be as shown in the figure.

Figure 27–4 View Object with a Polymorphic Entity Usage Handles Entity Subtypes



27.6.2 How To Create a View Object with a Polymorphic Entity Usage

To create a view object with a polymorphic entity usage, follow these steps:

1. Identify the entity object that represents the base type in the entity inheritance hierarchy you want to work with.
2. Create an entity-based view object with that base entity as its entity usage.

> **Note:** When an entity-based view object references an entity object with a discriminator attribute, then JDeveloper enforces that the discriminator attribute is included in the query as well (in addition to the primary key attribute).

3. On the **Entity Objects** panel of the Create View Object wizard, select the entity usage in the **Selected** list and click **Subtypes...**

- In the **Subtypes** dialog that appears, shuttle the desired entity subtypes you want to allow from the **Available** to the **Selected** list, and click **OK**

Then click **OK** to create the view object.

27.6.3 What Happens When You Create a View Object with a Polymorphic Entity Usage

When you create an entity-based view object with a polymorphic entity usage, JDeveloper adds information about the allowed entity subtypes to the view object's XML component definition. For example, when creating the `UserList` view object above, the names of the allowed subtype entity objects are recorded in an <AttrArray> tag like this:

```
<ViewObject Name="UserList" ... >
   <EntityUsage Name="TheUser"
                Entity="devguide.advanced.inheritance.entities.User" >
   </EntityUsage>
   <AttrArray Name="EntityImports">
      <Item Value="devguide.advanced.inheritance.entities.Manager" />
      <Item Value="devguide.advanced.inheritance.entities.Technician" />
   </AttrArray>
   <!-- etc. -->
</ViewObject>
```

27.6.4 What You May Need to Know

27.6.4.1 Your Query Must Limit Rows to Expected Entity Subtypes

If your view object expects to work with only a *subset* of the available entity subtypes in a hierarchy, you need to include an appropriate WHERE clause that limits the query to only return rows whose discriminator column matches the expected entity types.

27.6.4.2 Exposing Selected Entity Methods in View Rows Using Delegation

By design, clients do not work directly with entity objects. Instead, they work indirectly with entity objects through the view rows of an appropriate view object that presents a relevant set of information related to the task as hand. Just as a view object can expose a particular set of the underlying *attributes* of one or more entity objects related to the task at hand, it can also expose a selected set of *methods* from those entities. You accomplish this by enabling a custom view row Java class and writing a method in the view row class that:

- Accesses the appropriate underlying entity row using the generated entity accessor in the view row, and
- Invokes a method on it

For example, assume that the `User` entity object contains a `performUserFeature()` method in its `UserImpl` class. To expose this method to clients on the `UserList` view row, you can enable a custom view row Java class and write the method shown in Example 27–13. JDeveloper generates an entity accessor method in the view row class for each participating entity usage based on the entity usage *alias* name. Since the alias for the `User` entity in the `UserList` view object is "`TheUser`", it generates a `getTheUser()` method to return the entity row part related to that entity usage.

Example 27–13 Exposing Selected Entity Object Methods on View Rows Through Delegation

```
// In UserListRowImpl.java
public void performUserFeature() {
  getTheUser().performUserFeature();
}
```

The code in the view row's `performUserFeature()` method uses this `getTheUser()` method to access the underlying `UserImpl` entity row class and then invokes *its* `performUserFeature()` method. This style of coding is known as *delegation*, where a view row method delegates the implementation of one of *its* methods to a corresponding method on an underlying entity object. When delegation is used in a view row with a polymorphic entity usage, the delegated method call is handled by appropriate underlying entity row subtype. This means that if the `UserImpl`, `ManagerImpl`, and `TechnicianImpl` classes implement the `performUserFeature()` method in a different way, the appropriate implementation is used depending on the entity subtype for the current row.

After exposing this method on the client row interface, client programs can use the custom row interface to invoke custom business functionality on a particular view row. Example 27–14 shows the interesting lines of code from a `TestEntityPolymorphism` class. It iterates over all the rows in the `UserList` view object instance, casts each one to the custom `UserListRow` interface, and invokes the `performUserFeature()` method.

Example 27–14 Invoking a View Row Method That Delegates to an Entity Object

```
UserList userlist = (UserList)am.findViewObject("UserList");
userlist.executeQuery();
while (userlist.hasNext()) {
  UserListRow user = (UserListRow)userlist.next();
  System.out.print(user.getEmail()+"->");
  user.performUserFeature();
}
```

Running the client code in Example 27–14 produces the following output:

```
austin->## performUserFeature as Technician
hbaer->## performUserFeature as User
:
sking->## performUserFeature as Manager
:
```

Rows related to `User` entities display a message confirming that the `performUserFeature()` method in the `UserImpl` class was used. Rows related to `Technician` and `Manager` entities display a different message, highlighting the different implementations that the respective TechnicianImpl and ManagerImpl classes have for the inherited `performUserFeature()` method.

27.6.4.3 Creating New Rows With the Desired Entity Subtype

In a view object with a polymorphic entity usage, when you create a new view row it contains a new entity row part whose type matches the base entity usage. To create a new view row with one of the entity *subtypes* instead, use the `createAndInitRow()` method. Example 27–15 shows two custom methods in the `UserList` view object's Java class that use `createAndInitRow()` to allow a client to create new rows having entity rows either of `Manager` or `Technician` subtypes. To use the `createAndInitRow()`, as shown in the example, create an instance of the `NameValuePairs` object and set it to have an appropriate value for the discriminator attribute. Then, pass that `NameValuePairs` to the `createAndInitRow()` method to create a new view row with the appropriate entity row subtype, based on the value of the discriminator attribute you passed in.

Example 27–15 Exposing Custom Methods to Create New Rows with Entity Subtypes

```
// In UserListImpl.java
public UserListRow createManagerRow() {
  NameValuePairs nvp = new NameValuePairs();
  nvp.setAttribute("UserRole","manager");
  return (UserListRow)createAndInitRow(nvp);
}
public UserListRow createTechnicianRow() {
  NameValuePairs nvp = new NameValuePairs();
  nvp.setAttribute("UserRole","technician");
  return (UserListRow)createAndInitRow(nvp);
}
```

If you expose methods like this on the view object's custom interface, then at runtime, a client can call them to create new view rows with appropriate entity subtypes. Example 27–16 shows the interesting lines relevant to this functionality from a `TestEntityPolymorphism` class. First, it uses the `createRow()`, `createManagerRow()`, and `createTechnicianRow()` methods to create three new view rows. Then, it invokes the `performUserFeature()` method from the `UserListRow` custom interface on each of the new rows.

As expected, each row handles the method in a way that is specific to the subtype of entity row related to it, producing the results:

```
## performUserFeature as User
## performUserFeature as Manager
## performUserFeature as Technician
```

Example 27–16 Creating New View Rows with Different Entity Subtypes

```
// In TestEntityPolymorphism.java
UserListRow newUser = (UserListRow)userlist.createRow();
UserListRow newMgr  = userlist.createManagerRow();
UserListRow newTech = userlist.createTechnicianRow();
newUser.performUserFeature();
newMgr.performUserFeature();
newTech.performUserFeature();
```

27.6.5 What are Polymorphic View Rows?

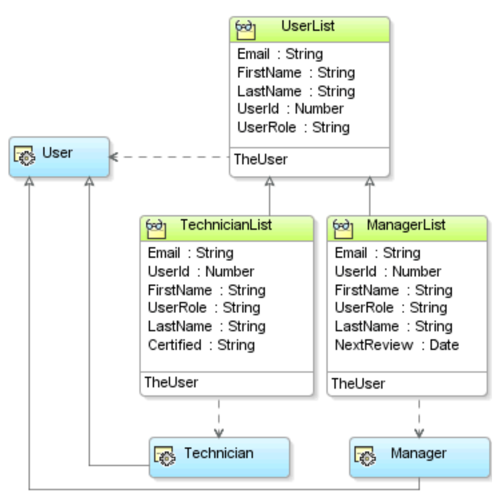
In the example shown in Section 27.6, "Using View Objects to Work with Multiple Row Types", the polymorphism occurs "behind the scenes" at the entity object level. Since the client code works with all view rows using the same `UserListRow` interface, it cannot distinguish between rows based on a `Manager` entity object from those based on a `User` entity object. The code works with all view rows using the same set of view row attributes and methods common to all types of underlying entity subtypes.

If you configure a view object to support polymorphic view rows, then the client can work with different types of view rows using a view row interface specific to the type of row it is. By doing this, the client can access view attributes or invoke view row methods that are specific to a given subtype as needed. Figure 27–5 illustrates the hierarchy of view objects that enables this feature for the `UserList` example considered above. `TechnicianList` and `ManagerList` are view objects that extend the base `UserList` view object. Notice that each one includes an additional attribute specific to the subtype of `User` they have as their entity usage. `TechnicianList` includes an additional `Certified` attribute, while `ManagerList` includes the additional `NextReview` attribute. When configured for view row polymorphism as described in the next section, a client can work with the results of the `UserList` view object using:

- `UserListRow` interface for view rows related to
- `TechnicianListRow` interface for view rows related to technicians
- `ManagerListRow` interface for view rows related to managers

As you'll see, this allows the client to access the additional attributes and view row methods that are specific to a given subtype of view row.

Figure 27–5 Hierarchy of View Object Subtypes Enables View Row Polymorphism



27.6.6 How to Create a View Object with Polymorphic View Rows

To create a view object with polymorphic view rows, follow these steps:

1. Identify an existing view object to be the base

   In the example above, the `UserList` view object is the base.

2. Identify a discriminator attribute for the view row, and give it a default value.

   Check the **Discriminator** checkbox on the attribute panel to mark the attribute as the one that distinguishes which view row interface to use. You must supply a value for the **Default** field that matches the attribute value for which you expect the base view object's view row interface to be used. For example, in the `UserList` view object, you would mark the `UserRole` attribute as the discriminator attribute and supply a default value of "`user`".

3. Enable a custom view row class for the base view object, and expose at least one method on the client row interface. This can be one or all of the view row attribute accessor methods, as well as any custom view row methods.

4. Create a new view object that *extends* the base view object

   In the example above, `TechnicianList` extends the base `UserList` view object.

5. Enable a custom view row class for the extended view object.

   If appropriate, add additional custom view row methods or override custom view row methods inherited from the parent view object's row class.

6. Supply a *distinct* value for the discriminator attribute in the extended view object.

   The `TechnicianList` view object provides the value of "`technician`" for the `UserRole` discriminator attribute.

7. Repeat steps 4-6 to add additional extended view objects as needed.

   For example, the `ManagerList` view object is a second one that extends `UserList`. It supplies the value "`manager`" for the `UserRole` discriminator attribute.

After setting up the view object hierarchy, you need to define the list of view object subtypes that participate in the view row polymorphism. To accomplish this, do the following:

1. Add an instance of each type of view object in the hierarchy to the data model of an application module.

   For example, the `UserModule` application module in the example has instances of `UserList`, `TechnicianList`, and `ManagerList` view objects.

2. In the **Data Model** panel of the Application Module Editor, click **Subtypes...**

3. In the **Subtypes** dialog that appears, shuttle the desired view object subtypes that you want to participate in view row polymorphism from the **Available** to the **Selected** list, and click **OK**

27.6.7 What You May Need to Know

27.6.7.1 Selecting Subtype-Specific Attributes in Extended View Objects

When you create an extended view object, it inherits the entity usage of its parent. If the parent view object's entity usage is based on an entity object with subtypes in your domain layer, you may want your extended view object to work with one of these subtypes instead of the inherited parent entity usage type. Two reasons you might want to do this are:

1. To select attributes that are *specific* to the entity subtype

2. To be able to write view row methods that delegate to methods *specific* to the entity subtype

In order to do this, you need to override the inherited entity usage to refer to the desired entity subtype. To do this, perform these steps in the View Object Editor for your extended view object:

1. On the **Entity Objects** panel, verify that you are working with an extended entity usage.

   For example, when creating the `TechnicianList` view object that extends the `UserList` view object, the entity usage with the alias `TheUser` will initially display in the **Selected** list as: **TheUser (User): extended**. The type of the entity usage is in parenthesis, and the "extended" label confirms that the entity usage is currently inherited from its parent.

2. Select the desired entity *subtype* in the **Available** list that you want to override the inherited one. It must be a subtype entity of the existing entity usage's type.

   For example, you would select the `Technician` entity object in the **Available** list to overridden the inherited entity usage based on the `User` entity type.

3. Click **>** to shuttle it to the **Selected** list

4. Acknowledge the alert that appears, confirming that you want to override the existing, inherited entity usage.

When you have performed these steps, the **Selected** list updates to reflect the overridden entity usage. For example, for the `TechnicianList` view object, after overriding the `User`-based entity usage with the `Technician` entity subtype, it updates to show: **TheUser (Technician): overridden**.

After overriding the entity usage to be related to an entity subtype, you can then use the **Attributes** tab of the editor to select additional attributes that are specific to the subtype. For example, the `TechnicianList` view object includes the additional attribute named `Certified` that is specific to the `Technician` entity object.

27.6.7.2 Delegating to Subtype-Specific Methods After Overriding the Entity Usage

After overriding the entity usage in an extended view object to reference a subtype entity, you can write view row methods that delegate to methods specific to the subtype entity class. Example 27–17 shows the code for a `performTechnicianFeature()` method in the custom view row class for the `TechnicianList` view object. It casts the return value from the `getTheUser()` entity row accessor to the subtype `TechnicianImpl`, and then invokes the `performTechnicianFeature()` method that is specific to `Technician` entity objects.

Example 27–17 View Row Method Delegating to Method in Subtype Entity

```
// In TechnicianListRowImpl.java
public void performTechnicianFeature() {
   TechnicianImpl tech = (TechnicianImpl)getTheUser();
   tech.performTechnicianFeature();
}
```

Note: You need to perform the explicit cast to the entity subtype here because JDeveloper does not yet take advantage of the new JDK 5.0 feature called covariant return types that would allow a subclass like `TechnicianListRowImpl` to override a method like `getTheUser()` and change its return type.

27.6.7.3 Working with Different View Row Interface Types in Client Code

Example 27–18 shows the interesting lines of code from a `TestViewRowPolymorphism` class that performs the following steps:

1. Iterates over the rows in the `UserList` view object.

   For each row in the loop, it uses Java's `instanceof` operator to test whether the current row is an instance of the `ManagerListRow` or the `TechnicianListRow`.

2. If the row is a `ManagerListRow`, then cast it to this more specific type and:

   - Call the `performManagerFeature()` method specific to the `ManagerListRow` interface, and
   - Access the value of the `NextReview` attribute that is specific to the `ManagerList` view object.

3. If the row is a `TechnicianListRow`, then cast it to this more specific type and:

   - Call the `performTechnicianFeature()` method specific to the `TechnicianListRow` interface, and
   - Access the value of the `Certified` attribute that is specific to the `TechnicianList` view object.

4. Otherwise, just call a method on the `UserListRow`

Example 27–18 Using View Row Polymorphism in Client Code

```
// In TestViewRowPolymorphism.java
ViewObject vo = am.findViewObject("UserList");
vo.executeQuery();
// 1. Iterate over the rows in the UserList view object
while (vo.hasNext()) {
  UserListRow user = (UserListRow)vo.next();
  System.out.print(user.getEmail()+"->");
  if (user instanceof ManagerListRow) {
    // 2. If the row is a ManagerListRow, cast it
    ManagerListRow mgr = (ManagerListRow)user;
    mgr.performManagerFeature();
    System.out.println("Next Review:"+mgr.getNextReview());
  }
  else if (user instanceof TechnicianListRow) {
    // 3. If the row is a ManagerListRow, cast it
    TechnicianListRow tech = (TechnicianListRow)user;
    tech.performTechnicianFeature();
    System.out.println("Certified:"+tech.getCertified());
  }
  else {
    // 4. Otherwise, just call a method on the UserListRow
    user.performUserFeature();
  }
}
```

Running the code in Example 27–18 produces the following output:

```
daustin->## performTechnicianFeature called
Certified:Y
hbaer->## performUserFeature as User
:
sking->## performManagerFeature called
Next Review:2006-05-09
:
```

This illustrates that by using the view row polymorphism feature the client was able to distinguish between view rows of different types and access methods and attributes specific to each subtype of view row.

27.6.7.4 View Row Polymorphism and Polymorphic Entity Usage are Orthogonal

While often even more useful when used *together*, the view row polymorphism and the polymorphic entity usage features are distinct and can be used separately. In particular, the view row polymorphism feature can be used for read-only view objects, as well as for entity-based view objects. When you combine both mechanisms, you can have both the entity row part being polymorphic, as well as the view row type.

Note to use view row polymorphism with either view objects or entity objects, you must configure the discriminator attribute property separately for each. This is necessary because read-only view objects contain no related entity usages from which to infer the discriminator information.

In summary, to use view row polymorphism:

1. Configure an attribute to be the discriminator at the view object level in the root view object in an inheritance hierarchy.
2. Have a hierarchy of inherited view objects each of which provides a distinct value for the "Default Value" property of that view object level discriminator attribute.

3. List the subclassed view objects in this hierarchy in the application module's list of Subtypes.

Whereas, to create a view object with a polymorphic entity usage:

1. Configure an attribute to be the discriminator at the entity object level in the root entity object in an inheritance hierarchy.
2. Have a hierarchy of inherited entity objects, each of which overrides and provides a distinct value for the "Default Value" property of that entity object level discriminator attribute.
3. List the subclassed entity objects in a view object's list of Subtypes.

27.7 Reading and Writing XML

The Extensible Markup Language (XML) standard from the Worldwide Web Consortium (W3C) defines a language-neutral approach for electronic data exchange. Its rigorous set of rules enables the structure inherent in data to be easily encoded and unambiguously interpreted using human-readable text documents.

View objects support the ability to write these XML documents based on their queried data. View objects also support the ability to read XML documents in order to apply changes to data including inserts, updates, and deletes. When you've introduced view links, this XML capability supports reading and writing multi-level nested information for master/detail hierarchies of any complexity. While the XML produced and consumed by view objects follows a canonical format, you can combine the view object's XML features with XML Stylesheet Language Transformations (XSLT) to easily convert between this canonical XML format and any format you need to work with.

Note: The examples in this section refer to the `ReadingAndWritingXML` project in the `AdvancedViewObjectExamples` workspace. See the note at the beginning of this chapter for download instructions.

27.7.1 How to Produce XML for Queried Data

To produce XML from a view object, use the `writeXML()` method. If offers two ways to control the XML produced:

1. For precise control over the XML produced, you can specify a view object attribute map indicating which attributes should appear, including which view link accessor attributes should be accessed for nested, detail information:

   ```
   Node writeXML(long options, HashMap voAttrMap)
   ```

2. To producing XML that includes all attributes, you can simply specify a depth-level that indicates how many levels of view link accessor attributes should be traversed to produce the result:

   ```
   Node writeXML(int depthCount, long options)
   ```

The `options` parameter is a integer flag field that can be set to one of the following bit flags:

- `XMLInterface.XML_OPT_ALL_ROWS`

  Includes all rows in the view object's row set in the XML.

- `XMLInterface.XML_OPT_LIMIT_RANGE`

  Includes only the rows in the current range in the XML.

Using the logical OR operation, you can combine either of the above flags with the `XMLInterface.XML_OPT_ASSOC_CONSISTENT` flag when you want to include new, unposted rows in the current transaction in the XML output.

Both versions of the `writeXML()` method accept an optional third argument which is an XSLT stylesheet that, if supplied, is used to transform the XML output before returning it.

27.7.2 What Happens When You Produce XML

When you produce XML using `writeXML()`, the view object begins by creating a wrapping XML element whose default name matches the name of the view object definition. For example, for a `Users` view object in the `devguide.advanced.xml.queries` package, the XML produces will be wrapped in an outermost <Users> tag.

Then, it converts the attribute data for the appropriate rows into XML elements. By default, each row's data is wrapped in an row element whose name is the name of the view object with the `Row` suffix. For example, each row of data from a view object named `Users` is wrapped in an <UsersRow> element. The elements representing the attribute data for each row appear as nested children inside this row element.

If any of the attributes is a view link accessor attribute, and if the parameters passed to `writeXML()` enable it, the view object will include the data for the detail rowset returned by the view link accessor. This nested data is wrapped by an element whose name is determined by the name of the view link accessor attribute. The return value of the `writeXML()` method is an object that implements the standard W3C `Node` interface, representing the root element of the generated XML.

> **Note:** The `writeXML()` method uses view link accessor attributes to programmatically access detail collections. It does not require adding view link instances in the data model.

For example, to produce an XML element for all rows of a `Users` view object instance, and following view link accessors as many levels deep as exists, Example 27–19 shows the code required.

Example 27–19 Generating XML for All Rows of a View Object to All View Link Levels

```
ViewObject vo = am.findViewObject("Users");
printXML(vo.writeXML(-1,XMLInterface.XML_OPT_ALL_ROWS));
```

The `Users` view object is linked to a `ServiceRequests` view object showing the service requests created by that user. In turn, the `ServiceRequests` view object is linked to a `ServiceHistories` view object providing details on the notes entered for the service request by customers and technicians. Running the code in Example 27–19 produces the XML shown in Example 27–20, reflecting the nested structure defined by the view links.

Example 27–20 XML from a Users View Object with Two Levels of View Linked Details

```
<Users>
    :
   <User>
      <UserId>316</UserId>
      <UserRole>user</UserRole>
      <EmailAddress>sbaida</EmailAddress>
      <FirstName>Shelli</FirstName>
      <LastName>Baida</LastName>
      <StreetAddress>4715 Sprecher Rd</StreetAddress>
      <City>Madison</City>
      <StateProvince>Wisconsin</StateProvince>
      <PostalCode>53704</PostalCode>
      <CountryId>US</CountryId>
      <UserRequests>
         <ServiceRequestsRow>
            <SvrId>101</SvrId>
            <Status>Closed</Status>
            <RequestDate>2006-04-16 13:32:54.0</RequestDate>
            <ProblemDescription>Agitator does not work</ProblemDescription>
            <ProdId>101</ProdId>
            <CreatedBy>316</CreatedBy>
            <AssignedTo>304</AssignedTo>
            <AssignedDate>2006-04-23 13:32:54.0</AssignedDate>
            <ServiceHistories>
               <ServiceHistoriesRow>
                  <SvrId>101</SvrId>
                  <LineNo>1</LineNo>
                  <SvhDate>2006-04-23 13:32:54.0</SvhDate>
                  <Notes>Asked customer to ensure the lid was closed</Notes>
                  <SvhType>Technician</SvhType>
                  <CreatedBy>304</CreatedBy>
               </ServiceHistoriesRow>
               <ServiceHistoriesRow>
                  <SvrId>101</SvrId>
                  <LineNo>2</LineNo>
                  <SvhDate>2006-04-24 13:32:54.0</SvhDate>
                  <Notes>Problem is fixed</Notes>
                  <SvhType>Customer</SvhType>
                  <CreatedBy>316</CreatedBy>
               </ServiceHistoriesRow>
            </ServiceHistories>
         </ServiceRequestsRow>
      </UserRequests>
   </User>
   :
</Users>
```

27.7.3 What You May Need to Know

27.7.3.1 Controlling XML Element Names

You can change the default XML element names used in the view object's canonical XML format by setting custom properties:

- Set the custom *attribute*-level property named `XML_ELEMENT` to a value *`SomeOtherName`* to change the XML element name used for that attribute to <SomeOtherName>

  For example, the `Email` attribute in the `Users` view object defines this property to change the XML element you see in Example 27–20 to be <EmailAddress> instead of <Email>.

- Set the custom view object-level property named `XML_ROW_ELEMENT` to a value *`SomeOtherRowName`* to change the XML element name used for that attribute to <SomeOtherRowName>

  For example, the `Users` view object defines this property to change the XML element name for the rows you see in Example 27–20 to be <User> instead of <UsersRow>.

- To change the name of the element names that wrapper nested row set data from view link attribute accessors, you need to use the **View Link Properties** panel of the View Link Editor to change the name of the view link accessor attribute.

27.7.3.2 Controlling Element Suppression for Null-Valued Attributes

By default, if a view row attribute is `null`, then its corresponding element is omitted from the generated XML. You can set the custom attribute-level property named `XML_EXPLICIT_NULL` to any value (e.g. "`true`" or "`yes`") to cause an element to be included for the attribute if its value is null. For example, if an attribute named `AssignedDate` has this property set, then a row containing a `null` assigned date will contain a corresponding <AssignedDate null="true"/> element. If you want this behavior for all attributes of a view object, you can define the `XML_EXPLICIT_NULL` custom property at the view object level as a shortcut for defining it on each attribute.

27.7.3.3 Printing or Searching the Generated XML Using XPath

Two of the most common things you might want to do with the XML `Node` object returned from `writeXML()` are:

1. Printing the node to its serialized text representation — to send across the network or save in a file, for example
2. Searching the generated XML using W3C XPath expressions

Unfortunately, the standard W3C Document Object Model (DOM) API does not include methods for doing *either* of these useful operations. But there is hope. Since ADF Business Components uses the Oracle XML parser's implementation of the DOM, you can cast the `Node` return value from `writeXML()` to the Oracle specific classes `XMLNode` or `XMLElement` (in the `oracle.xml.parser.v2` package) to access additional useful functionality like:

- Printing the XML element to its serialized text format using the `print()` method
- Searching the XML element in memory with XPath expressions using the `selectNodes()` method
- Finding the value of an XPath expression related to the XML element using the `valueOf()` method.

Example 27–21 shows the `printXML()` method in the `TestClientWriteXML`. It casts the `Node` parameter to an XMLNode and calls the `print()` method to dump the XML to the console.

Example 27–21 Using the XMLNode's print() Method to Serialize XML

```
// In TestClientWriteXML.java
private static void printXML(Node n) throws IOException {
  ((XMLNode)n).print(System.out);
}
```

27.7.3.4 Using the Attribute Map For Fine Control Over Generated XML

When you need fine control over which attributes appear in the generated XML, use the version of the `writeXML()` method that accepts a `HashMap`. Example 27–22 shows the interesting lines from a `TestClientWriteXML` class that use this technique. After creating the `HashMap`, you put `String[]`-valued entries into it containing the names of the attributes you want to include in the XML, keyed by the fully-qualified name of the view definition those attributes belong to. The example includes the `UserId`, `Email`, `StateProvince`, and `UserRequests` attributes from the `Users` view object, and the `SvrId`, `Status`, `AssignedDate`, and `ProblemDescription` attributes from the `ServiceRequests` view object.

> **Note:** For upward compatibility reasons with earlier versions of ADF Business Components the `HashMap` expected by the `writeXML()` method is the one in the `com.sun.java.util.collections` package.

While processing the view rows for a given view object instance:

- If an entry exists in the attribute map with a key matching the fully-qualified view definition name for that view object, then only the attributes named in the corresponding `String` array are included in the XML.

  Furthermore, if the string array includes the name of a view link accessor attribute, then the nested contents of its detail row set are included in the XML. If a view link accessor attribute name does not appear in the string array, then the contents of its detail row set are not included.

- If no such entry exists in the map, then *all* attributes for that row are included in the XML.

Example 27–22 Using a View Definition Attribute Map for Fine Control Over Generated XML

```
HashMap viewDefMap = new HashMap();
viewDefMap.put("devguide.advanced.xml.queries.Users",
        new String[]{"UserId",
                     "Email",
                     "StateProvince",
                     "UserRequests" /* View link accessor attribute */
                     });
viewDefMap.put("devguide.advanced.xml.queries.ServiceRequests",
        new String[]{"SvrId","Status","AssignedDate","ProblemDescription"});
printXML(vo.writeXML(XMLInterface.XML_OPT_ALL_ROWS,viewDefMap));
```

Running the example produces the XML shown in Example 27–23, including only the exact attributes and view link accessors indicated by the supplied attribute map.

Example 27–23 XML from a Users View Object Produced Using an Attribute Map

```
<Users>
   <User>
      <UserId>300</UserId>
      <EmailAddress>sking</EmailAddress>
      <StateProvince>Washington</StateProvince>
      <UserRequests>
         <ServiceRequestsRow>
            <SvrId>200</SvrId>
            <Status>Open</Status>
            <AssignedDate null="true"/>
            <ProblemDescription>x</ProblemDescription>
         </ServiceRequestsRow>
      </UserRequests>
   </User>
   <User>
      <UserId>301</UserId>
      <EmailAddress>nkochhar</EmailAddress>
      <StateProvince>Maryland</StateProvince>
   </User>
   :
</Users>
```

27.7.3.5 Use the Attribute Map Approach with Bi-Directional View Links

If your view objects are related through a view link that you have configured to be bi-directional, then you must use the `writeXML()` approach that uses the attribute map. If you were to use the `writeXML()` approach in the presence of bi-directional view links and were to supply a maximum depth of `-1` to include all levels of view links that exist, the `writeXML()` method will go into an infinite loop as it follows the bi-directional view links back and forth, generating deeply nested XML containing duplicate data until it runs out of memory. Use `writeXML()` with an attribute map instead in this situation. Only by using this approach can you control which view link accessors are included in the XML and which are not to avoid infinite recursion while generating the XML.

27.7.3.6 Transforming Generated XML Using an XSLT Stylesheet

When the canonical XML format produced by `writeXML()` does not meet your needs, you can supply an XSLT stylesheet as an optional argument. It will produce the XML as it would normally, but then transform that result using the supplied stylesheet before returning the final XML to the caller.

Consider the XSLT stylesheet shown in Example 27–24. It is a simple transformation with a single template that matches the root element of the generated XML from Example 27–23 to create a new <CustomerEmailAddresses> element in the result. The template uses the <xsl:for-each> instruction to process all <User> element children of <Users> that contain more than one <ServiceRequestsRow> child element inside a nested <UserRequests> element. For each <User> element that qualifies, it creates a <Customer> element in the result whose `Contact` attribute is populated from the value of the <EmailAddress> child element of the <User>.

Example 27–24 XSLT Stylesheet to Transform Generated XML Into Another Format

```
<?xml version="1.0" encoding="windows-1252" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <CustomerEmailAddresses>
      <xsl:for-each
           select="/Users/User[count(UserRequests/ServiceRequestsRow) > 1]">
        <xsl:sort select="EmailAddress"/>
        <Customer Contact="{EmailAddress}"/>
      </xsl:for-each>
    </CustomerEmailAddresses>
  </xsl:template>
</xsl:stylesheet>
```

Example 27–25 shows the interesting lines from a `TestClientWriteXML` class that put this XSLT stylesheet into action when calling `writeXML()`.

Example 27–25 Passing an XSLT Stylesheet to writeXML() to Transform the Resulting XML

```
// In TestClientWriteXML.java
XSLStylesheet xsl = getXSLStylesheet();
printXML(vo.writeXML(XMLInterface.XML_OPT_ALL_ROWS,viewDefMap,xsl));
```

Running the code in Example 27–25 produces the transformed XML shown here:

```
<CustomerEmailAddresses>
   <Customer Contact="dfaviet"/>
   <Customer Contact="jchen"/>
   <Customer Contact="ngreenbe"/>
</CustomerEmailAddresses>
```

The `getXSLStylesheet()` helper method shown in Example 27–26 is also interesting to study since it illustrates how to read a resource like an XSLT stylesheet from the classpath at runtime. The code expects the `Example.xsl` stylesheet to be in the same directory as the `TestClientWriteXML` class. By referencing the `Class` object for the `TestClientWriteXML` class using the `.class` operator, the code uses the `getResource()` method to get a `URL` to the resource. Then, it passes the URL to the `newXSLStylesheet()` method of the `XSLProcessor` class to create a new `XSLStylesheet` object to return. That object represents the compiled version of the XSLT stylesheet read in from the `*.xsl`file.

Example 27–26 Reading an XSLT Stylesheet as a Resource from the Classpath

```
private static XSLStylesheet getXSLStylesheet()
        throws XMLParseException, SAXException,IOException,XSLException {
  String xslurl = "Example.xsl";
  URL xslURL = TestClientWriteXML.class.getResource(xslurl);
  XSLProcessor xslProc = new XSLProcessor();
  return xslProc.newXSLStylesheet(xslURL);
}
```

Note: When working with resources like XSLT stylesheets that you want to be included in the output directory along with your compiled Java classes and XML metadata, you can use the **Compiler** page of the **Project Properties** dialog to update the **Copy File Types to Output Directory** field to include `.xsl` in the semicolon-separated list.

27.7.3.7 Generating XML for a Single Row

In addition to calling `writeXML()` on a view object, you can call the same method with the same parameters and options on any `Row` as well. If the `Row` object on which you call `writeXML()` is a entity row, you can bitwise-OR the additional `XMLInterface.XML_OPT_CHANGES_ONLY` flag if you only want the changed entity attributes to appear in the XML.

27.7.4 How to Consume XML Documents to Apply Changes

To have a view object consume an XML document to process inserts, updates, and deletes, use the `readXML()` method:

```
void readXML(Element elem, int depthcount)
```

The canonical format expected by `readXML()` is the same as what would be produced by a call to the `writeXML()` method on the same view object. If the XML document to process does not correspond to this canonical format, you can supply an XSLT stylesheet as an optional third argument to `readXML()` to transform the incoming XML document *into* the canonical format before it is read for processing.

27.7.5 What Happens When You Consume XML Documents

When a view object consumes an XML document in canonical format, it processes the document to recognize row elements, their attribute element children, and any nested elements representing view link accessor attributes. It processes the document recursively to a maximum level indicated by the `depthcount` parameter. Passing `-1` for the `depthcount` to request that it process all levels of the XML document.

27.7.5.1 How ViewObject.readXML() Processes an XML Document

For each row element it recognizes, the `readXML()` method does the following:

- Identifies the related view object to process the row.
- Reads the children attribute elements to get the values of the primary key attributes for the row.
- Performs a `findByKey()` using the primary key attributes to detect whether the row already exists or not.

- If the row exists:
  - If the row element contains the marker attribute `bc4j-action="remove"`, then the existing row is deleted.
  - Otherwise, the row's attributes are updated using the values in any attribute element children of the current row element in the XML
- If the row does not exist, then a new row is created, inserted into the view object's rowset. Its attributes are populated using the values in any attribute element children of the current row element in the XML.

27.7.5.2 Using readXML() to Processes XML for a Single Row

The same `readXML()` method is also supported on any `Row` object. The canonical XML format it expects is the same format produced by a call to `writeXML()` on the same row. You can invoke `readXML()` method on a row to:

- Update its attribute values from XML
- Remove the row, if the `bc4j-action="remove"` marker attribute is present on the corresponding row element.
- Insert, update, or delete any nested rows via view link accessors

Consider the XML document shown in Example 27–27. It is in the canonical format expected by a single row in the `Technicians` view object. Nested inside the root <TechniciansRow> element, the <City> attribute represents the technician's city. The nested <ExpertiseAreas> element corresponds to the `ExpertiseAreas` view link accessor attribute and contains three <ExpertiseAreasRow> elements. Each of these includes <ProdId> elements representing part of the two-attribute primary key of a `ExpertiseAreas` row. The other primary key attribute, `UserId`, is inferred from the enclosing parent <TechniciansRow> element.

Example 27–27 XML Document in Canonical Format to Insert, Update, and Delete Rows

```
<TechniciansRow>
   <!-- This will update Techncian's City attribute -->
   <City>Padova</City>
   <ExpertiseAreas>
      <!-- This will be an update since it does exist -->
      <ExpertiseAreasRow>
         <ProdId>100</ProdId>
         <ExpertiseLevel>Expert</ExpertiseLevel>
      </ExpertiseAreasRow>
      <!-- This will be an insert since it doesn't exist -->
      <ExpertiseAreasRow>
         <ProdId>110</ProdId>
         <ExpertiseLevel>Expert</ExpertiseLevel>
      </ExpertiseAreasRow>
      <!-- This will be deleted -->
      <ExpertiseAreasRow bc4j-action="remove">
         <ProdId>112</ProdId>
      </ExpertiseAreasRow>
   </ExpertiseAreas>
</TechniciansRow>
```

Example 27–28 shows the interesting lines of code from a `TestClientReadXML` class that applies this XML datagram to a particular row in the `Technicians` view object, and to the nested set of the technician's areas of expertise via the view link accessor attribute to the `ExpertiseAreas` view object. `TestClientReadXML` class performs the following basic steps:

1. Finds a target row by key (e.g. for technician "ahunold").
2. Shows the XML produced for the row before changes are applied.
3. Obtains the parsed XML document with changes to apply using a helper method.
4. Reads the XML document to apply changes to the row.
5. Shows the XML with the pending changes applied.

   `TestClientReadXML` class is using the `XMLInterface.XML_OPT_ASSOC_CONSISTENT` flag described in Section 27.7.1, "How to Produce XML for Queried Data" to ensure that new, unposted rows are included in the XML.

Example 27–28 Applying Changes to and Existing Row with readXML()

```
ViewObject vo = am.findViewObject("Technicians");
Key k = new Key(new Object[] { 303 });
// 1. Find a target row by key (e.g. for technician "ahunold")
Row ahunold = vo.findByKey(k, 1)[0];
// 2. Show the XML produced for the row before changes are applied
printXML(ahunold.writeXML(-1, XMLInterface.XML_OPT_ALL_ROWS));
// 3. Obtain parsed XML document with changes to apply using helper method
Element xmlToRead = getInsertUpdateDeleteXMLGram();
printXML(xmlToRead);
// 4. Read the XML document to apply changes to the row
ahunold.readXML(getInsertUpdateDeleteXMLGram(), -1);
// 5. Show the XML with the pending changes applied
printXML(ahunold.writeXML(-1, XMLInterface.XML_OPT_ALL_ROWS |
                              XMLInterface.XML_OPT_ASSOC_CONSISTENT));
```

Running the code in Example 27–28 initially displays the "before" version of Alexander Hunold's information. Notice that:

- The `City` attribute has the value "Southlake"
- The expertise area for product `100` has a level of "Qualified"
- There is an expertise row for product `112`, and

- There is no expertise area row related to product `110`.

```
<TechniciansRow>
   <UserId>303</UserId>
   <UserRole>technician</UserRole>
   <Email>ahunold</Email>
    :
   <City>Southlake</City>
    :
   <ExpertiseAreas>
      <ExpertiseAreasRow>
         <ProdId>100</ProdId>
         <UserId>303</UserId>
         <ExpertiseLevel>Qualified</ExpertiseLevel>
      </ExpertiseAreasRow>
       :
      <ExpertiseAreasRow>
         <ProdId>112</ProdId>
         <UserId>303</UserId>
         <ExpertiseLevel>Expert</ExpertiseLevel>
      </ExpertiseAreasRow>
       :
   </ExpertiseAreas>
</TechniciansRow>
```

After applying the changes from the XML document using `readXML()` to the row and printing its XML again using writeXML() you see that:

- The `City` is now "Padova"
- The expertise area for product `100` has a level of "Expert"
- The expertise row for product `112` is removed, and
- A new expertise area row for product `110` got created.

```
<TechniciansRow>
   <UserId>303</UserId>
   <UserRole>technician</UserRole>
   <Email>ahunold</Email>
    :
   <City>Padova</City>
    :
   <ExpertiseAreas>
      <ExpertiseAreasRow>
         <ProdId>110</ProdId>
         <UserId>303</UserId>
         <ExpertiseLevel>Expert</ExpertiseLevel>
      </ExpertiseAreasRow>
      <ExpertiseAreasRow>
         <ProdId>100</ProdId>
         <UserId>303</UserId>
         <ExpertiseLevel>Expert</ExpertiseLevel>
      </ExpertiseAreasRow>
</TechniciansRow>
```

Note: The example illustrated using `readXML()` to apply changes to a single row. If the XML document contained a wrapping <Technicians> row, including the primary key attribute in each of its one or more nested <TechniciansRow> elements, then that document could be processed using the readXML() method on the `Technicians` view object for handling operations for multiple `Technician` rows.

27.8 Using Programmatic View Objects for Alternative Data Sources

By default view objects read their data from the database and automate the task of working with the Java Database Connectivity (JDBC) layer to process the database result sets. However, by overriding appropriate methods in its custom Java class, you can create a view object that programmatically retrieves data from alterative data sources like a `REF CURSOR`, an in-memory array, or a Java `*.properties` file, to name a few.

27.8.1 How to Create a Read-Only Programmatic View Object

To create a read-only programmatic view object, use the Create View Object wizard and follow these steps:

1. In step 1 on the **Name** panel, provide a name and package for the view object. In the **What kind of data do you need this view object to manage?** radio group, select **Rows Populated Programmatically, not Based on a Query**
2. In step 2 on the **Attributes** panel, click **New** one or more times to define the view object attributes your programmatic view object requires.
3. In step 3 on the **Attribute Settings** panel, adjust any setting you may need to for the attributes you defined.
4. In step 4 on the **Java** panel, enable a custom view object class to contain your code.
5. Click **Finish** to create the view object.

In your view object's custom Java class, override the methods described in Section 27.8.3, "Key Framework Methods to Override for Programmatic View Objects" to implement your custom data retrieval strategy.

27.8.2 How to Create an Entity-Based Programmatic View Object

To create a entity-based view object with programmatic data retrieval, create the view object in the normal way, enable a custom Java class for it, and override the methods described in the next section to implement your custom data retrieval strategy.

27.8.3 Key Framework Methods to Override for Programmatic View Objects

A programmatic view object typically overrides all of the following methods of the base `ViewObjectImpl` class to implement its custom strategy for retrieving data:

- `create()`

  This method is called when the view object instance is created and can be used to initialize any state required by the programmatic view object. At a minimum, this overridden method will contain the following lines to ensure the programmatic view object has no trace of a SQL query related to it:

  ```
  // Wipe out all traces of a query for this VO
  getViewDef().setQuery(null);
  getViewDef().setSelectClause(null);
  setQuery(null);
  ```

- `executeQueryForCollection()`

  This method is called whenever the view object's query needs to be executed (or re-executed).

- `hasNextForCollection()`

  This method is called to support the `hasNext()` method on the row set iterator for a row set created from this view object. Your implementation returns `true` if you have not yet exhausted the rows to retrieve from your programmatic data source.

- `createRowFromResultSet()`

  This method is called to populate each row of "fetched" data. Your implementation will call `createNewRowForCollection()` to create a new blank row and then `populateAttributeForRow()` to populate each attribute of data for the row.

- `getQueryHitCount()`

  This method is called to support the `getEstimatedRowCount()` method. Your implementation returns a count, or estimated count, of the number of rows that will be retrieved by the programmatic view object's query.

- `protected void releaseUserDataForCollection()`

  Your code can store and retrieve a user data context object with each row set. This method is called to allow you to release any resources that may be associated with a row set that is being closed.

Since the view object component can be related to several active row sets at runtime, many of the above framework methods receive an `Object` parameter named `qc` in which the framework will pass the collection of rows in question that your code is supposed to be filling, as well as the array of bind variable values that might affect which rows get populated into the specific collection.

You can store a user-data object with each collection of rows so your custom datasource implementation can associate any needed datasource context information. The framework provides the `setUserDataForCollection()` and `getUserDataForCollection()` methods to get and set this per-collection context information. Each time one of the overridden framework methods is called, you can use the `getUserDataForCollection()` method to retrieve the correct `ResultSet` object associated with the collection of rows the framework wants you to populate.

The examples in the following sections each override these methods to implement different kinds of programmatic view objects.

27.8.4 How to Create a View Object on a REF CURSOR

Sometimes your application might need to work with the results of a query that is encapsulated within a stored procedure. PL/SQL allows you to open a cursor to iterate through the results of a query, and then return a reference to this cursor to the client. This so-called `REF CURSOR` is a handle with which the client can then iterate the results of the query. This is possible even though the client never actually issued the original SQL `SELECT` statement.

Note: The examples in this section refer to the `ViewObjectOnRefCursor` project in the `AdvancedViewObjectExamples` workspace. See the note at the beginning of this chapter for download instructions. Run the `CreateRefCursorPackage.sql` script in the **Resources** folder against the `SRDemo` connection to set up the additional database objects required for the project.

Declaring a PL/SQL package with a function that returns a `REF CURSOR` is straightforward. For example, your package might look like this:

```
CREATE OR REPLACE PACKAGE RefCursorExample IS
  TYPE ref_cursor IS REF CURSOR;
  FUNCTION get_requests_for_tech(p_email VARCHAR2) RETURN ref_cursor;
  FUNCTION count_requests_for_tech(p_email VARCHAR2) RETURN NUMBER;
END RefCursorExample;
```

After defining an entity-based `RequestForTech` view object with an entity usage for a `ServiceRequest` entity object, go to its custom Java class `RequestForTechImpl.java`. At the top of the view object class, define some constant Strings to hold the anonymous blocks of PL/SQL that you'll execute using JDBC `CallableStatement` objects to invoke the stored functions:

```
/*
 * Execute this block to retrieve the REF CURSOR
 */
 private static final String SQL =
           "begin ? := RefCursorSample.getEmployeesForDept(?);end;";
/*
 * Execute this block to retrieve the count of service requests that
 * would be returned if you executed the statement above.
 */
private static final String COUNTSQL =
           "begin ? := RefCursorSample.countEmployeesForDept(?);end;";
```

Then, override the methods of the view object as described in the following sections.

27.8.4.1 The Overridden create() Method

The `create()` method removes all traces of a SQL query for this view object.

```
protected void create() {
  getViewDef().setQuery(null);
  getViewDef().setSelectClause(null);
  setQuery(null);
}
```

27.8.4.2 The Overridden executeQueryForCollection() Method

The `executeQueryForCollection()` method calls a helper method `retrieveRefCursor()` to execute the stored function and return the `REF CURSOR` return value, cast as a JDBC `ResultSet`. Then, it calls the helper method `storeNewResultSet()` that uses the `setUserDataForCollection()` method to store this `ResultSet` with the collection of rows for which the framework is asking to execute the query.

```
protected void executeQueryForCollection(Object qc,Object[] params,
                                         int numUserParams) {
  storeNewResultSet(qc,retrieveRefCursor(qc,params));
  super.executeQueryForCollection(qc, params, numUserParams);
}
```

The `retrieveRefCursor()` uses the helper method described in Section 25.5, "Invoking Stored Procedures and Functions" to invoke the stored function and return the `REF CURSOR`:

```
private ResultSet retrieveRefCursor(Object qc, Object[] params) {
  ResultSet rs = (ResultSet)callStoredFunction(OracleTypes.CURSOR,
                   "RefCursorExample.get_requests_for_tech(?)",
                   new Object[]{getNamedBindParamValue("Email",params)});
  return rs ;
}
```

27.8.4.3 The Overridden createRowFromResultSet() Method

For each row that the framework needs fetched from the datasource, it will invoke your overridden `createRowFromResultSet()` method. The implementation retrieves the collection-specific `ResultSet` object from the user-data context, uses the `createNewRowForCollection()` method to create a new blank row in the collection, and then use the `populateAttributeForRow()` method to populate the attribute values for each attribute defined at design time in the View Object Editor.

```
protected ViewRowImpl createRowFromResultSet(Object qc, ResultSet rs) {
  /*
   * We ignore the JDBC ResultSet passed by the framework (null anyway) and
   * use the resultset that we've stored in the query-collection-private
   * user data storage
   */
  rs = getResultSet(qc);

  /*
   * Create a new row to populate
   */
  ViewRowImpl r = createNewRowForCollection(qc);
  try {
    /*
     * Populate new row by attribute slot number for current row in Result Set
     */
    populateAttributeForRow(r,0, rs.getLong(1));
    populateAttributeForRow(r,1, rs.getString(2));
    populateAttributeForRow(r,2, rs.getString(3));
  }
  catch (SQLException s) {
    throw new JboException(s);
  }
  return r;
}
```

27.8.4.4 The Overridden hasNextForCollectionMethod()

The overridden implementation of the framework method `hasNextForCollection()` has the responsibility to return `true` or `false` based on whether there are more rows to fetch. When you've hit the end, you call the `setFetchCompleteForCollection()` to tell view object that this collection is done being populated.

```
protected boolean hasNextForCollection(Object qc) {
  ResultSet rs = getResultSet(qc);
  boolean nextOne = false;
  try {
    nextOne = rs.next();
    /*
     * When were at the end of the result set, mark the query collection
     * as "FetchComplete".
     */
    if (!nextOne) {
      setFetchCompleteForCollection(qc, true);
      /*
       * Close the result set, we're done with it
       */
      rs.close();
    }
  }
  catch (SQLException s) {
   throw new JboException(s);
  }
  return nextOne;
}
```

27.8.4.5 The Overridden releaseUserDataForCollection() Method

Once the collection is done with its fetch-processing, the overridden `releaseUserDataForCollection()` method gets invoked and closes the `ResultSet` cleanly so no database cursors are left open.

```
  protected void releaseUserDataForCollection(Object qc, Object rs) {
     ResultSet userDataRS = getResultSet(qc);
     if (userDataRS != null) {
      try {
        userDataRS.close();
      }
      catch (SQLException s) {
        /* Ignore */
      }
    }
    super.releaseUserDataForCollection(qc, rs);
  }
```

27.8.4.6 The Overridden getQueryHitCount() Method

Lastly, in order to properly support the view object's `getEstimatedRowCount()` method, the overridden `getQueryHitCount()` method returns a count of the rows that would be retrieved if all rows were fetched from the row set. Here the code uses a `CallableStatement` to get the job done. Since the query is completely encapsulated behind the stored function API, the code also relies on the PL/SQL package to provide an implementation of the count logic as well to support this functionality.

```
public long getQueryHitCount(ViewRowSetImpl viewRowSet) {
  Object[] params = viewRowSet.getParameters(true);
```

```
  BigDecimal id = (BigDecimal)params[0];
  CallableStatement st = null;
  try {
    st = getDBTransaction().createCallableStatement(COUNTSQL,
                            DBTransaction.DEFAULT);
    /*
     * Register the first bind parameter as our return value of type CURSOR
     */
    st.registerOutParameter(1,Types.NUMERIC);
    /*
     * Set the value of the 2nd bind variable to pass id as argument
     */
    if (id == null) st.setNull(2,Types.NUMERIC);
    else            st.setBigDecimal(2,id);
    st.execute();
    return st.getLong(1);
  }
  catch (SQLException s) {
    throw new JboException(s);
  }
  finally {try {st.close();} catch (SQLException s) {}}
}
```

27.8.5 Populating a View Object from Static Data

The SRDemo application's `SRStaticDataViewObjectImpl` class in the `FrameworkExtensions` project provides a programmatic view object implementation you can extend to populate code and description "lookup" data from static data in an in-memory array.

As shown in Example 27–29, it performs the following tasks in its overridden implementation of the key programmatic view object methods:

- `create()`

  When the view object is created, the data is loaded from the in-memory array. It calls a helper method to set up the `codesAndDescriptions` array of codes and descriptions and wipes out all traces of a query for this view object.

- `executeQueryForCollection()`

  Since the data is static, you don't really need to perform any query, but you still need to call the `super` to allow other framework setup for the row set to be done correctly. Since the code nulls out of traces of a query in the `create()` method, the view object won't actually perform any query during the call to `super`.

- `hasNextForCollection()`

  The code returns `true` if the `fetchPosition` is still less than the number of rows in the in-memory array

- `createRowFromResultSet()`

  Populates the "fetched" data for one row when the base view object implementation asks it to. It gets the data from the `codesAndDescriptions` array to populate the first and second attributes in the view object row (by zero-based index position).

- `getQueryHitCount()`

  The code returns the number of "rows" in the `codesAndDescriptions` array that was previously stored in the `rows` member field.

In addition, the following other methods help get the data setup:

- `setFetchPos()`

  Sets the current fetch position for the query collection. Since one view object can be used to create multiple row sets, you need to keep track of the current fetch position of each rowset in its "user data" context. It calls the `setFetchCompleteForCollection()` to signal to the view object that it's done fetching rows.

- `getFetchPos()`

  Get the current fetch position for the query collection. This returns the fetch position for a given row set that was stored in its user data context.

- `initializeStaticData()`

  Subclasses override this method to initialize the static data for display.

- `setCodesAndDescriptions()`

  Sets the static code and description data for this view object.

Example 27–29 Custom View Object Class to Populate Data from a Static Array

```
package oracle.srdemo.model.frameworkExt;
// Imports omitted
public class SRStaticDataViewObjectImpl extends SRViewObjectImpl {
  private static final int CODE = 0;
  private static final int DESCRIPTION = 1;
  int rows = -1;
  private String[][] codesAndDescriptions = null;

  protected void executeQueryForCollection(Object rowset, Object[] params,
                                           int noUserParams) {
    // Initialize our fetch position for the query collection
    setFetchPos(rowset, 0);
    super.executeQueryForCollection(rowset, params, noUserParams);
  }
  // Help the hasNext() method know if there are more rows to fetch or not
  protected boolean hasNextForCollection(Object rowset) {
    return getFetchPos(rowset) < rows;
  }
  // Create and populate the "next" row in the rowset when needed
  protected ViewRowImpl createRowFromResultSet(Object rowset,ResultSet rs) {
    ViewRowImpl r = createNewRowForCollection(rowset);
    int pos = getFetchPos(rowset);
    populateAttributeForRow(r, 0, codesAndDescriptions[pos][CODE]);
    populateAttributeForRow(r, 1, codesAndDescriptions[pos][DESCRIPTION]);
    setFetchPos(rowset, pos + 1);
    return r;
  }
  // When created, initialize static data and remove trace of any SQL query
  protected void create() {
    super.create();
    // Setup string arrays of codes and values from VO custom properties
    initializeStaticData();
    rows = (codesAndDescriptions != null) ? codesAndDescriptions.length : 0;
    // Wipe out all traces of a query for this VO
    getViewDef().setQuery(null);
    getViewDef().setSelectClause(null);
    setQuery(null);
  }
```

```
  // Return the estimatedRowCount of the collection
  public long getQueryHitCount(ViewRowSetImpl viewRowSet) {
    return rows;
  }
  // Subclasses override this to initialize their static data
  protected void initializeStaticData() {
    setCodesAndDescriptions(new String[][]{
      {"Code1","Description1"},
      {"Code2","Description2"}
    });
  }
  // Allow subclasses to initialize the codesAndDescriptions array
  protected void setCodesAndDescriptions(String[][] codesAndDescriptions) {
    this.codesAndDescriptions = codesAndDescriptions;
  }
  // Store the current fetch position in the user data context
  private void setFetchPos(Object rowset, int pos) {
    if (pos == rows) {
      setFetchCompleteForCollection(rowset, true);
    }
    setUserDataForCollection(rowset, new Integer(pos));
  }
  // Get the current fetch position from the user data context
  private int getFetchPos(Object rowset) {
    return ((Integer)getUserDataForCollection(rowset)).intValue();
  }
}
```

27.8.5.1 Basing Lookup View Object on SRStaticDataViewObjectImpl

The `ServiceRequestStatusList` view object in the SRDemo application defines two String attributes named `Code` and `Description`, and extends the `SRStaticDataViewObjectImpl` class. It overrides the `initializeStaticData()` method to supply the values of the legal service request status codes:

```
public class ServiceRequestStatusListImpl
       extends SRStaticDataViewObjectImpl {
  protected void initializeStaticData() {
    setCodesAndDescriptions(new String[][]{
        {"Open","Open"},
        {"Pending","Pending"},
        {"Closed","Closed"}
      });
  }
}
```

27.8.5.2 Creating a View Object Based on Static Data from a Properties File

Rather than compiling the static data for a view object into the Java class itself, it can be convenient to externalize it into a standard Java properties file with a *`Name=Value`* format like this:

```
#This is the property file format. Comments like this are ok
US=United States
IT=Italy
```

The `SRPropertiesFileViewObjectImpl` in the SRDemo application extends `SRStaticDataViewObjectImpl` to override the `initializeStaticData()` method and invoke the `loadDataFromPropertiesFile()` method shown in

Example 27–30 to read the static data from a properties file. This method does the following basic steps:

1. Derives the property file name based on the view definition name.

   For example, a `CountryList` view object in a *`x.y.z`*`.queries` package that extends this class would expect to read the properties file named `./`*`x`*`/`*`y`*`/`*`z`*`/queries/CountryList.properties` file.

2. Initializes a list to hold the name=value pairs.
3. Opens an input stream to read the properties file from the class path.
4. Loops over each line in the properties file.
5. If line contains and equals sign and is not a comment line that begins with a hash, then add a string array of `{`*`code`*`,`*`description`*`}` to the list.
6. Closes the line number reader and input stream.
7. Returns the list contains as a two-dimensional String array.

Example 27–30 Reading Static Data for a View Object from a Properties File

```
// In SRPropertiesFileViewObjectImpl.java
private synchronized String[][] loadDataFromPropertiesFile() {
  // 1. Derive the property file name based on the view definition name
  String propertyFile =
    getViewDef().getFullName().replace('.', '/') + ".properties";
  // 2. Initialize a list to hold the name=value pairs
  List codesAndDescriptionsList = new ArrayList(20);
  try {
    // 3. Open an input stream to read the properties file from the class path
    InputStream is = Thread.currentThread().getContextClassLoader()
                                       .getResourceAsStream(propertyFile);
    LineNumberReader lnr = new LineNumberReader(new InputStreamReader(is));
    String line = null;
    // 4. Loop over each line in the properties file
    while ((line = lnr.readLine()) != null) {
      line.trim();
      int eqPos = line.indexOf('=');
      if ((eqPos >= 1) && (line.charAt(0) != '#')) {
        // 5. If line contains "=" and isn't a comment, add String[]
        //    of {code,description} to the list
        codesAndDescriptionsList.add(new String[]{
                   line.substring(0, eqPos),
                   line.substring(eqPos + 1)});
      }
    }
    // 6. Close the line number reader and input stream
    lnr.close();
    is.close();
  } catch (IOException iox) {
    iox.printStackTrace();
    return new String[0][0];
  }
  // 7. Return the list contains as a two-dimensional String array
  return (String[][])codesAndDescriptionsList.toArray();
}
```

27.8.5.3 Creating Your Own View Object with Static Data

To create your own view object with static data that extends one of the example classes provided in the SRDemo application, define a new read-only programmatic view object with String attributes named `Code` and `Description`. On the **Java** panel of the View Object Editor, click **Class Extends** to specify the fully-qualified name of the `SRStaticDataViewObjectImpl` or `SRPropertiesFileViewObjectImpl` class as the custom class in the **Object** field. Then, enable a custom Java class for your view object and do the following:

If you extend `SRStaticDataViewObjectImpl`**...**
Then override the `initializeStaticData()` method and invoke the `loadDataFromPropertiesFile()` method shown

If you extend `SRStaticDataViewObjectImpl` **...**
Then create the appropriate *.properties file in the same directory as the view object's XML component definition, with a name that matches the name of the view object (`*ViewObjectName*.properties`).

27.9 Creating a View Object with Multiple Updatable Entities

> **Note:** To experiment with the example described in this section, use the same `ControllingPostingOrder` project in the `AdvancedEntityExamples` workspace used in Section 26.7, "Controlling Entity Posting Order to Avoid Constraint Violations".

When you create a view object with multiple entity usages, you can enable a secondary entity usage to be updatable by selecting it in the **Selected** list of the **Entity Objects** panel of the View Object Editor and:

- Selecting the **Reference** checkbox
- Deselecting the **Updatable** checkbox

If you only plan to use the view object to update or delete existing data, then this is the only step required. The user can update attributes related to any of the non-reference, updatable entity usages and the view row will delegate the changes to the appropriate underlying entity rows.

However, if you need a view object with multiple updatable entities to support creating new rows, then you need to write a bit of code to enable that to work correctly. When you call `createRow()` on a view object with multiple update entities, it creates new entity row parts for each updatable entity usage. Since the multiple entities in this scenario are related by an association, there are three pieces of code you might need to implement to ensure the new, associated entity rows can be saved without errors:

1. You may need to override the `postChanges()` method on entity objects involved to control the correct posting order.
2. If the primary key of the associated entity is populated by a database sequence using `DBSequence`, and if the multiple entity objects are associated but not composed, then you need to override the `postChanges()` and `refreshFKInNewContainees()` method to handle cascading the refreshed primary key value to the associated rows that were referencing the temporary value.

3. You need to override the `create()` method of the view object's custom view row class to modify the default row creation behavior to pass the context of the parent entity object to the newly-created child entity.

In Section 26.7, "Controlling Entity Posting Order to Avoid Constraint Violations", you've already seen the code required for 1 and 2 above in an example with associated `Product` and `ServiceRequest` entity objects. The only thing remaining is the overridden `create()` method on the view row. Consider a `ServiceRequestAndProduct` view object with a primary entity usage of `ServiceRequest` and secondary entity usages of `Product` and `User`. Assume the `Product` entity usage is marked as updatable and non-reference, while the `User` entity usage is a reference entity usage.

Example 27–31 shows the commented code required to correctly sequence the creation of the multiple, updatable entity row parts during a view row create operation.

Example 27–31 Overriding View Row create() Method for Multiple Updatable Entities

```
/**
 * By default, the framework will automatically create the new
 * underlying entity object instances that are related to this
 * view object row being created.
 *
 * We override this default view object row creation to explicitly
 * pre-populate the new (detail) ServiceRequestImpl instance using
 * the new (master) ProductImpl instance. Since all entity objects
 * implement the AttributeList interface, we can directly pass the
 * new ProductImpl instance to the ServiceRequestImpl create()
 * method that accepts an AttributeList.
 */
protected void create(AttributeList attributeList) {
  // The view row will already have created "blank" entity instances
  ProductImpl newProduct = getProduct();
  ServiceRequestImpl newServiceRequest = getServiceRequest();
   try {
      // Let product "blank" entity instance to do programmatic defaulting
      newProduct.create(attributeList);
      // Let service request "blank" entity instance to do programmatic
      // defaulting passing in new ProductImpl instance so its attributes
      // are available to the EmployeeImpl's create method.
      newServiceRequest.create(newProduct);
   }
   catch (JboException ex) {
     newProduct.revert();
     newServiceRequest.revert();
     throw ex;
   }
   catch (Exception otherEx) {
     newProduct.revert();
     newServiceRequest.revert();
     throw new RowCreateException(true      /* EO Row? */,
                                  "Product" /* EO Name */,
                                  otherEx   /* Details */);
   }
}
```

In order for this view row class to be able to invoke the protected `create()` method on the `Product` and `ServiceRequest` entity objects, they need to override the `create()` method. If the view object and entity objects are in the same package, the overridden `create()` method can have protected access. Otherwise, it requires `public` access.

```
/**
 * Overridding this method in this class allows friendly access
 * to the create() method by other classes in this same package, like the
 * ServiceRequestsAndProduct view object implementation class, whose overridden
 * create() method needs to call this.
 * @param nameValuePair
 */
  protected void create(AttributeList nameValuePair) {
    super.create(nameValuePair);
  }
```

27.10 Declaratively Preventing Insert, Update, and Delete

Some 4GL tools like Oracle Forms provide declarative properties that control whether a given data collection allows inserts, updates, or deletes. While the view object does not yet support this as a built-in feature in the current release, it's easy to add this facility using a framework extension class that exploits custom metadata properties as the developer-supplied flags to control insert, update, or delete on a view object.

> **Note:** The examples in this section refer to the `DeclarativeBlockOperations` project in the `AdvancedViewObjectExamples` workspace. See the note at the beginning of this chapter for download instructions.

To allow developers to have control over individual view object instances, you could adopt the convention of using application module custom properties by the same name as the view object instance. For example, if an application module has view object instances named `ProductsInsertOnly`, `ProductsUpdateOnly`, `ProductsNoDelete`, and `Products`, your generic code might look for application module custom properties by these same names. If the property value contains `Insert`, then insert is enabled for that view object instance. If the property contains `Update`, then update allowed. And, similarly, if the property value contains `Delete`, then delete is allowed. You could use helper methods like this to test for these application module properties and determine whether insert, update, and delete are allowed for a given view object:

```
private boolean isInsertAllowed() {
  return isStringInAppModulePropertyNamedAfterVOInstance("Insert");
}
private boolean isUpdateAllowed() {
  return isStringInAppModulePropertyNamedAfterVOInstance("Update");
}
private boolean isDeleteAllowed() {
  return isStringInAppModulePropertyNamedAfterVOInstance("Delete");
}
private boolean isStringInAppModulePropertyNamedAfterVOInstance(String s) {
  String voInstName = getViewObject().getName();
  String propVal = (String)getApplicationModule().getProperty(voInstName);
  return propVal != null ? propVal.indexOf(s) >= 0 : true;
}
```

Example 27–32 shows the other code required in a custom framework extension class for view rows to complete the implementation. It overrides the following methods:

- `isAttributeUpdateable()`

  To enable the user interface to disable fields in a new row if insert is not allowed or to disable fields in an existing row if update is not allowed.

- `setAttributeInternal()`

  To prevent setting attribute values in a new row if insert is not allowed or to prevent setting attributes in an existing row if update is not allowed.

- `remove()`

  To prevent remove if delete is not allowed.

- `create()`

  To prevent create if insert is not allowed.

Example 27–32 Preventing Insert, Update, or Delete Based on Custom Properties

```
public class CustomViewRowImpl extends ViewRowImpl {
  public boolean isAttributeUpdateable(int index) {
    if (hasEntities() &&
        ((isNewOrInitialized() && !isInsertAllowed()) ||
         (isModifiedOrUnmodified() && !isUpdateAllowed()))) {
      return false;
    }
    return super.isAttributeUpdateable(index);
  }
  protected void setAttributeInternal(int index, Object val) {
    if (hasEntities()) {
      if (isNewOrInitialized() && !isInsertAllowed())
        throw new JboException("No inserts allowed in this view");
      else if (isModifiedOrUnmodified() && !isUpdateAllowed())
        throw new JboException("No updates allowed in this view");
    }
    super.setAttributeInternal(index, val);
  }
  public void remove() {
    if (!hasEntities() || isDeleteAllowed() || isNewOrInitialized())
      super.remove();
    else
        throw new JboException("Delete not allowed in this view");
  }
  protected void create(AttributeList nvp) {
    if (isInsertAllowed()) {
      super.create(nvp);
    } else {
      throw new JboException("Insert not allowed in this view");
    }
  }
  // private helper methods omitted
}
```

28

Application Module State Management

This chapter describes the application module state management facility and how to use it.

This chapter includes the following sections:

- Section 28.1, "Understanding Why State Management is Necessary"
- Section 28.2, "The ADF Business Components State Management Facility"
- Section 28.3, "Controlling the State Management Release Level"
- Section 28.4, "What State Is Saved and When is It Cleaned Up?"
- Section 28.5, "Managing Custom User Specific Information"
- Section 28.6, "Managing State for Transient View Objects"
- Section 28.7, "Using State Management for Middle-Tier Savepoints"
- Section 28.8, "Testing to Ensure Your Application Module is Activation-Safe"
- Section 28.9, "Caveats Regarding Pending Database State"

28.1 Understanding Why State Management is Necessary

Most real-world business applications need to support multi-step user tasks. Modern sites tend to use a "step-by-step" style user interface to guide the end user through a logical sequence of pages to complete these tasks. When the task is done, the user can save or cancel everything as a unit.

28.1.1 Examples of Multi-Step Tasks

In a typical search-then-edit scenario, the end user searches to find an appropriate row to update, then may open several different pages of related master/detail information to make edits before deciding to save or cancel his work. Consider another scenario where the end user wants to book a vacation online. The process may involve the end user's entering details about:

- One or more flight segments that comprise the journey
- One or more passengers taking the trip
- Seat selections and meal preferences
- One or more hotel rooms in different cities
- Car they will rent

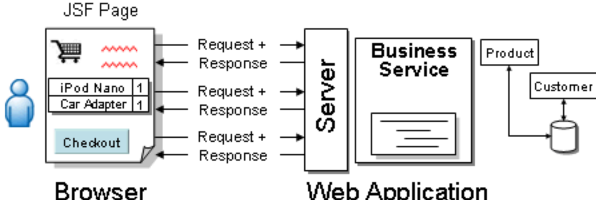
Along the way, the user might decide to complete the transaction, save the reservation for finishing later, or abandoning the whole thing.

It's clear these scenarios involve a logical unit of work that spans multiple web pages. You've seen in previous chapters how to use JDeveloper's JSF page navigation diagram to design the page flow for these use cases, but that is only part of the puzzle. The pending changes the end user makes to business domain objects along the way — `Trip`, `Flight`, `Passenger`, `Seat`, `HotelRoom`, `Auto`, etc. — represent the in-progress state of the application for each end user. Along with this, other types of "bookkeeping" information about selections made in previous steps comprise the complete picture of the application state.

28.1.2 Stateless HTTP Protocol Complicates Stateful Applications

While it may be easy to imagine these multi-step scenarios, *implementing* them in web applications is complicated by the stateless nature of HTTP, the hypertext transfer protocol. Figure 28–1 illustrates how an end user's "visit" to a site comprises a series of HTTP request/response pairs. However, HTTP affords a web server no way to distinguish one user's request from another user's, or to differentiate between a single user's first request and any subsequent requests he makes while interacting with the site. The server gets each request from any user always as if it were the first (and only!) one they make.

Figure 28–1 Web Applications Use the Stateless HTTP Protocol



But even if you've never *implemented* your own web applications before, since you've undoubtedly *used* a web application to buy a book, plan a holiday, or even just read your email, it's clear that a solution must exist to distinguish one user from another.

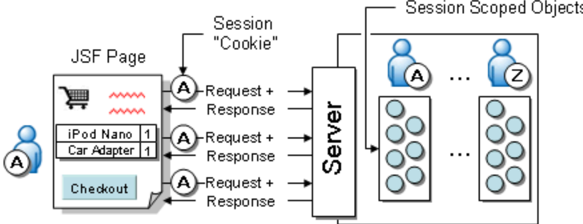
28.1.3 How Cookies Are Used to Track a User Session

As shown in Figure 28–2, the technique to recognize an ongoing sequence of requests from the same end user over the stateless HTTP protocol involves a unique identifier called a "cookie.." A cookie is a name/value pair that is sent in the header information of each HTTP request the user makes to a site. On the initial request made by a user, the cookie is not part of the request. The server uses the *absence* of the cookie to detect the start of a user's session of interactions with the site, and it returns a unique identifier to the browser that represents this session for this user. In practice, the cookie value is a long string of letters and numbers, but for simplicity's sake, assume that the unique identifier is a letter like "A" or "Z" that corresponds to different users using the site.

Web browsers support a standard way of recognizing the cookie returned by the server, along with a way of identifying what site sent the cookie and how long it should remember the cookie value. On each subsequent request made by that user, until the cookie "expires" the browser sends the cookie along in the header of the request.

The server uses the value of the cookie to distinguish the requests made by different users. A cookie that expires when you close your browser is known as a "session cookie," while other cookies set to live beyond a single browser session might expire in a week, a month, or a year from when they were first created.

Figure 28–2 Tracking State Using a Session Cookies and Server-Side Session



J2EE-compliant web servers provide a standard server-side facility called the `HttpSession` that allows a web developer to store Java objects related to a particular user's session as named attribute/value pairs. An object placed in this session `Map` on one request can be retrieved by the developer while handling a subsequent request during the same session. The session stays "active" while the user continues to send new requests within the timeframe configured by the <session-timeout> element in the `web.xml` file. The default session length is 35 minutes.

28.1.4 Performance and Reliability Impact of Using HttpSession

The `HttpSession` facility is an ingredient in most application state management strategies, but it can present performance and reliability problems if not used judiciously. First, since the session-scope Java objects you create are held in the memory of the J2EE web server, the objects in the HTTP session are lost if the server should fail.

As shown in Figure 28–3, one way to improve the reliability is to configure multiple J2EE servers in a cluster. By doing this, the J2EE application server replicates the objects in the HTTP session for each user across multiple servers in the cluster so that if one server goes down, the objects exist in the memory of the other servers in the cluster that can continue to handle the users requests. Since the cluster comprises separate servers, replicating the HTTP session contents among them involves broadcasting the changes made to HTTP session objects over the network.

Figure 28–3 Session Replication in a Server Cluster

You can begin to see some of the performance implications of overusing the HTTP session:

- The more active users, the more HTTP sessions will be created on the server.
- The more objects stored in each HTTP session, the more memory you will need.
- In a cluster, the more objects in each HTTP session that *change*, the more network traffic will be generated to replicate the changed objects to other servers in the cluster.

At the outset, it would seem that keeping the number of objects stored in the session to a minimum addresses the problem. However, this implies leveraging an alternative mechanism for temporary storage for each user's pending application state. The most popular alternatives involve saving the application state to the database between requests or to a file of some kind on a shared file system.

Of course, this is easier said than done. A possible approach involves eagerly saving the pending changes to your underlying database tables and committing the transaction at the end of each HTTP request. But this idea has two key drawbacks:

- *Your database constraints might fail.*

  At any given step of the multi-step process, the information may only be partially complete, and this could cause errors at the database level when trying to save the changes.

- *You complicate rolling back the changes*.

  Cancelling the logical of unit of work would involves carefully deleting all of the eagerly-committed rows in possible multiple tables.

These limitations have lead developers in the past to invent solutions involving a "shadow" set of database tables with no constraints and with all of the column types defined as character-based. Using such a solution becomes very complex very quickly. Ultimately, you will conclude that you need some kind of generic application state management facility to address these issues in a more generic and workable way. The solution comes in the form of ADF Business Components, which implements this for you out of the box.

28.2 The ADF Business Components State Management Facility

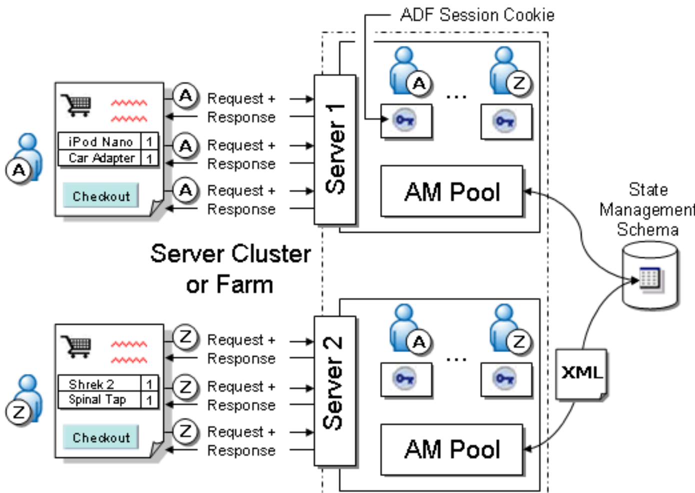
The application module component and application module pool cooperate to offer a generic solution to database-backed, application state management. This feature enables you to easily create web applications that support multi-step use cases without falling prey to the memory, reliability, or implementation complexity problems described in the previous section.

Your ADF Business Components-based application automatically manages the application state of each user session. This provides the simplicity of a stateful programming model that you are used to in previous 4GL tools, yet, implemented in a way that delivers scalability nearing that of a purely stateless application. Understanding what happens behind the scenes is essential to make the most efficient use of this important feature.

28.2.1 Basic Architecture of the State Management Facility

You can use application module components to implement completely stateless applications or to support a unit of work that spans multiple browser pages. Figure 28–4 illustrates the basic architecture of the state management facility to support these multi-step scenarios. An application module supports *passivating* its pending transaction state to an XML document, which is stored in the database in a single, generic table, keyed by a unique passivation snapshot ID. It also supports the reverse operation of *activating* pending transaction state from one of these saved XML "snapshots." This passivation and activation is performed automatically by the application module pool when needed.

Figure 28–4 ADF Provides Generic, Database-Backed State Management



The ADF binding context is the one object that lives in the `HttpSession` for each end user. It hold references to lightweight application module data control objects that manage acquiring an application module instance from the pool at the beginning of each request and releasing it to the pool at the end of each request. The data control holds a reference to the ADF "session cookie" that identifies the user session. In particular, business domain objects created or modified in the pending transaction are *not* saved in the `HttpSession` using this approach. This minimizes both the session memory required per user and eliminates the network traffic related to session replication if the servers are configured in a cluster.

For improved reliability, if you have multiple application servers and you enable the optional ADF Business Components failover support, then subsequent end-user requests can be handled by any server in your server farm or cluster. The ADF session cookie saved in the client browser is enough to "reactivate" the pending application state from the database-backed XML snapshot if required, regardless of what server handles the request.

28.2.2 Understanding When Passivation and Activation Occurs

To better understand when the automatic passivation and activation of application module state occurs, consider the following simple case:

1. At the beginning of an HTTP request, the application module data control handles the `beginRequest` event by checking out an application module instance from the pool.

   The application module pool returns an *unreferenced* instance. An unreferenced application module is one that is not currently managing the pending state for any other user session.

2. At the end of the request, the application module data control handles the `endRequest` event by checking the application module instance back into the pool in "managed state" mode.

   That application module instance is now *referenced* by the data control that just used it. And the application module instance is an object that still contains pending transaction state made by the data control (that is, entity object and view object caches; updates made but not committed; and cursor states), stored in memory. As you'll see below, it's not *dedicated* to this data control, just referenced by it.

3. On a subsequent request, the same data control — identified by its `SessionCookie` — checks out an application module instance again.

   Due to the "stateless with user affinity" algorithm the pool uses, you might assume that the pool returns the exact same application module instance, with the state still there in memory.

Sometimes due to a high number of users simultaneously accessing the site, application module instances must be sequentially reused by different user sessions. In this case, the application pool must *recycle* a currently referenced application module instance for use by another session, as follows:

1. The application module data control for User A's session checks an application module instance into the application pool at the end of a request. Assume this instance is named `AM1`.

2. The application module data control for User Z's new session requests an application module instance from the pool for the first time, but there are no unreferenced instances available. The application module pool then:

   - Passivates the state of instance `AM1` to the database.
   - Resets the state of `AM1` in preparation to be used by another session.
   - Returns the `AM1` instance to User Z's data control.

3. On a subsequent request, the application module data control for User A's session requests an application module instance from the pool. The application module pool then:

   - Obtains an unreference instance.

     This could be instance `AM1`, obtained by following the same steps as in (2) above, or another `AM2` instance if it had become unreferenced in the meantime.

   - Activates the appropriate pending state for User A from the database.
   - Returns the application module instance to User A's data control.

The process of passivation, activation, and recycling allows the state referenced by the data control to be preserved across requests without requiring a dedicated application module instance for each data control. Both browser users in the above scenario are carrying on an application transaction that spans multiple HTTP requests, but the end users are unaware whether the passivation and activation is occurring in the background. They just continue to see the pending changes. In the process, the pending changes never need to be saved into the underlying application database tables until the end user is ready to commit the logical unit of work.

The application module pool makes a best effort to keep an application module instance "sticky" to the current data control whose pending state it is managing. This is known as maintaining user session affinity. The best performance is achieved if a data control continues to use exactly the same application module instance on each request, since this avoids any overhead involved in reactivating the pending state from a persisted snapshot.

28.2.3 How Passivation Changes When Optional Failover Mode is Enabled

There is a parameter called `jbo.dofailover` that can be set in your application module configuration on the **Pooling and Scalability** tab of the Configuration Editor. This parameter controls when and how often passivation occurs. When the failover feature is disabled, which it is by default, then application module pending state will only be passivated on demand when it must be. This occurs just before the pool determines it must hand out a currently-referenced application module instance to a *different* data control.

In contrast, with the failover feature turned on, the application module’s pending state is passivated every time it is checked back into application module pool. This provides the most pessimistic protection against application server failure. The application module instances' state is always saved and may be activated by any application module instance at any time. Of course, this capability comes at expense of the additional overhead of eager passivation on each request.

Note: When running or debugging an application that uses failover support within the JDeveloper environment, you are frequently starting and stopping the embedded OC4J server. The ADF failover mechanism has no way of knowing whether you stopped the embedded server to simulate an application server failure, or whether you stopped it because you want to retest something from scratch in a "fresh" server instance. If you intend doing the latter, Oracle recommends exiting out of your browser before restarting the application on the embedded server. This eliminates the chance that you will be confused by the correct functioning of the failover mechanism when you didn't intend to be testing that aspect of your application.

28.3 Controlling the State Management Release Level

When a data control handles the `endRequest` notification indicating the processing for the current HTTP request has completed, it releases the application module instance by checking it back into the application module pool. The application module pool manages instances and performs state management tasks (or not) based on the *release level* you use when returning the instance to the pool.

ADF supports the release levels described in the following sections.

28.3.1 Supported Release Levels

Managed Level

This is the *default* level. This level implies that application module's state is relevant and has to be preserved for this data control to span over several HTTP requests. Managed level does not guarantee that for the next request this data control will receive the same physical application module instance, but it does guarantees that an application module with identical state will be provided so it is logically the same application module instance each time. It is important to note that the framework makes the best effort it can to provide the same instance of application module for the same data control if it is available at the moment. This is done for better performance since the same application module does not need to activate the previous state which it still has intact after servicing the same data control during previous request. However, the data control is not guaranteed to receive the same instance for all its requests and if the application module that serviced that data control during previous is busy or unavailable, then a different application module will activate this data control's state. For this reason, it is not valid to cache references to application module objects, view objects, or view rows across HTTP requests in controller-layer code.

This mode was called the "Stateful Release Mode" in previous releases of JDeveloper.

Note: If the `jbo.ampool.doampooling` configuration property is `false` — corresponding to your unchecking the **Enable Application Module Pooling** option in the **Pooling and Scalability** tab of the Configuration Editor — then there is effectively no pool. In this case, when the application module instance is released at the end of a request it is immediately removed. On subsequent requests made by the same user session, a new application module instance must be created to handle each user request, and pending state must be reactivated from the passivation store. Setting this property to `false` is useful to discover problems in your application logic that might occur when reactivation does occur due to unpredictable load on your system. However, you'll typically never run a production system with this option set to `false`.

Unmanaged Level

This mode implies that no state associated with this data control has to be preserved to survive beyond the current HTTP request. This level is the most efficient in performance because there is no overhead related to state management. However, you should limited in its use to applications that require no state management, or to cases when state no longer needs to be preserved at this point (a classic example is releasing the application module after servicing the HTTP request from logout page).

This mode was called the "Stateless Release Mode" in previous releases of JDeveloper.

Reserved Level

This level guarantees that each data control will be assigned its own application module during its first request and for all subsequent requests coming from the `HttpSession` associated with this data control. This data control will always receive the same physical instance of application module. This mode exists for legacy compatibility reasons and for very rare special use cases. In general, it is strongly recommended never to use this mode. You would normally avoid using this mode because the data control to application module correlation becomes one to one, the scalability of the application reduces very sharply, and so does reliability of the application.

Reliability suffers because if for whatever reason the application module is lost, the data control will not be able to receive any other application module in its place from the pool, and so `HttpSession` gets lost as well, which is not the case for managed level.

> **Note:** The failover option is ignored for an application module released with Reserved release level since its use implies your application absolutely requires working with the same application module instance on each request.

28.3.2 Setting the Release Level at Runtime

If you do not want to use the default "Managed State" release level, you can set your desired level programmatically. Use the API's describe in the following section:

Unmanaged Level

To set a data control to release its application module using the unmanaged level, call the `resetState()` method on the `DCDataControl` class (in the `oracle.adf.model.binding` package).

You can call this method any time during the request. This will cause application module not to passivate its state at all when it is released to the pool at the end of the request. Note that this method only affects the current application module instance in the current request. After this, the application module is released in unmanaged level to the pool, it becomes unreferenced and gets reset. The next time the application module is used by a client, it will be used in the managed level again by default.

> **Note:** You can programmatically release the application module with the unmanaged level when you want to signal that the user has ended a logical unit of work. This will happen automatically when the `HTTPSession` times out, as described below.

Reserved Level

To set a data control to release its application module using the reserved level, call the `setReleaseLevel()` method of the `DCJboDataControl` class (in the `oracle.adf.model.bc4j` package), and pass the integer constant `ApplicationModule.RELEASE_LEVEL_RESERVED`.

When the release level for an application module has been changed to "Reserved" it will stay so for all subsequent requests until explicitly changed.

Managed Level

If you have set an application module to use reserved level, you can later set it back to use managed level by calling the `setReleaseLevel()` method of the `DCJboDataControl` class, and passing the integer constant `ApplicationModule.RELEASE_LEVEL_MANAGED`.

The sections below illustrate four different contexts you might make use of these release mode API's.

28.3.2.1 Setting Release Level in a JSF Backing Bean

Example 28–1 shows calling the `resetState()` method on a data control named `UserModuleDataControl` from the action method of a JSF backing bean.

Example 28–1 Calling resetState() on Data Control in a JSF Backing Bean Action Method

```
package devguide.advanced.releasestateless.controller.backing;
import devguide.advanced.releasestateless.controller.JSFUtils;
import oracle.adf.model.BindingContext;
import oracle.adf.model.binding.DCDataControl;
/**
 * JSF Backing bean for the "Example.jspx" page
 */
public class Example {
  /**
   * In an action method, call resetState() on the data control to cause
   * it to release to the pool with the "unmanaged" release level.
   * In other words, as a stateless application module.
   */
  public String commandButton_action() {
    // Add event code here...
    getDataControl("UserModuleDataControl").resetState();
    return null;
  }
  private DCDataControl getDataControl(String name) {
    BindingContext bc =
      (BindingContext)JSFUtils.resolveExpression("#{data}");
    return bc.findDataControl(name);
  }
}
```

28.3.2.2 Setting Release Level in an ADF PagePhaseListener

Example 28–2 shows calling the `resetState()` method on a data control named `UserModuleDataControl` from the after-prepareRender phase of the ADF lifecycle using a custom ADF page phase-listener class. You would associate this custom class to a particular page by setting the `ControllerClass` attribute on the page's page definition to the fully-qualified name of this class.

Example 28–2 Calling resetState() on Data Control in a Custom PagePhaseListener

```
package devguide.advanced.releasestateless.controller;
import oracle.adf.controller.v2.lifecycle.Lifecycle;
import oracle.adf.controller.v2.lifecycle.PagePhaseEvent;
import oracle.adf.controller.v2.lifecycle.PagePhaseListener;
import oracle.adf.model.binding.DCDataControl;
public class ReleaseStatelessPagePhaseListener
       implements PagePhaseListener {
  /**
   * In the "after" phase of the final "prepareRender" ADF Lifecycle
   * phase, call resetState() on the data control to cause it to release
   * to the pool with the "unmanaged" release level. In other words,
   * as a stateless application module.
   *
   * @param event ADF page phase event
   */
  public void afterPhase(PagePhaseEvent event) {
    if (event.getPhaseId() == Lifecycle.PREPARE_RENDER_ID) {
      getDataControl("UserModuleDataControl", event).resetState();
    }
  }
```

```
  // Required to implement the PagePhaseListener interface
  public void beforePhase(PagePhaseEvent event) {}
  private DCDataControl getDataControl(String name,
                                       PagePhaseEvent event) {
    return event.getLifecycleContext()
                .getBindingContext()
                .findDataControl(name);
  }
}
```

28.3.2.3 Setting Release Level in an ADF PageController

Example 28–3 shows calling the `resetState()` method on a data control named `UserModuleDataControl` from an overridden prepareRender() method of a custom ADF page controller class. You would associate this custom class to a particular page by setting the `ControllerClass` attribute on the page's page definition to the fully-qualified name of this class.

> **Note:** You can accomplish basically the same kinds of page-specific lifecycle customization tasks using a custom `PagePhaseListener` or a custom `PageController` class. The key difference is that the `PagePhaseListener` interface can be implemented on any class, while a custom `PageController` must extend the PageController class in the `oracle.adf.controller.v2.lifecycle` package.

Example 28–3 Calling resetState() on Data Control in a Custom ADF PageController

```
package devguide.advanced.releasestateless.controller;
import oracle.adf.controller.v2.context.LifecycleContext;
import oracle.adf.controller.v2.lifecycle.PageController;
import oracle.adf.controller.v2.lifecycle.PagePhaseEvent;
import oracle.adf.model.binding.DCDataControl;
public class ReleaseStatelessPageController extends PageController {
  /**
   * After calling the super in the final prepareRender() phase
   * of the ADF Lifecycle, call resetState() on the data control
   * to cause it to release to the pool with the "unmanaged"
   * release level. In other words, as a stateless application module.
   *
   * @param lcCtx ADF lifecycle context
   */
  public void prepareRender(LifecycleContext lcCtx) {
    super.prepareRender(lcCtx);
    getDataControl("UserModuleDataControl", lcCtx).resetState();
  }
  private DCDataControl getDataControl(String name,
                                       LifecycleContext lcCtx) {
    return lcCtx.getBindingContext().findDataControl(name);
  }
}
```

28.3.2.4 Setting Release Level in an Custom ADF PageLifecycle

If you wanted to build an ADF application where every request was handled in a completely stateless way, use a global custom `PageLifecycle` class as shown in Example 28–4. See Section 10.5.4.1, "Globally Customizing the ADF Page Lifecycle" for details on how to configure your application to use your custom lifecycle in a global way.

Example 28–4 Calling resetState() on Data Control in a Custom ADF PageLifecycle

```
package devguide.advanced.releasestateless.controller;
import oracle.adf.controller.faces.lifecycle.FacesPageLifecycle;
import oracle.adf.controller.v2.context.LifecycleContext;
import oracle.adf.model.binding.DCDataControl;
public class ReleaseStatelessPageLifecycle extends FacesPageLifecycle {
  /**
   * After calling the super in the final prepareRender() phase
   * of the ADF Lifecycle, call resetState() on the data control
   * to cause it to release to the pool with the "unmanaged"
   * release level. In other words, as a stateless application module.
   *
   * @param lcCtx ADF lifecycle context
   */
  public void prepareRender(LifecycleContext lcCtx) {
    super.prepareRender(lcCtx);
    getDataControl("UserModuleDataControl", lcCtx).resetState();
  }
  private DCDataControl getDataControl(String name,
                                       LifecycleContext lcCtx) {
    return lcCtx.getBindingContext().findDataControl(name);
  }
}
```

28.4 What State Is Saved and When is It Cleaned Up?

The information saved by passivation is divided in two parts: transactional and non-transactional state. Transactional state is the set of updates made to entity object data – performed either directly on entity objects or on entities through view object rows – that are intended to be saved into the database. Non-transactional state comprises view object runtime settings, such as the current row index, WHERE clause, and ORDERBY clause.

28.4.1 What State is Saved?

The information saved as part of the application module passivation "snapshot" includes the following.

Transactional State

- New, modified, and deleted entities in the entity caches of the root application module for this user session's (including old/new values for modified ones).

Non-Transactional State

- For each active view object (both statically and dynamically created):
  - Current row indicator for each row set (typically one)
  - New rows and their positions (New rows are treated differently then updated ones. Their index in the VO is traced as well)
  - ViewCriteria and all its related parameters such as view criteria row etc.
  - Flag indicating whether or not a row set has been executed
  - Range start and Range size
  - Access mode
  - Fetch mode and fetch size
  - Any view object-level custom data

- SELECT, FROM, WHERE, and ORDER BY clause if created dynamically or changed from the View definition

---

Note: If you enable ADF Business Components runtime diagnostics, the contents of each XML state snapshot. See Section 5.5.3.2, "Enabling ADF Business Components Debug Diagnostics" for more information on how to enable diagnostics.

28.4.2 Where is the State Saved?

By default, passivation snapshots are saved in the database, but you can configure it to use the file system as an alternative.

28.4.2.1 How Database-Backed Passivation Works

The passivated XML snapshot is written to a `BLOB` column in a table named `PS_TXN`, using a connection specified by the `jbo.server.internal_connection` property. Each time a passivation record is saved, it is assigned a unique passivation snapshot ID based on the sequence number taken from the `PS_TXN_SEQ` sequence. The ADF session cookie held by the application module data control in the ADF binding context remembers the latest passivation snapshot ID that was created on its behalf and remembers the previous ID that was used.

28.4.2.2 Controlling the Schema Where the State Management Table Resides

The ADF runtime recognizes a configuration property named `jbo.server.internal_connection` that controls which database connection/schema should be used for the creation of the `PS_TXN` table and the `PS_TXN_SEQ` sequence. If you don't set the value of this configuration parameter explicitly, then the state management facility creates the temporary tables using the credentials of the current application database connection.

To keep the temporary information separate, the state management facility will use a different connection *instance* from the database connection pool, but the database credentials will be the same as the current user. Since the framework creates temporary tables, and possibly a sequence if they don't already exists, the implication of not setting a value for the `jbo.server.internal_connection` is that the current database user must have `CREATE TABLE`, `CREATE INDEX` and `CREATE SEQUENCE` privileges. Since this is often not desirable, Oracle recommends always supplying an appropriate value for the `jbo.server.internal_connection` property, providing the credentials for a "state management" schema where table and schema be created. Valid values for the `jbo.server.internal_connection` property in your configuration are:

- A fully-qualified JDBC connection URL like:

  `jdbc:oracle:thin:`*`someuser/somepassword@host`*`:`*`port`*`:`*`SID`*

- A JDBC datasource name like:

  `java:/comp/env/jdbc/YourJ2EEDataSourceName`

28.4.2.3 Configuring the Type of Passivation Store

Passivated information can be stored in several places. You can control it programmatically or by configuring an option in the application module configuration. The choices are database or a file stored on local file system:

- **File**

  This choice may be the fastest available as access to the file is faster then access to the database. This choice is good if the entire middle tier (one or multiple Oracle Application Server installation(s) and all their OC4J instances) is either installed on the same machine or have access to a commonly shared file system, so passivated information is accessible to all. Usually, this choice may be good for a small middle tier where one Oracle Application Server is used. In other words this is very suitable choice for small middle tier such as one Oracle Application Server with all its components installed on one physical machine. The location and name of the persistent snapshot files are determined by `jbo.tmpdir` property if specified. It follows usual rules of ADF property precedence for a configuration property. If nothing else is specified, then the location is determined by `user.dir` if specified. This is a default property and the property is OS specific.

- **Database**

  This is the *default* choice. While it may be a little slower than passivating to file, it is by far the most reliable choice. With passivation to file, the common problem might be that it is not accessible to Oracle Application Server instances that are remotely installed. In this case, in a cluster environment, if one node goes down the other may not be able to access passivated information and then failover will not work. Another possible problem is that even if file is accessible to the remote node, the access time for the local and remote node may be very different and performance will be inconsistent. With database access, time should be about the same for all nodes.

To set the value of your choice in design time, set the property `jbo.passivationstore` to `database` or `file`. The value `null` will indicate that a connection-type-specific default should be used. This will use database passivation for Oracle or DB2, and file serialization for any others.

To set the storage programmatically use the method `setStoreForPassiveState()` of interface `oracle.jbo.ApplicationModule`. The parameter values that you can pass are:

- `PASSIVATE_TO_DATABASE`
- `PASSIVATE_TO_FILE`

28.4.3 When is the State Cleaned Up?

Under normal circumstances, the ADF state management facility provides automatic cleanup of the passivation snapshot records.

28.4.3.1 Previous Snapshot Removed When Next One Taken

When a passivation record is saved to the database on behalf of a session cookie, as described above, this passivation record gets a new, unique snapshot ID. The passivation record with the previous snapshot ID used by that same session cookie is deleted as part of the same transaction. In this way, assuming no server failures, there will only ever be a single passivation snapshot record per active end-user session.

28.4.3.2 Passivation Snapshot Removed on Unmanaged Release

The passivation snapshot record related to a session cookie is removed when the application module is checked into the pool with the unmanaged state level. This can occur when:

- Your code specifically calls `resetState()` on the application module data control.
- Your code explicitly invalidates the `HttpSession`, for example, as part of implementing an explicit "Logout" functionality.
- The `HttpSession` times out due to exceeding the session timeout threshold for idle time and failover mode is disabled (which is the default).

In each of these cases, the application module pool also resets the application module referenced by the session cookie to be "unreferenced" again. Since no changes were ever saved into the underlying database tables, once the pending session state snapshots are removed, there remains no trace of the unfinished work the user session had completed up to that point.

28.4.3.3 Passivation Snapshot Retained in Failover Mode

When the failover mode is enabled, if the `HttpSession` times out due to session inactivity, then the passivation snapshot is retained so that the end user can continue their work when they return to their browser.

After a break in the action, when the end user returns to his browser and continues to use the application, it continues working as if nothing had changed. In failover mode, the ADF runtime saves an additional browser cookie that the ADF runtime uses to track the latest passivation snapshot ID for each client running an application in failover mode. So, even though the users next request will be processed in the context of a new `HttpSession` (perhaps even in a different application server instance), the user is unaware that this has occurred. The additional browser cookie is used to reactivate any available application module instance with the user's last pending state snapshot before handling the request.

> **Note:** If an application module was released with reserved level then the `HttpSession` times out, the user will have to go through authentication process, and all unsaved changes are lost.

28.4.4 Approaches for Timing Out the HttpSession

Since HTTP is a stateless protocol, the server receives no implicit notice that a client has closed his browser or gone away for the weekend. Therefore any J2EE-compliant server provides a standard, configurable session timeout mechanism to allow resources tied to the HTTP session to be freed when the user has stopped performing requests. You can also programmatically force a timeout.

28.4.4.1 Configuring the Implicit Timeout Due to User Inactivity

You configure the session timeout threshold using the <session-timeout> tag in the `web.xml` file. The default value is 35 minutes. When the `HttpSession` times out the `BindingContext` goes out of scope, and along with it, any data controls that might have referenced application modules released to the pool in the managed state level. The application module pool resets any of these referenced application modules and marks the instances unreferenced again.

28.4.4.2 Coding an Explicit HttpSession Timeout

To end a user's session before the session timeout expires, you can call the `invalidate()` method on the `HttpSession` object from a backing bean in response to the user's click on a "Logout" button or link. This cleans up the `HttpSession` in the same way as if the session time had expired. Using JSF and ADF, after invalidating the session, you *must* perform a redirect to the next page you want to display, rather than just doing a forward. Example 28–5 shows the code used by the `SRLogout.java` backing bean in the SRDemo application to perform this task.

Example 28–5 Programatically termin

```
// In SRLogout.java, backing bean for the logout.jspx page
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;
}
```

As with the implicit timeouts, when the HTTP session is cleaned up this way, it ends up causing any referenced application modules to be marked unreferenced.

28.4.5 Cleaning Up Temporary Storage Tables

JDeveloper supplies the `bc4jcleanup.sql` script in the `./BC4J/bin` directory to help with periodically cleaning up the state management table. Persistent snapshot records can accumulate over time if the server has been shutdown in an abnormal way, such as might occur during development or due to a server failure. Running the script in SQL*Plus will create the `BC4J_CLEANUP` PL/SQL package. The two relevant procedures in this package are:

- `PROCEDURE Session_State( olderThan DATE )`

  This procedure cleans-up application module session state storage for sessions older than a given date.

- `PROCEDURE Session_State( olderThan_minutes INTEGER )`

  This procedures cleans-up application module session state storage for sessions older than a given number of minutes.

You can schedule periodic cleanup of your ADF temporary persistence storage by submitting an invocation of the appropriate procedure in this package as a database job.

You can use an anonymous PL/SQL block like the one shown in Example 28–6 to schedule the execution of `bc4j_cleanup.session_state()` to run starting tomorrow at 2:00am and each day thereafter to cleanup sessions whose state is over 1 day (1440 minutes) old.

Example 28–6 Scheduling Periodic Cleanup of the State Management Table

```
SET SERVEROUTPUT ON
DECLARE
  jobId    BINARY_INTEGER;
  firstRun DATE;
BEGIN
  -- Start the job tomorrow at 2am
  firstRun := TO_DATE(TO_CHAR(SYSDATE+1,'DD-MON-YYYY')||' 02:00',
              'DD-MON-YYYY HH24:MI');
   -- Submit the job, indicating it should repeat once a day
  dbms_job.submit(job       => jobId,
                  -- Run the BC4J Cleanup for Session State
                  -- to cleanup sessions older than 1 day (1440 minutes)
                  what      => 'bc4j_cleanup.session_state(1440);',
                  next_date => firstRun,
                  -- When completed, automatically reschedule
                  -- for 1 day later
                  interval  => 'SYSDATE + 1'
                 );
  dbms_output.put_line('Successfully submitted job. Job Id is '||jobId);
END;
.
/
```

28.5 Managing Custom User Specific Information

It is fairly common practice to add custom user-defined information in the application module in the form of member variables or some custom information stored in `oracle.jbo.Session` user data hashtable. The ADF state management facility provides a mechanism to save this custom information to the passivation snapshot as well. By overriding the following two methods on the `ApplicationModuleImpl` class, you can write out and read back your custom information:

```
protected void passivateState(Document doc, Element parent)
public void activateState(Element elem)
```

Consider the following simple example that shows how to override this pair of methods to ensure custom application module state is included in the passivation/activation cycle. Assume that you would like to keep some custom parameter in your application module called `jbo.counter` whose value you want to preserve across passivation and activation of the application module state. Each application module has an `oracle.jbo.Session` object associated with it that stores application module-specific session-level state. The session contains a "user data" hashtable where you can store transient information. For the user-specific data to "survive" across application module passivation and reactivation, you need to write code to save and restore this custom value into the application module state passivation snapshot. Example 28–7 shows the code you can write in your pair of overridden `passivateState()` and `activateState()` methods in a custom application module class to do the job.

The overridden `passivateState()` method performs the following steps:

1. Retrieve the value of the value to save.
2. Create an XML element to contain the value.
3. Create an XML text node to represent the value.
4. Append the text node as a child of the element.
5. Append the element to the parent element passed in.

Note: The API's used to manipulate nodes in an XML document are provided by the Document Object Model (DOM) interfaces in the `org.w3c.dom` package. These are part of the Java API for XML Processing (JAXP). See the JavaDoc for the `Node`, `Element`, `Text`, `Document`, and `NodeList` interfaces in this package for more details.

The overridden `activateState()` method performs the reverse job by doing the following:

1. Search the element for any <jbo.counter> elements.
2. If any are found, loop over the nodes found in the node list.
3. Get first child node of the <jbo.counter> element.

   It should be a DOM Text node whose value is the string you saved when your `passivateState()` method above got called, representing the value of the `jbo.counter` attribute.
4. Set the counter value to the activated value from the snapshot.

Example 28–7 Passivating and Activating Custom Information in the State Snapshot XML Document

```
/**
 * Overridden framework method to passivate custom XML elements
 * into the pending state snapshot document
 */
public void passivateState(Document doc, Element parent) {
  // 1. Retrieve the value of the value to save
  int counterValue = getCounterValue();
  // 2. Create an XML element to contain the value
  Node node = doc.createElement(COUNTER);
  // 3. Create an XML text node to represent the value
  Node cNode = doc.createTextNode(Integer.toString(counterValue));
  // 4. Append the text node as a child of the element
  node.appendChild(cNode);
  // 5. Append the element to the parent element passed in
  parent.appendChild(node);
}
/**
 * Overridden framework method to activate  custom XML elements
 * into the pending state snapshot document
 */
public void activateState(Element elem) {
  super.activateState(elem);
  if (elem != null) {
    // 1. Search the element for any <jbo.counter> elements
    NodeList nl = elem.getElementsByTagName(COUNTER);
    if (nl != null) {
```

```
      // 2. If any found, loop over the nodes found
      for (int i=0, length = nl.getLength(); i < length; i++) {
        // 3. Get first child node of the <jbo.counter> element
        Node child = nl.item(i).getFirstChild();
        if (child != null) {
          // 4. Set the counter value to the activated value
          setCounterValue(new Integer(child.getNodeValue()).intValue()+1);
          break;
        }
      }
    }
  }
}
/*
 * Helper Methods
 */
private int getCounterValue() {
  String counterValue = (String)getSession().getUserData().get(COUNTER);
  return counterValue == null ? 0 : Integer.parseInt(counterValue);
}
private void setCounterValue(int i) {
  getSession().getUserData().put(COUNTER,Integer.toString(i));
}
private static final String COUNTER = "jbo.counter";
```

Note: Similar methods are available on the `ViewObjectImpl` class and the `EntityObjectImpl` class to save custom state for those objects to the passivation snapshot as well.

28.6 Managing State for Transient View Objects

Each view object can be declaratively configured to be passivation-enabled or not by using the **Passivate State** checkbox on the **Tuning** page of the View Object Editor. If a view object is not passivation enabled, then no information about it gets written in the application module passivation snapshot.

For passivation/activation purposes, both transient and SQL-calculated view object attributes are treated in the same way.

Transient view object attributes are not passivated by default. Due to their nature, they are usually intended to be "read only" and are very easily recreateable. So, it often doesn’t make sense to passivate their values as part of the XML snapshot. However, by checking the **Passivate** checkbox on the **Attribute** page of the View Object Editor for any transient attribute, you can declaratively configure it to be passivation-enabled.

By default, all view objects are marked as passivation-enabled, and all transient attributes are not. That means that a transient view object — one that contains only transient attributes — is marked to be passivation enabled, but only passivates its information related to the current row and other non-transactional state.

It is worth noting that passivating transient view object attributes is more costly resource-wise and performance- wise, because transactional functionality is usually managed on the entity object level. Since transient view objects are not based on an entity object, this means that all updates are managed in the view object row cache and not in entity cache. Therefore, passivating transient view objects or transient view object attributes requires special runtime handling.

Usually passivation only saves the values that have been changed, but with transient view objects passivation has to save entire row. The row will only include the view object attributes marked for passivation).

28.7 Using State Management for Middle-Tier Savepoints

In the database server you are likely familiar with the savepoint feature that allows a developer to rollback to a certain point within a transaction instead of rolling back the entire transaction. An application module offers the same feature but implemented in the middle tier. There are three methods in `oracle.jbo.ApplicationModule` interface that allow you to take advantage of this feature. The methods are:

```
public String passivateStateForUndo(String id,byte[] clientData,int flags)
public byte[] activateStateForUndo(String id,int flags)
public boolean isValidIdForUndo(String id)
```

You can use these methods to create a stack of named snapshots and restore the pending transaction state from them by name. Keep in mind that those snapshots do not survive past duration of transaction (i.e. events of commit or rollback). This feature could be used to develop complex capabilities of the application, such as the ability to undo and redo changes. Another ambitious goal that could exploit this functionality would be functionality to make the browser back and forward buttons behave in an application-specific way. Otherwise, simple uses of these methods can come quite in handy.

28.8 Testing to Ensure Your Application Module is Activation-Safe

If you have not *explicitly* tested that your application module functions when its pending state gets activated from a passivation snapshot, then you may encounter an unpleasant surprise in your production environment when heavy system load "tests" this aspect of your system for the first time.

28.8.1 Understanding the jbo.ampool.doampooling Configuration Parameter

The `jbo.ampool.doampooling` configuration property corresponds to the **Enable Application Module Pooling** option in the **Pooling and Scalability** tab of the Configuration Editor. By default, this checkbox is checked so that application module pooling is enabled. Nearly always this default setting of `jbo.ampool.doampooling` to `true` is the way you will run your applications in production. However, setting the property to `false` should play an important role in your testing. When this property is false, there is effectively no application pool. When the application module instance is released at the end of a request it is immediately removed. On subsequent requests made by the same user session, a new application module instance must be created to handle it and the pending state of the application module must be reactivated from the passivation store.

28.8.2 Disabling Application Module Pooling to Test Activation

Oracle recommends, as part of your overall testing plan, that you adopt the practice of testing your application modules with the `jbo.ampool.doampooling` configuration parameter set to `false`. This setting completely disables application module pooling and forces the system to activate your application module's pending state from a passivation snapshot on *each* page request. It is an excellent way to detect problems that might occur in your production environment due to assumptions made in your custom application code.

For example, if you have transient view object attributes you believe *should* be getting passivated, this technique allows you to test that they are working as you expect. In addition, consider situations where you might have introduced:

- Private member fields in application modules, view objects, or entity objects
- Custom user session state in the `Session` user data hashtable

Your custom code likely assumes that this custom state will be maintained across HTTP requests. As long as you test with a single user on the JDeveloper embedded OC4J server, or test with a small number of users, things will appear to work fine. This is due to the "stateless with affinity" optimization of the ADF application module pool. If system load allows, the pool will continue to return the same application module instance to a user on subsequent requests. However, under heavier load, during real-world use, it may not be able to achieve this optimization and will need to resort to grabbing any available application module instance and reactivating its pending state from a passivation snapshot. If you have not correctly overridden `passivateState()` and `activateState()` (as described in Section 28.5, "Managing Custom User Specific Information") to save and reload your custom component state to the passivation snapshot, then your custom state will be missing (i.e. `null` or back to your default values) after this reactivation step. Testing with `jbo.ampool.doampooling` set to false allows you to quickly isolate these kinds of situations in your code.

28.9 Caveats Regarding Pending Database State

As you have seen, the ADF state management mechanism relies on passivation and activation to manage the state of an application module instance. Implementing this feature in a robust way is only possible if all pending changes are managed by the application module transaction in the middle tier. The most scalable strategy is to keep pending changes in middle-tier objects and not perform operations that cause pending database state to exist across HTTP requests. This allows the highest leverage of the performance optimizations offered by the application module pool and the most robust runtime behavior for your application.

28.9.1 Web Applications Should Use Optimistic Locking

Oracle recommends using optimistic locking for web applications. Pessimistic locking, which is the default, should not be used for web applications as it creates pending transactional state in the database in the form of row-level locks. If pessimistic locking is set, state management will work, but the locking mode will not perform as expected. Behind the scenes, every time an application module is recycled, a rollback is issued in the JDBC connection. This would release all the locks that pessimistic locking had created.

To change your configuration to use optimistic locking, open the **Properties** tab of the Configuration Editor and set the value of the `jbo.locking.mode` to optimistic.

28.9.2 Use PostChanges Only During the Current Request

As you saw in Section 9.2.2, "Understanding Commit Processing and Validation", using the `postChanges()` method needs to be done with careful consideration, if at all. Nowhere is this more true than when considering state management, application pooling, and database connection pooling. The `postChanges()` method should only be used as part of the commit processing lifecycle so that any pending database state it creates is committed or rolled back during the same request.

28.9.3 Pending Database State Across Requests Requires Reserved Level

If for some reason you need to create a transactional state in the database in some request by invoking `postChanges()` method or by calling PL/SQL stored procedure, but you cannot issue a commit or rollback by the end of that same request, then you *must* release the application module instance with the **reserved** level from that request until a subsequent request when you either commit or rollback. Oracle recommends that this be as short a period of time as possible between creation of transactional state in the database performing the concluding commit or rollback, so that reserved level doesn't have to be used for long time. This is due to the fact that the reserved level has adverse effects on application's scalability and reliability.

Once an application module has been released with reserved level, it remains at that release level for all subsequent requests until release level is explicitly changed back to managed or unmanaged level. So, it is your responsibility to set release level back to managed level once commit or rollback has been issued.

28.9.4 Connection Pooling Prevents Pending Database State

When you check the **Disconnect Application Module Upon Release** property on the **Pooling and Scalability** tab of the Configuration Editor, this translates to setting the `jbo.doconnectionpooling` configuration parameter to `true`.

With this connection pooling option enabled — typically in order to share a common pool of database connections across multiple application module pools — upon releasing your application module to the application module pool, its JDBC connection is released back to the database connection pool and a `ROLLBACK` will be issued on that connection. This implies that all changes which were posted but *not commited* will be lost. On the next request, when the application module is used, it will receive a JDBC connection from the pool, which may be a different JDBC connection instance from the one it used previously. Those changes that were posted to the database but not commited during the previous request are no longer there.

29

Understanding Application Module Pooling

This chapter describes how ADF Business Components application module pools work and how you can tune the pools to optimize application performance.

This chapter includes the following sections:

- Section 29.1, "Overview of Application Module Pooling"
- Section 29.2, "Lifecycle of a Web Page Request Using Oracle ADF and JSF"
- Section 29.3, "Understanding Configuration Property Scopes"
- Section 29.4, "Setting Pool Configuration Parameters"
- Section 29.5, "How Many Pools are Created, and When?"
- Section 29.6, "Application Module Pool Parameters"
- Section 29.7, "Database Connection Pool Parameters"
- Section 29.8, "How Database and Application Module Pools Cooperate"
- Section 29.9, "Database User State and Pooling Considerations"

29.1 Overview of Application Module Pooling

An application module pool is a collection application module instances of the same type. For example, the SRDemo application has one or more instances of the SRService application module in it, based on the number of users that are visiting the site. This pool of application module instances is shared by multiple browser clients whose typical "think time" between submitting web pages allows optimizing the number of application module components to be effectively smaller than the total number of active users working on the system. That is, twenty users visiting the web site from their browser might be able to be serviced by 5 or 10 application module instances instead of having as many application module instances as you have browser users.

Application module components can be used to support web application scenarios that are completely stateless, or they can be used to support a unit of work that spans multiple browser pages. As a performance optimization, when an instance of an application module is returned to the pool in "managed state" mode, the pool keeps track that the application module is referenced by that particular session. The application module instance is still in the pool and available for use, but it would *prefer* to be used by the same session that was using it last time because maintaining this so-called "session affinity" improves performance.

So, at any one moment in time, the instances of application modules in the pool are logically partitioned into three groups, reflecting their state:

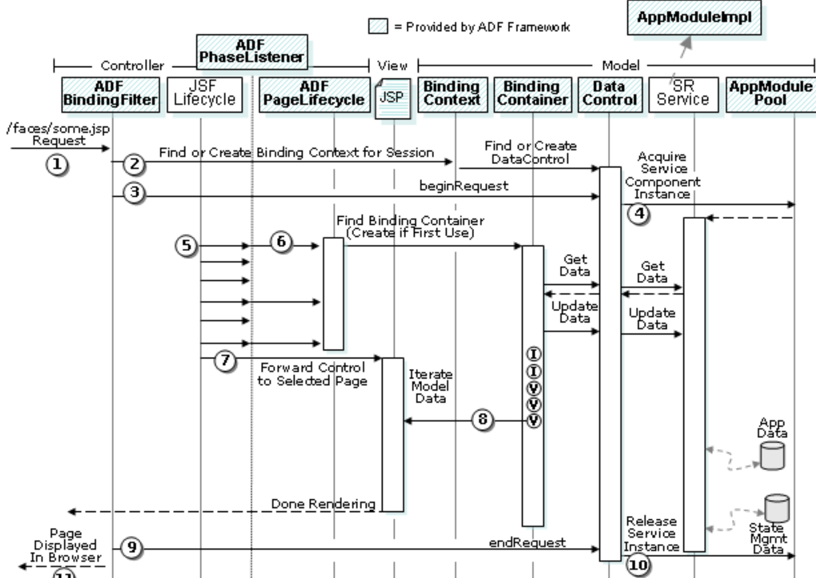
- Unconditionally *available* for use
- Available for use, but *referenced* for session affinity reuse by an active user session
- *Unavailable*, inasmuch as it's currently in use (at that very moment) by some thread in the web container.

Section 29.3, "Understanding Configuration Property Scopes" describes the application module pool configuration parameters and how they affect the behavior of the pool.

29.2 Lifecycle of a Web Page Request Using Oracle ADF and JSF

Figure 29–1 shows a sequence diagram of the lifecycle of a web page request using JSF and Oracle ADF in tandem.

Figure 29–1 Lifecycle of a Web Page Request Using JSF and Oracle ADF



As shown in the figure, the basic flow of processing happens as follows:

1. A web request for `http://yourserver/yourapp/faces/some.jsp` arrives from the client to the application server
2. The `ADFBindingFilter` finds the ADF binding context in the HTTP session, and if not yet present, initializes it for the first time.

   During binding context initialization, the `ADFBindingFilter`:

   - Consults the servlet context initialization parameter named `CpxFileName` and appends the `*.cpx` file extension to its *value* to determine the name of the binding context metadata file. By default the parameter value will be "`DataBindings`", so it will look for a file named `DataBindings.cpx`.
   - Reads the binding context metadata file to discover the data control definitions, the page definition file names used to instantiate binding containers at runtime, and the page map that relates a JSP page to its page definition file.

- Constructs an instance of each Data Control, and a reference to each BindingContainer. The *contents* of each binding container are loaded lazily the first time they are used by a page.

3. The `ADFBindingFilter` invokes the `beginRequest()` method on each data control participating in the request. This gives every data control a notification at the start of every request where they can perform any necessary setup.

4. An application module data control uses the `beginRequest` notification to acquire an instance of the application module from the application module pool.

5. The JSF `Lifecycle` class, which is responsible for orchestrating the standard processing phases of each request, notifies the `ADFPhaseListener` class during each phase of the lifecycle so that it can perform custom processing to coordinate the JSF lifecycle with the Oracle ADF Model data binding layer.

> **Note:** The `FacesServlet` (in `javax.faces.webapp`) is configured in the `web.xml` file of a JSF application and is responsible for initially creating the JSF `Lifecycle` class (in `javax.faces.lifecycle`) to handle each request. However, since it is the `Lifecycle` class that does all the interesting work, the FacesServlet is not shown in the diagram.

6. The `ADFPhaseListener` creates an ADF `PageLifecycle` object to handle each request and delegates appropriate before/after phase methods to corresponding methods in the ADF `PageLifecycle` class as shown in Figure 29–2. If the appropriate binding container for the page has never been used before during the user's session, it is created.

Figure 29–2 How JSF Page Lifecycle and ADF Page Lifecycle Phases Relate

7. The JSF `Lifecycle` forwards control to the page to be rendered.

8. The UI components on the page access value bindings and iterator bindings in the page's binding container and render the formatted output to appear in the browser.

9. The `ADFBindingFilter` invokes the `endRequest()` method on each data control participating in the request. This gives every data control a notification at the end of every request where they can perform any necessary resource cleanup.

10. An application module data control uses the `endRequest` notification to release the instance of the application module back to the application module pool.

11. The user sees the resulting page in the browser.

29.3 Understanding Configuration Property Scopes

Each runtime configuration property used by ADF Business Components has a scope. The scope of each property indicates at what level the property's value is evaluated and whether its value is effectively shared (i.e. static) in a single Java VM, or not. The ADF Business Components `PropertyManager` class is the registry of all supported properties. It defines the property names, their default values, and their scope. This class contains a `main()` method so that you can run the class from the command line to see a list of all the configuration property information.

Assuming JDEVHOME is the JDeveloper 10g installation directory, to see this list of settings for reference, do the following:

```
$ java -cp JDEVHOME/BC4J/lib/bc4jmt.jar oracle.jbo.common.PropertyManager
```

Issuing this comman will send all of the ADF Business Components configuration properties to the console. It also lists a handy reference about the different levels at which you can set configuration property values and remind you of the precedence order these levels have:

```
---------------------------------------------------------------
Properties loaded from following sources, in order:
1. Client environment [Provided programmatically
                       or declaratively in bc4j.xcfg]
2. Applet tags
3. -D flags (appear in System.properties)
4. bc4j.properties file (in current directory)
5. /oracle/jbo/BC4J.properties resource
6. /oracle/jbo/commom.jboserver.properties resource
7. /oracle/jbo/common.Diagnostic.properties resource
8. System defined default
---------------------------------------------------------------
```

You'll see each property is listed with one of the following scopes:

- MetaObjectManager

  Properties at this scope are initialized once per Java VM when the ADF PropertyManager is first initialized.

- SessionImpl

  Properties at this scope are initialized once per invocation of `ApplicationModule.prepareSession()`.

- Configuration

  Properties at this scope are initialized when the ApplicationModule pool is first created and the application module's configuration is read the first time.

- Diagnostic

  Properties at this scope are specific to the built-in ADF Business Components diagnostic facility.

At each of these scopes, the layered value resolution described above is performed when the properties are initialized. Whenever property values are initialized, if you have specified them in the *Client Environment* (level 1 in the resolution order) the values will take precedence over values specified as System parameters (level 3 in the resolution order).

The Client Environment is a hashtable of name/value pairs that you can either programatically populate, or which will be automatically populated for you by the `Configuration` object when loaded, with the name/value pairs it contains in its entry in the `bc4j.xcfg` file. The implication of this is that for any properties scoped at MetaObjectManager level, the most reliable way to ensure that all of your application modules use the same default value for those properties is to do both of the following:

1. Make sure the property value does not appear in any of your application module's `bc4j.xcfg` file configuration name/value pair entries.
2. Set the property value using a Java system property in your runtime environment.

If, instead, you leave any MetaObjectManager-scoped properties in your `bc4j.xcfg` files, you will have the undesirable behavior that they will take on the value specified in the configuration of the *first* application module whose pool gets created after the Java VM starts up.

29.4 Setting Pool Configuration Parameters

You control the runtime behavior of an application module pool by setting appropriate configuration parameters. You can set these declaratively in an application module configuration, supply them as Java System parameters, or set them programmatically at runtime.

29.4.1 Setting Configuration Properties Declaratively

The **Pooling and Scalability** tab of the Configuration Editor shown in Figure 29–3 is used for seeing and setting parameters.

Figure 29–3 Pooling and Scalability Tab of the Configuration Manager

The values that you supply through the **Configuration Manager** are saved in an XML file named bc4j.xcfg in the ./common subdirectory relative to the application module's XML component definition. All of the configurations for all of the application modules in a single Java package are saved in that same file.

For example, if you look at the bc4j.xcfg file in the ./oracle/srdemo/model/common directory of the SRDemo application's DataModel project, you will see the two named configurations for its SRService application module, as shown in Example 29–1. The SRServiceLocalTesting configuration uses JDBC URL connections for use by the SRDemo application’s JUnit tests and by the Business Components Browser. The SRServiceLocal uses JDBC Datasource names and is used by the web application.

Example 29–1 Configuration Settings for the SRService Application Module

```
<BC4JConfig>
   <AppModuleConfigBag>
      <AppModuleConfig name="SRServiceLocal">
         <DeployPlatform>LOCAL</DeployPlatform>
         <JDBCDataSource>java:comp/env/jdbc/SRDemoDS</JDBCDataSource>
         <jbo.project>DataModel</jbo.project>
         <jbo.locking.mode>optimistic</jbo.locking.mode>
         <AppModuleJndiName>oracle.srdemo.model.SRService</AppModuleJndiName>
         <jbo.security.enforce>Must</jbo.security.enforce>
         <ApplicationName>oracle.srdemo.model.SRService</ApplicationName>
         <jbo.server.internal_connection
           >java:comp/env/jdbc/SRDemoCoreDS</jbo.server.internal_connection>
      </AppModuleConfig>
      <AppModuleConfig name="SRServiceLocalTesting">
         <DeployPlatform>LOCAL</DeployPlatform>
         <JDBCName>SRDemo</JDBCName>
         <jbo.project>DataModel</jbo.project>
         <AppModuleJndiName>oracle.srdemo.model.SRService</AppModuleJndiName>
         <jbo.locking.mode>optimistic</jbo.locking.mode>
         <jbo.security.enforce>Must</jbo.security.enforce>
         <ApplicationName>oracle.srdemo.model.SRService</ApplicationName>
      </AppModuleConfig>
   </AppModuleConfigBag>
   <ConnectionDefinition name="SRDemo">
      <ENTRY name="JDBC_PORT" value="1521"/>
      <ENTRY name="ConnectionType" value="JDBC"/>
      <ENTRY name="HOSTNAME" value="localhost"/>
      <ENTRY name="DeployPassword" value="true"/>
      <ENTRY name="user" value="srdemo"/>
      <ENTRY name="ConnectionName" value="SRDemo"/>
      <ENTRY name="SID" value="XE"/>
      <ENTRY name="password">
         <![CDATA[{904}05708016F4BB90FC04CFE36B6C9D2BDFE5]]>
      </ENTRY>
      <ENTRY name="JdbcDriver" value="oracle.jdbc.OracleDriver"/>
      <ENTRY name="ORACLE_JDBC_TYPE" value="thin"/>
      <ENTRY name="DeployPassword" value="true"/>
   </ConnectionDefinition>
</BC4JConfig>
```

Note that child elements of the <AppModuleConfig> tag appear with tag names matching their property values. It's also important to understand that if a property is currently set to its runtime *default* value, then the **Configuration Manager** does *not* write the entry to the bc4j.xcfg file.

29.4.2 Setting Configuration Properties as System Parameters

As an alternative to specifying configuration properties in the `bc4j.xcfg` file, you can also set Java VM system parameters with the same property names. These system parameters will be used *only if* a corresponding property *does not exist* in the relevant `bc4j.xcfg` file for the application module in question. In other words, configuration parameters that appear in the application module configuration take precedence over parameters of the same name supplied as Java system parameters.

You typically set Java system parameters using the `-D` command line flag to the Java VM like this:

```
java -Dproperty=value -jar yourserver.jar
```

Alternatively, your J2EE container probably has a section in its own configuration files where Java system parameters can be specified for use at J2EE container startup time.

Many customers adopt the best practice technique of supplying site-specific *default* values for ADF configuration parameters as Java system parameters and then make sure that their `bc4j.xcfg` files do *not* include references to these parameters unless an application-module-specific exception to these global default values is required.

Using OC4J you can specify these parameters either as `-D` Java system parameters on the command line that starts OC4J, or provide them — one per line — in the `oc4j.properties` file and add the `-properties oc4j.properties` command line flag to OC4J at startup.

Caution: The values of **Idle Instance Timeout**, **Pool Polling Interval** settings for both the **Application Pool** and the database **Connection Pool** are displayed and edited in this dialog as a number of *seconds*, but are saved to the configuration file in milliseconds. If you provide a value for any of these four parameters as a Java System parameter — or if you hand-edit the `bc4j.xcfg` file — make sure to provide these time interval values in milliseconds!

29.4.3 Programmatically Setting Configuration Properties

You can set configuration properties programmatically by creating a Java class that implements the `EnvInfoProvider` interface in the `oracle.jbo.common.ampool` package. In your class, you override the `getInfo()` method and call `put()` to put values into the environment Hashtable passed in as shown in Example 29–2

Example 29–2 Setting Environment Properties with a Custom EnvInfoProvider

```
package devguide.advanced.customenv.view;
import java.util.Hashtable;
import oracle.jbo.common.ampool.EnvInfoProvider;
/**
 * Custom EnvInfoProvider implementation to set
 * environment properties programmatically
 */
public class CustomEnvInfoProvider implements EnvInfoProvider {
  /**
   * Overridden framework method to set custom values in the
   * environment hashtable.
   *
   * @param string - ignore
   * @param environment Hashtable of config parameters
   * @return null - not used
   */
  public Object getInfo(String string, Object environment) {
    Hashtable envHashtable = (Hashtable)environment;
    envHashtable.put("some.property.name","some value");
    return null;
  }
  /* Required to implement EnvInfoProvider */
  public void modifyInitialContext(Object object) {}
  /* Required to implement EnvInfoProvider */
  public int getNumOfRetries() {return 0;}
}
```

When creating an application module for a stateless or command-line-client, with the `createRootApplicationModule()` method of the `Configuration` class, you can pass the custom EnvInfoProvider as the optional second argument. In order to use a custom EnvInfoProvider in an ADF web-based application, you need to implement a custom session cookie factory class as shown in Example 29–3. To use your custom session cookie factory, set the `jbo.ampool.sessioncookiefactoryclass` configuration property to the fully-qualified name of your custom session cookie factory class.

Example 29–3 Custom SessionCookieFactory to Install a Custom EnvInfoProvider

```
package devguide.advanced.customenv.view;
import java.util.Properties;
import oracle.jbo.common.ampool.ApplicationPool;
import oracle.jbo.common.ampool.EnvInfoProvider;
import oracle.jbo.common.ampool.SessionCookie;
import oracle.jbo.http.HttpSessionCookieFactory;
/**
 * Example of custom http session cookie factory
 * to install a custom EnvInfoProvider implementation
 * for an ADF web-based application.
 */
public class CustomHttpSessionCookieFactory
       extends HttpSessionCookieFactory {
  public SessionCookie createSessionCookie(String appId,
                                           String sessionId,
                                           ApplicationPool pool,
                                           Properties props) {
    SessionCookie cookie =
      super.createSessionCookie(appId, sessionId,pool, props);
    EnvInfoProvider envInfoProv = new CustomEnvInfoProvider();
    cookie.setEnvInfoProvider(envInfoProv);
    return cookie;
  }
}
```

29.5 How Many Pools are Created, and When?

There are two kinds of pools in use when running a typical ADF web application, Application Module pools and database connection pools. It's important to understand how many of each kind of pool your application will create.

29.5.1 Application Module Pools

Application Module components can be used at runtime in two ways:

- As an application module the client accesses directly
- As a reusable component aggregated (or "nested") inside of another application module instance

When a client accesses it directly, an application module is called a *root application module*. Clients access nested application modules *indirectly* as a part of their containing application module instance. It's possible, but not common, to use the same application module at runtime in both ways. The important point is that ADF only creates an application module pool for a *root* application module.

The basic rule is that one application module pool is created for each *root* application module used by an ADF web application in *each* Java VM where a root application module of that type is used by the ADF controller layer.

29.5.2 Database Connection Pools

ADF web applications always use a database connection pool, but which one they use for your application modules depends on how they define their connection:

- JDBC URL (e.g. `jdbc:oracle:thin:@penguin:1521:ORCL`)
- JNDI Name for a Datasource (e.g. `java:comp/env/jdbc/YourConnectionDS`)

If you supply a JDBC URL connection while configuring your application module — which happens when you select a JDeveloper named connection which encapsulates the JDBC URL and username information — then the ADF database connection pool will be used for managing the connection pool.

If you supply the JNDI name of a JDBC Datasource then the ADF database connection pool will *not* be used and the configuration parameters described below relating to the ADF database connection pool are not relevant.

Note: To configure the database connection pool for JDBC Datasources looked-up by JNDI from your J2EE Web and/or EJB container, consult the documentation for your J2EE container to understand the pooling configuration options and how to set them.

When using ADF database connection pooling, you have the following basic rule: One database connection pool is created for each unique <JDBCURL,Username> pair, in *each* Java VM where a <JDBCURL,Username> connection is requested by a root application used by the ADF controller layer.

29.5.3 Understanding Application Module and Connection Pools

Section 29.5.3.1, "Single Oracle Application Server Instance, Single OC4J Container, Single JVM" and Section 29.5.3.2, "Multiple Oracle Application Server Instances, Single OC4J Container, Multiple JVMs" illustrate how application module pools and connection pools are created in different scenarios. Both of these sections make the following assumptions:

- Your web application makes use of two application modules `HRModule` and `PayablesModule`.
- You have a `CommonLOVModule` containing a set of commonly used view objects to support list-of-values in your application, and that both `HRModule` and `PayablesModule` aggregate a nested instance of `CommonLOVModule` to access the common LOV view objects it contains.
- You have configured both `HRModule` and `PayablesModule` to use the same JDeveloper connection definition named `appuser`.
- In both `HRModule` and `PayablesModule` you have configured `jbo.passivationstore=database` (the default) and configured the ADF "internal connection" (`jbo.server.internal_connection`) used for state management persistence to have the value of a fully-qualified JDBC URL that points to a *different* username than the `appuser` connection does.

Consider how many pools of which kinds are created for this application in both a single JVM and multiple JVM runtime scenario.

29.5.3.1 Single Oracle Application Server Instance, Single OC4J Container, Single JVM

If you deploy this application to a single Oracle Application Server instance, configured with a single OC4J container having a single Java VM, there is only a single Java VM available to service the web requests coming from your application users.

Assuming that all the users are making use of web pages that access both the `HRModule` and the `PayablesModule`, this will give:

- One application module pool for the `HRModule` root application module
- One application module pool for the `PayablesModule` root application module
- One DB connection pool for the `appuser` connection
- One DB connection pool for the JDBC URL supplied for the internal connection for state management.

This gives a total of two application module pools and two database pools in this single Java VM.

Note: There is no separate application module pool for the nested instances of the reusable `CommonLOVModule`. Instances of CommonLOVModule are wrapped by instances of `HRModule` and `PayablesModule` in their respective application module pools.

29.5.3.2 Multiple Oracle Application Server Instances, Single OC4J Container, Multiple JVMs

Next consider a deployment environment involving multiple Java VMs. Assume that you have installed Oracle Application Server 10g (version 9.0.4) onto two different physical machines, with a hardware load-balancer in front of it. On each of these two machines, imagine that the Oracle Application Server instance is configured to have one OC4J container with two JVMs. As users of your application access the application, their requests are shared across these two Oracle Application Server instances, and within each Oracle Application Server instance, across the two JVMs that its OC4J container has available.

Again assuming that all the users are making use of web pages that access both the `HRModule` and the `PayablesModule`, this will give:

- Four application module pools for `HRModule`, one in each of four JVMs.

  (1 `HRModule` root application module) x (2 Oracle Application Server Instances) x (2 OC4J JVMs each)

- Four application module pools for `PayablesModule`, one in each of four JVMs.

  (1 `PayablesModule` root application module) x (2 Oracle Application Server Instances) x (2 OC4J JVMs each)

- Four DB connection pools for `appuser`, one in each of four JVMs.

  (1 `appuser` DB connection pool) x (2 Oracle Application Server Instances) x (2 OC4J JVMs each)

- Four DB connection pools for the internal connection JDBC URL, one in each of four JVMs.

  (1 internal connection JDBC URL DB connection pool) x (2 Oracle Application Server Instances) x (2 OC4J JVMs each)

This gives a total of eight application module pools and eight DB connection pools spread across four JVMs.

As you begin to explore the configuration parameters for the application module pools in Section 29.6, "Application Module Pool Parameters", keep in mind that the parameters apply to a given application module pool for a given application module in a single JVM.

As the load balancing spreads user request across the multiple JVMs where ADF is running, each individual application module pool in each JVM will have to support one N^{th} of the user load — where N is number of JVMs available to service those user requests. The appropriate values of the application module and DB connection pools need to be set with the number of Java VMs in mind.

29.6 Application Module Pool Parameters

The application module pool configuration parameters fall into three logical categories relating to pool behavior, pool sizing, and pool cleanup behavior.

29.6.1 Pool Behavior Parameters

Table 29–2 lists the application module configuration parameters that affect the behavior of the application module pool.

Table 29–1 Application Module Pool Behavior Configuration Parameters

Pool Configuration Parameter	Description
Failover Transaction State Upon Managed Release (`jbo.dofailover`)	Enables eager passivation of pending transaction state each time an application module is released to the pool in "Managed State" mode. See Section 28.2.3, "How Passivation Changes When Optional Failover Mode is Enabled" for more information. This feature is disabled by default (`false`)
Disconnect Application Module Upon Release (`jbo.doconnectionpooling`)	Forces the application module pool to release the JDBC connection used each time the application module is released to the pool. See Section 29.8, "How Database and Application Module Pools Cooperate" for more information. This feature is disabled by default (`false`).
Support Dynamic JDBC Credentials (`jbo.ampool.dynamicjdbccredentials`)	Enables additional pooling lifecycle events to allow developer-written code to change the database credentials (username/password) each time a new user session begins to use the application module. This feature is enabled by default (`true`), however this setting is a necessary but not sufficient condition to implement the feature. The complete implementation requires additional developer-written code.
Reset Non-Transactional State Upon Unmanaged Release (`jbo.ampool.resetnontransactionalstate)`)	Forces the application module to reset any non-transactional state like view object runtime settings, JDBC prepared statements, bind variable values, etc. when the application module is released to the pool in unmanaged or "stateless" mode. This feature is enabled by default (`true`). Disabling this feature can improve performance, however since it does not clear bind variable values, your application needs to ensure that it systemically sets bind variable values correctly. Failure to do so with this feature disabled can mean one user might see data with another users bind variable values.)

Table 29–1 (Cont.) Application Module Pool Behavior Configuration Parameters

Pool Configuration Parameter	Description
Enable Application Module Pooling (`jbo.ampool.timetolive`)	Enables the application module pooling facility. See Section 28.8, "Testing to Ensure Your Application Module is Activation-Safe" for more information on when Oracle recommends to disable this feature as a routine part of your application testing. This feature is enabled by default (`true`)

29.6.2 Pool Sizing Parameters

Table 29–2 lists the application module configuration parameters that affect the sizing of the application module pool.

Table 29–2 Application Module Pool Sizing Configuration Parameters

Pool Configuration Parameter	Description
Initial Pool Size (`jbo.ampool.initpoolsize`)	The number of application module instances to created when the pool is initialized. The default is `0` (zero) instances.
Maximum Pool Size (`jbo.ampool.maxpoolsize`)	The maximum number of application module instances that the pool can allocate. The pool will never create more application module instances than this limit imposes. The default is `5000` instances.
Referenced Pool Size (`jbo.recyclethreshold`)	The maximum number of application module instances in the pool that attempt to preserve session affinity for the next request made by the session which used them last before releasing them to the pool in managed-state mode. The referenced pool size should always be less than or equal to the maximum pool size. The default is to allow `10` available instances to try and remain "loyal" to the affinity they have with the most recent session that released them in managed state mode.
Maximum Instance Time to Live (`jbo.ampool.timetolive`)	The number of milliseconds after which to consider an application module instance in the pool as a candidate for removal during the next resource cleanup regardless of whether it would bring the number of instances in the pool below minavailablesize. The default is 3600000 milliseconds of total time to live (which is 3600 seconds, or one hour)

29.6.3 Pool Cleanup Parameters

A single "application module pool monitor" per Java VM runs in a background thread and wakes up every so often to do resource reclamation. Table 29–3 lists the parameters that affect how resources are reclaimed when the pool monitor does one of its resource cleanup passes.

Note: Since there is only a single application monitor pool monitor per Java VM, the value that will effectively be used for the application module pool monitor polling interval will be the value found in the application module configuration read by the *first* application module pool that gets created. To make sure this value is set in a predictable way, it is best practice to set all application modules to use the same **Pool Polling Interval** value.

Table 29–3 Application Module Resource Management Configuration Parameters

Pool Configuration Parameter	Description
Pool Polling Interval (`jbo.ampool.monitorsleepinterval`)	The length of time in milliseconds between pool resource cleanup. While the number of application module instances in the pool will never exceed the maximum pool size, available instances which are candidates for getting removed from the pool do not get "cleaned up" until the next time the application module pool monitor wakes up to do its job. The default is to have the application module pool monitor wake up every `600000` milliseconds (which is 600 seconds, or ten minutes).
Maximum Available Size (`jbo.ampool.maxavailablesize`)	The ideal maximum number of application module instances in the pool when not under abnormal load. When the pool monitor wakes up to do resource cleanup, it will try to remove available application module instances to bring the total number of available instances down to this ideal maximum. Instances that have been not been used for a period longer than the idle instance time-out will always get cleaned up at this time, then additional available instances will be removed if necessary to bring the number of available instances down to this size. The default maximum available size is `25` instances.
Minimum Available Size (`jbo.ampool.minavailablesize`)	The minimum number of available application module instances that the pool monitor should leave in the pool during a resource cleanup operation. Set to zero (`0`) if you want the pool to shrink to contain no instances when all instances have been idle for longer than the idle time-out. The default is `5` instances.
Idle Instance Timeout (`jbo.ampool.maxinactiveage`)	The number of milliseconds after which to consider an inactive application module instance in the pool as a candidate for removal during the next resource cleanup. The default is 600000 milliseconds of idle time (which is 600 seconds, or ten minutes).

Table 29–3 (Cont.) Application Module Resource Management Configuration Parameters

Pool Configuration Parameter	Description
Maximum Instance Time to Live (`jbo.pooltimetolive`)	The number of milliseconds after which to consider an connection instance in the pool as a candidate for removal during the next resource cleanup regardless of whether it would bring the number of instances in the pool below minavailablesize. The default is 3600000 milliseconds of total time to live (which is 3600 seconds, or one hour)

29.7 Database Connection Pool Parameters

If you are using a JDBC URL for your connection information so that the ADF database connection pool is used, then configuration parameters listed in Table 29–4 can be used to tune the behavior of the database connection pool. A single "database connection pool monitor" per Java VM runs in a background thread and wakes up every so often to do resource reclamation. The parameters in Table 29–3 include the ones that affect how resources are reclaimed when the pool monitor does one of its resource cleanup passes.

Note: The configuration parameters for database connection pooling have MetaObjectManager scope (described in Section 29.3, "Understanding Configuration Property Scopes" earlier). This means their settings are global and will be set once when the first application module pool in your application is created. To insure the most predictable behavior, Oracle recommends leaving the values of these parameters in the **Connection Pooling** section of the **Pooling and Scalability** tab at their *default* values — so that no entry for them is written into the `bc4j.xcfg` file — and to instead set the desired values for the database connection pooling tuning parameters as Java System Parameters in your J2EE container.

Table 29–4 Database Connection Pool Parameters

Pool Configuration Parameter	Description
Initial Pool Size (`jbo.initpoolsize`)	The number of JDBC connection instances to created when the pool is initialized The default is an initial size of `0` instances.
Maximum Pool Size (`jbo.maxpoolsize`)	The maximum number of JDBC connection instances that the pool can allocate. The pool will never create more JDBC connections than this imposes. The default is `5000` instances.

Table 29–4 (Cont.) Database Connection Pool Parameters

Pool Configuration Parameter	Description
Pool Polling Interval (`jbo.poolmonitorsleepinterval`)	The length of time in milliseconds between pool resource cleanup. While the number of JDBC connection instances in the pool will never exceed the maximum pool size, available instances which are candidates for getting removed from the pool do not get "cleaned up" until the next time the JDBC connection pool monitor wakes up to do its job. The default is `600000` milliseconds of idle time (which is 600 seconds, or ten minutes).
Maximum Available Size (`jbo.poolmaxavailablesize`)	The ideal maximum number of JDBC connection instances in the pool when not under abnormal load. When the pool monitor wakes up to do resource cleanup, it will try to remove available JDBC connection instances to bring the total number of available instances down to this ideal maximum. Instances that have been not been used for a period longer than the idle instance time-out will always get cleaned up at this time, then additional available instances will be removed if necessary to bring the number of available instances down to this size. The default is an ideal maximum of `25` instances (when not under load).
Minimum Available Size (`jbo.poolminavailablesize`)	The minimum number of available JDBC connection instances that the pool monitor should leave in the pool during a resource cleanup operation. Set to zero (`0`) if you want the pool to shrink to contain no instances when all instances have been idle for longer than the idle time-out. The default is to not let the minimum available size drop below `5` instances.
Idle Instance Timeout (`jbo.poolmaxinactiveage`)	The number of seconds after which to consider an inactive JDBC connection instance in the pool as a candidate for removal during the next resource cleanup. The default is `600000` milliseconds of idle time (which is 600 seconds, or ten minutes).

Notice that since the database connection pool does not implement the heuristic of session affinity, there is *no* configuration parameter for the database connection pool which controls the referenced pool size.

You should take care *not* to configure the `jbo.ampool.monitorsleepinterval` (for the application module pools) or the `jbo.poolmonitorsleepinterval` (for the DB pools) to be too short of a time period because the chance exists — with a large number of application module pools to cleanup — that your next pool monitor "wakeup" might occur while your previous cleanup-cycle is still going on. The default of 10 minutes (`600000` milliseconds) is reasonable. Setting it to something like 10 seconds (`10000` milliseconds) might cause trouble.

29.8 How Database and Application Module Pools Cooperate

How ADF application module pools use the database connection pool depends on the setting of the `jbo.doconnectionpooling` application module configuration parameter. In the **Configuration Manager** panel that you see in Figure 29–3, you set this parameter using the checkbox labelled **Disconnect Application Module Upon Release**.

> **Note:** The notion of *disconnecting* the application module upon release to the pool better captures what the actual feature is doing than the related configuration parameter name (`jbo.doconnectionpooling`) does. The setting of `jbo.doconnectionpooling=false` does not mean that there is no database connection pooling happening. What it means is that the application module is not disconnected from its JDBC connection upon check in back to the application module pool.

If the default setting of `jbo.doconnectionpooling=false` is used, then when an application module instance is created in any pool it acquires a JDBC connection from the appropriate connection pool (based on the JDBC URL in the ADF case, or from the underlying JDBC data source implementation's pool in the case of a JNDI data source name). That application module instance holds onto the JDBC connection object that it acquired from the pool until the application module instance is removed from the application module pool. During its lifetime, that application module instance may service many different users, and ADF worries about issuing rollbacks on the database connection so that different users don't end up getting pending database state confused. By holding onto the JDBC connection, it allows each application module instance to keep its JDBC PreparedStatements's open and usable across subsequent accesses by clients, thereby providing the best performance.

If `jbo.doconnectionpooling=true`, then each time a user session finishes using an application module (typically at the end of each HTTP request), the application module instance disassociates itself with the JDBC connection it was using on that request and it returns it to the JDBC connection pool. The next time that application module instance is used by a user session, it will reacquire a JDBC connection from the JDBC connection pool and use it for the span of time that application module is checked out of the application module pool (again, typically the span of one HTTP request). Since the application module instance "unplugs" itself from the JDBC connection object used to create the PreparedStatements it might have used during the servicing of the current HTTP request, those PreparedStatements are no longer usable on the next HTTP request because they are only valid in the context of the Connection object in which they were created. So, when using the connection pooling mode turned on like this, the trade-off is slightly more JDBC overhead setup each time, in return for using a smaller number of overall database connections.

The key difference is seen when many application module pools are all using the same underlying database user for their application connection.

- If 50 different application module pools each have even just a single application module instance in them, with `jbo.doconnectionpooling=false` there will be 50 JDBC application connections in use. If the application module pooling parameters are set such that the application module pools are allowed to shrink to `0` instances after an appropriate instance idle timeout by setting `jbo.ampool.minavailablesize=0`, then when the application module is removed from its pool, it will put back the connection its holding onto.

- In contrast, if 50 different application module pools each have a single application module instance and `jbo.doconnectionpooling=true`, then the amount of JDBC connections in use will depend on how many of those application modules are *simultaneously* being used by different clients. If an application module instance is in the pool and is not currently being used by a user session, then with `jbo.doconnectionpooling=true` it will have released its JDBC connection back to the connection pool and while the application module instance is sitting there waiting for either another user to need it again, or to eventually be cleaned up by the application module pool monitor, it will not be "hanging on" to a JDBC connection.

For highest performance, Oracle recommends not disconnecting the application module instance from its database connection on each check in to the application module pool. Accordingly, the default setting of the `jbo.doconnectionpooling` configuration parameter is `false`. The pooling of application module instances is already an effective way to optimize resource usage, and there are runtime efficiencies that Oracle ADF can gain if you do not have to disconnect application module instances from their associated JDBC connection after each release to the pool. Effectively, by pooling the application modules which are related one-to-one with a JDBC connection, you are already achieving a pooling of database connections that is optimal for most web applications.

In contrast to Oracle's default recommendation, one situation in which it might be opportune to use database connection pooling is when you have a large number of application module pools all needing to use database connections from the same underlying application user at the database level. In this case, the many application module pools can perhaps economize on the total overall database sessions by sharing a single, underlying database connection pool of JDBC connections, albeit at a loss of efficiency of each one. This choice would be favored only if total overall database sessions is of maximum priority.

29.9 Database User State and Pooling Considerations

Sometimes you may need to invoke stored procedures to initialize database state related to the current user's session. The correct place to perform this initialization is in an overridden `prepareSession()` method of your application module.

29.9.1 How Often prepareSession() Fires When jbo.doconnectionpooling = false

The default setting for `jbo.doconnectionpooling` is `false`. This means the application module instance hangs onto its JDBC connection while it's in the application module pool. This is the most efficient setting because the application module can keep its JDBC prepared statements open across application module checkouts/checkins.The application module instance will trigger its `prepareSession()` method each time a new user session begins using it.

29.9.2 Setting Database User State When jbo.doconnectionpooling = true

If you set `jbo.doconnectionpooling` to `true`, then on each checkout of an application module from the pool, that application module pool will acquire a JDBC connection from the database connection pool and use it during the span of the current request. At the end of the request when the application module is released back to the application module pool, that application module pool releases the JDBC connection it was using back to the database connection pool.

It follows that with jbo.doconnectionpooling set to true the application module instance in the pool may have a completely different JDBC connection each time you check it out of the pool. In this situation, the prepareSession() method will fire each time the application module is checked out of the pool to give you a chance to reinitialize the database state.

29.9.3 Understanding How the SRDemo Application Sets Database State

The SRDemo application includes a simple example of setting database state on a per-user basis. It uses the following PL/SQL package to set and get a package-level variable that holds the name of the currently authenticated web application user.

Example 29–4 SRDemo CONTEXT_PKG PL/SQL Package

```
create or replace package context_pkg as
  procedure set_app_user_name(username varchar2);
  function app_user_name return varchar2;
end context_pkg;
```

The WHERE clause of the ServiceHistories view object in the demo references the context_pkg.app_user_name function to illustrate how your application queries might reference the per-user state.

> **Note:** In practice, you will typically create a database CONTEXT namespace, associate a PL/SQL procedure with it, and then use the SYS_CONTEXT() SQL function to reference values from the context. however the simple PL/SQL package above was enough to illustrate the mechanics involved in setting and referencing the user state without further complicating the installation of the SRDemo application, so the demo's authors chose this simpler approach to keep the demo more straightforward.

The SRApplicationModuleImpl framework extension class in the FrameworkExtensions project includes a callStoredProcedure() helper method similar to the ones in Section 25.5.2, "Invoking Stored Procedure with Only IN Arguments". The SRServiceImpl application module class extends this class and defines the setCurrentUserInPLSQLPackage() helper method shown in Example 29–5 that uses the callStoredProcedure() method to invoke context_pkg.set_app_user_name() stored procedure, passing the value of the currently authenticated user as a parameter value.

Example 29–5 Method to Call Context_Pkg.Set_App_User_Name Stored Procedure

```
// In SRServiceImpl.java
public void setCurrentUserInPLSQLPackage() {
  String user = getUserPrincipalName();
  callStoredProcedure("context_pkg.set_app_user_name(?)",new Object[]{user});
}
```

With this helper method in place, the SRServiceImpl class then overrides the prepareSession() method as shown in Example 29–6.

Example 29–6 Overridden afterConnect() and prepareSession() to Set Database State

```
// In SRServiceImpl.java
  protected void prepareSession(Session session) {
    super.prepareSession(session);
    getLoggedInUser().retrieveUserInfoForAuthenticatedUser();
    setUserIdIntoUserDataHashtable();
   setCurrentUserInPLSQLPackage();
  }
```

30

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 30.1, "Introduction to Security in Oracle ADF Web Applications"
- Section 30.2, "Specifying the JAZN Resource Provider"
- Section 30.3, "Configuring Authentication Within the web.xml File"
- Section 30.4, "Configuring the ADF Business Components Application to Use Container-Managed Security"
- Section 30.5, "Creating a Login Page"
- Section 30.6, "Creating a Logout Page"
- Section 30.7, "Implementing Authorization Using Oracle ADF Security"
- Section 30.8, "Implementing Authorization Programmatically"

30.1 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 30.2, "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 30.3, "Configuring Authentication Within the web.xml File"
- Section 30.4, "Configuring the ADF Business Components Application to Use Container-Managed Security"
- Section 30.5, "Creating a Login Page"
- Section 30.6, "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 30.7, "Implementing Authorization Using Oracle ADF Security"
- Section 30.8, "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.

30.2 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.

30.2.1 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"
/>
```

30.2.2 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.

30.3 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 30.3.1, "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.

30.3.1 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.

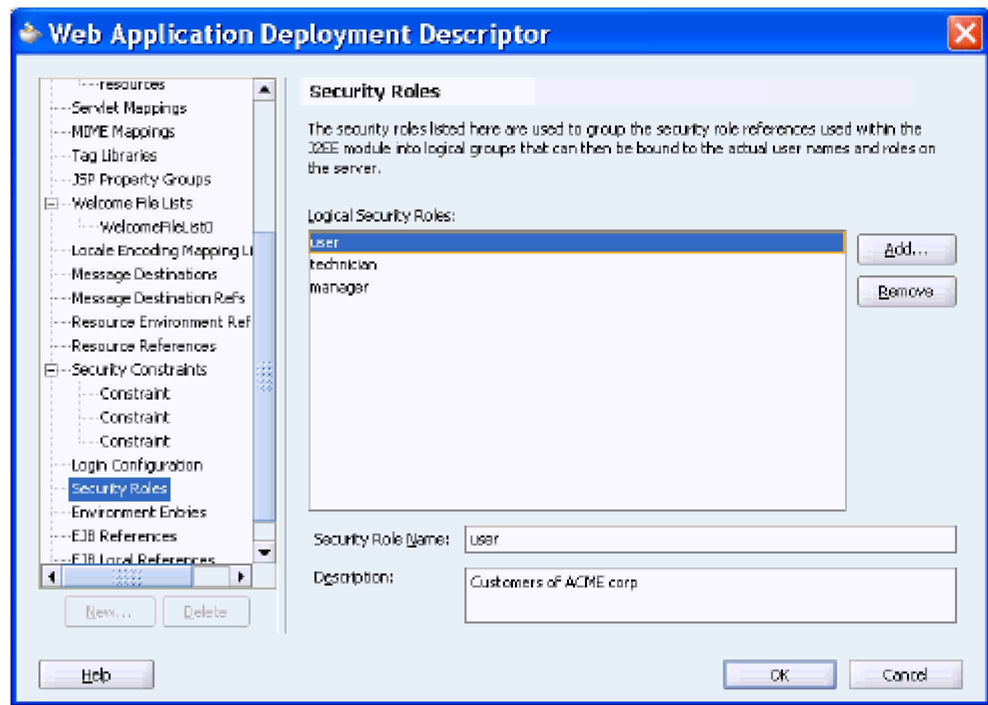
To specify security roles for J2EE container-managed security:

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 30–1 shows the web.xml editor with the Security Roles definition displayed. In the SRDemo application, three security roles are defined.

Figure 30–1 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 30.5, "Creating a Login Page".

To create a login configuration for J2EE container-managed security:

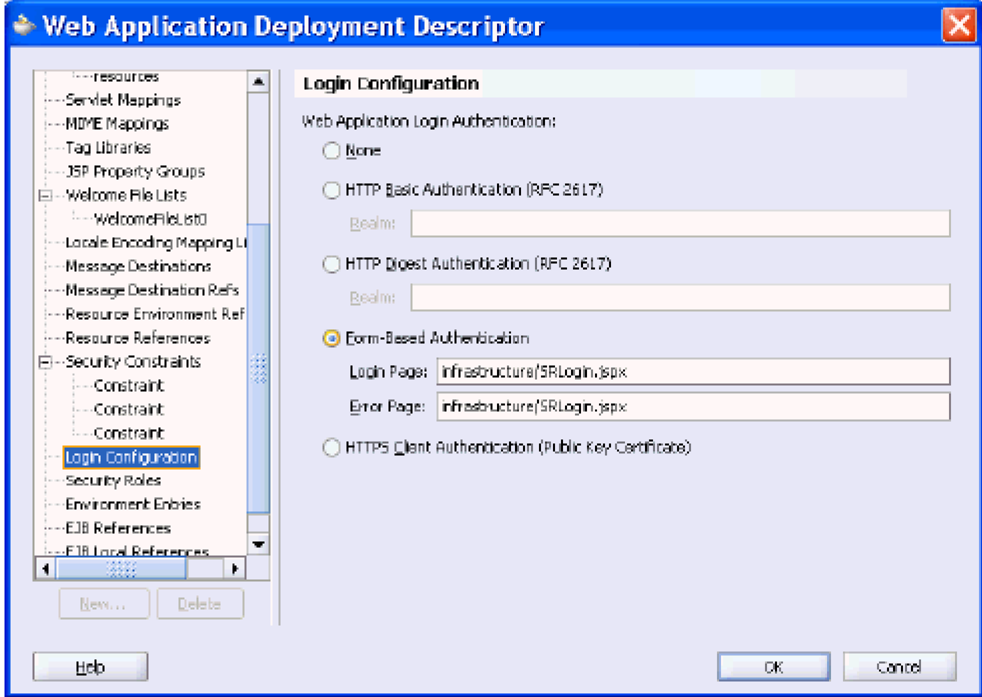
1. In the Navigator, expand your JSP project, right-click the **web.xml** file and choose **Properties**. The `web.xml` file resides in the WEB-INF folder of your project.

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 30.5.1, "Wiring the Login and Error Pages".

3. Save all changes and close the Web Application Deployment Descriptor editor.

Figure 30–2 shows the `web.xml` editor with the Login Configuration definition displayed.

Figure 30–2 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.

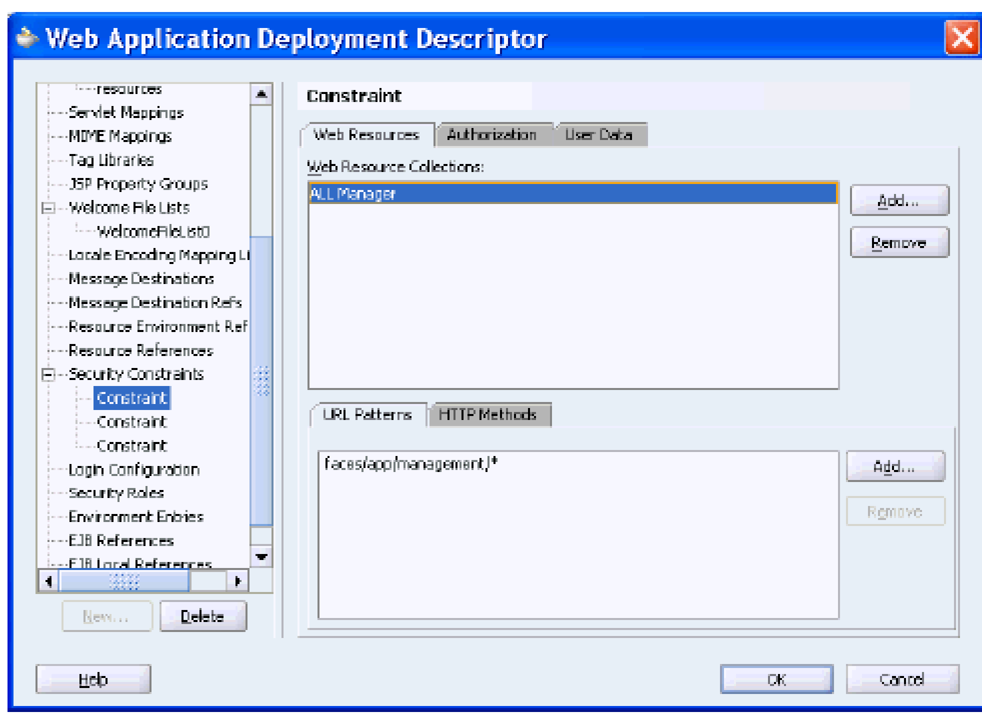
5. To specify authorized security roles, select the **Authorization** tab. Select the security roles that require authentication. The roles available are the roles you configured in step 2.

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 30–3 shows the `web.xml` editor with a Security Constraint definition displayed.

Figure 30–3 Web Application Deployment Descriptor Dialog, Security Constraints Panel



30.3.2 What Happens When You Use Security Constraints without Oracle ADF Security

Example 30–1 shows sample definitions similar to the ones that your `web.xml` file should contain when you have finished configuring J2EE container-managed security.

Example 30–1 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.

30.3.3 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`

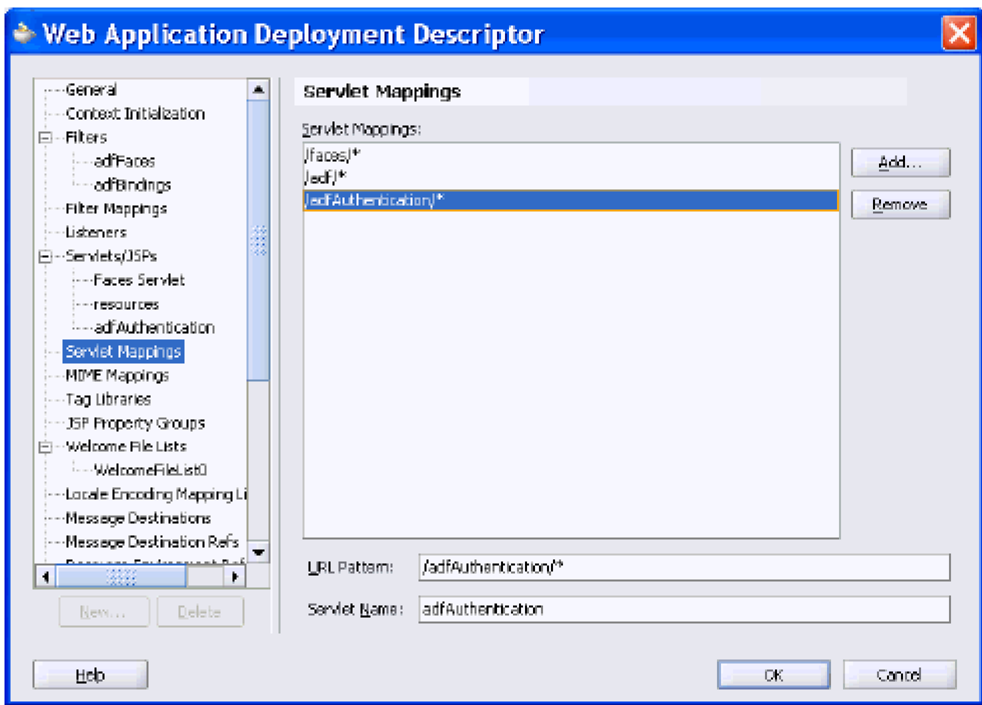
   To add an initialization parameter that contains the URL for the resulting page if authentication succeeds, select **Initialization Parameters** and click **Add**. If you do not enter a URL, the user will return to the current page.
4. To create a servlet mapping, select **Servlet Mapping** on the left 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 30–3 shows the `web.xml` editor with the Servlet Mapping definition displayed for the adfAuthentication servlet.

Figure 30–4 Web Application Deployment Descriptor Dialog, Servlet Mapping Panel



30.3.4 What Happens When You Use Security Constraints with Oracle ADF

Example 30–2 shows sample definitions similar to the ones that your `web.xml` file should contain.

Example 30–2 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.

30.4 Configuring the ADF Business Components Application to Use Container-Managed Security

When you want to work with security in an Oracle ADF Business Components application, the ADF Business Components application module must be enabled to recognize the authenticated user. This will permit the application to create the application module based on the presence of an authenticated user. If the user attempts to login and is not authenticated, no application module will be created for the session.

30.4.1 How to Configure Security in an Oracle ADF Business Components Application

To enable security in an Oracle ADF Business Components application, you must edit the `jbo.security.enforce` property of the application modules configuration (maintained in the `bc4j.xcfg` file). The `jbo.security.enforce` property set to `Must` specifies that an authenticated user must be logged in before the application module will be created. This is a requirement for any business components application that will work with container-managed security.

To configure security for Oracle ADF Business Components:

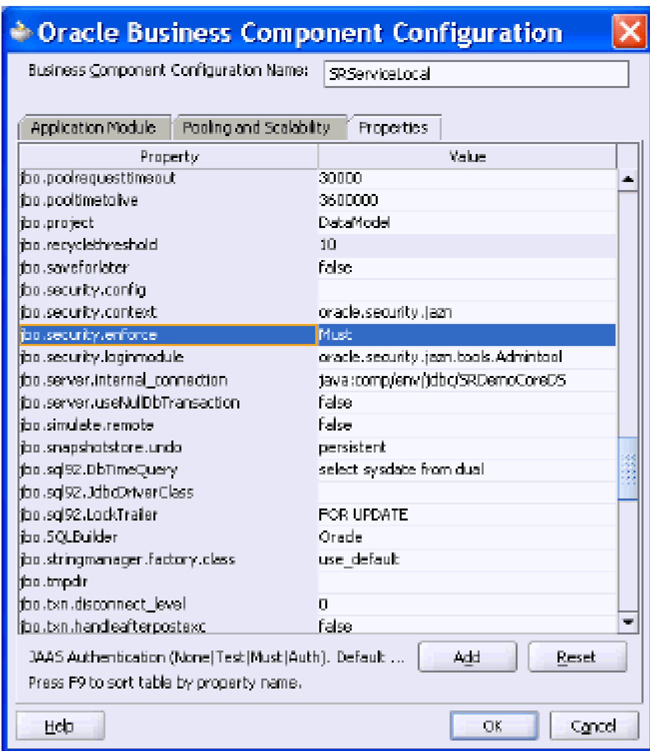
1. In the Application Navigator, expand the data model project and located the application module node.
2. Right-click the application module node and choose **Configurations**.
3. In the Configuration Manager, select the configuration for your application and click **Edit**.
4. In the Oracle Business Components Configuration dialog, select the **Properties** tab.

   The dialog display the full list of ADF Business Components configuration properties. The security properties begin with `jbo.security`.

5. Scroll to locate the `jbo.security.enforce` property and enter the value `Must`.
6. Click OK to close the dialogs and save the configuration changes.

Figure 30–6 shows the application module configuration SRServiceLocal and security property as they appear in the SRDemo application for ADF Business Components.

Figure 30–5 ADF BC Configuration Dialog, Properties Panel



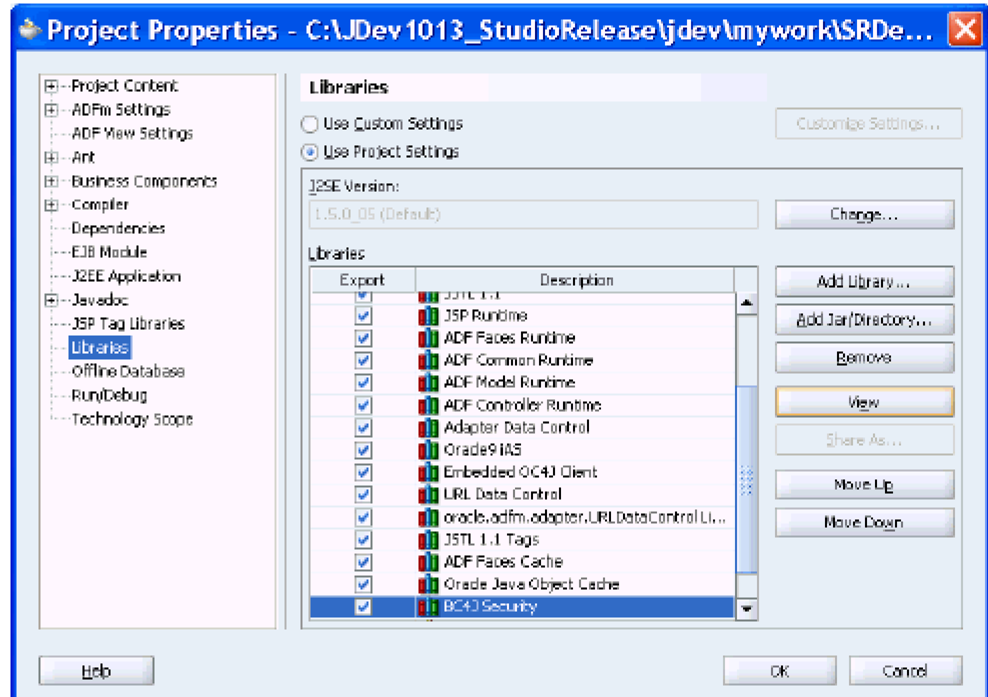
In order to use the JAZN realm in an ADF Business Components application, you must export the BC4J Security library to your user interface project. This will ensure that the `jazn.jar` is available to the user interface project at runtime. Without this library defined, an exception such as `NullPointerException in JboJAZNUserManager.isUserInRole()` will be returned when your application attempts to verify the logged in user.

To add required libraries for ADFBC Security to the user interface project:

1. In the Application Navigator, right-click the user interface project and choose **Project Properties**.
2. In the Project Properties dialog, select **Libraries** to view the list of available libraries.
3. Scroll to locate **BC4J Security** and make sure that **Export** is enabled.
4. Click **OK** to close the dialog and save the setting.

Figure 30–6 shows the BC4J Security library as it should appear for the user interface project.

Figure 30–6 Project Properties Dialog, Libraries Panel



30.4.2 What Happens When You Configure Security in an ADF Business Components Application

The `jbo.security.enforce` property in the ADF Business Components application module's configuration settings when set to `Must` requires the application module to obtain an authenticated user principal from the SecurityContext before the application module is created for the specified configuration.

30.4.3 What You May Need to Know About the ADF Business Components Security Property

The security mechanism provided by ADF Business Components can be combined with ADF Security when you want to want to configure security constraints against the adfAuthentication servlet. You enable ADF Security when you want to grant authorization permissions to ADF binding objects in the application. For details about the authorization features provided by ADF security, see Section 30.7, "Implementing Authorization Using Oracle ADF Security".

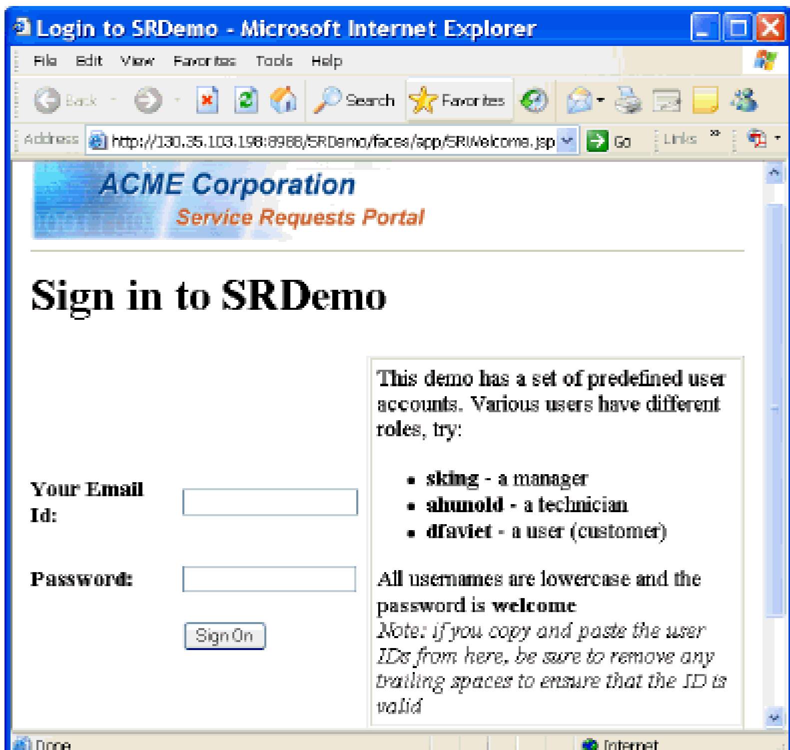
> **Tip:** Starting in JDeveloper 10.1.3.1 maintenance release, you need only set the ADF Security property `authorizationEnforce` to `true` and you will automatically enable security in an ADF Business Components application. In this case, the `jbo.security.enforce` property is not required.

When you enable ADF Security in an ADF Business Components application, the application module obtains the principal from the security context under the ADF context instead of the JAAS security context. For details about enabling ADF servlet authentication, see Section 30.3.3, "How to Enable Oracle ADF Authentication".

30.5 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 30–7 shows a sample login page from the SRDemo application.

Figure 30–7 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 30–3 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 30–3 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>
```

30.5.1 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 30–8 shows the `web.xml` editor with the Login Configuration definition displayed.

Figure 30–8 Web Application Deployment Descriptor Dialog, Login Configuration Panel

30.5.2 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 30–3 shows the conventions of that permit the HTML Form to invoke the authentication servlet. Specifically, the form must specify three pieces of information:

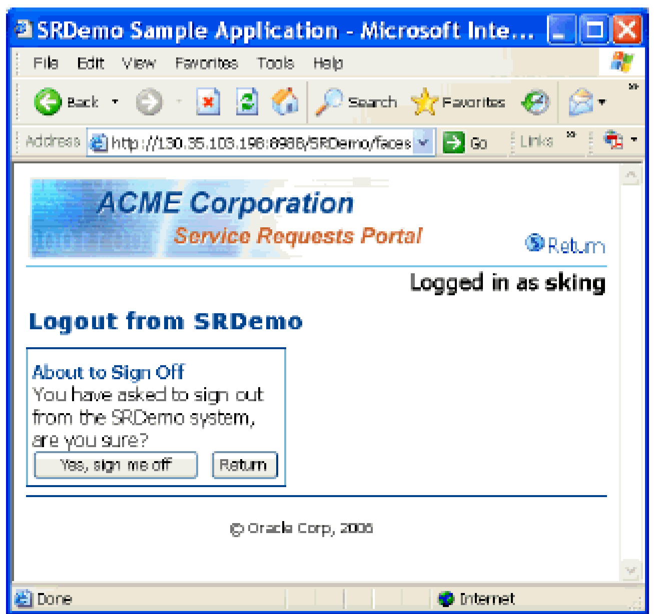
1. `<form action="j_security_check" method="post">` to invoke the security check method `j_security_check` on the container's login bean.
2. `<input type="text" name="j_username"/>` to assign the username value to the container's login bean attribute `j_username`.
3. `<input type="password" name="j_password"/>` 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.

30.6 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 30–9 shows the logout page from the SRDemo application.

Figure 30–9 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 30.6.1, "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 30–4 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 30–4 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>
```

30.6.1 Wiring the Logout Action

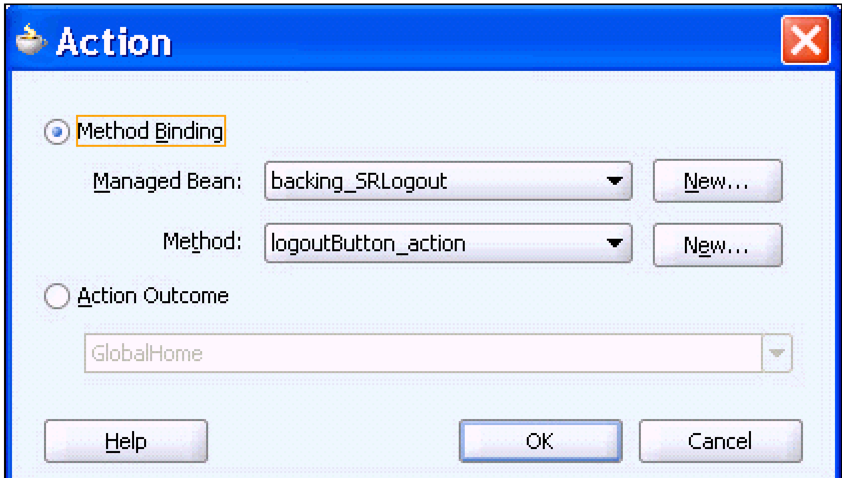
To handle the logout action, the JSPX document can use a managed bean with properties that correspond to the logout page's logout command button.

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 30–10 shows the Action property dialog with the managed bean backing_SRLogout and the method logoutButton_action() entered.

Figure 30–10 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 30–5 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 30–5 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 30–5 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;
}
```

30.6.2 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 30–5, invalidates the session and redirects to the home page.

30.7 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 30.8, "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 30–1 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 30–1 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 30.3.3, "How to Enable Oracle ADF Authentication".
- Configure your application to use Oracle ADF Security authorization. For details, see Section 30.7.1, "Configuring the Application to Use Oracle ADF Security Authorization".

30.7.1 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.

30.7.1.1 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.

30.7.1.2 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.

30.7.1.3 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
}
```

Note: Starting in JDeveloper 10.1.3.1 maintenance release, you need only set the ADF Security property `authorizationEnforce` to `true` and you will automatically enable security for ADF Business Components applications. In this case, the `jbo.security.enforce` property is not required. To understand how ADF Business Components enforces security, see Section 30.4, "Configuring the ADF Business Components Application to Use Container-Managed Security".

30.7.2 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 30–1 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.

30.7.3 Setting Authorization on ADF Iterator Bindings

You use the Authorization Editor to grant permissions for users on iterator bindings. See Table 30–1 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.

30.7.4 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 30–1 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.

30.7.5 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 30–6 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 `RowSetPermission` to create, read, and update the attributes of the bound collection `EmployeesView1`, and also an `AttributePermission` to read the `DepartmentID` attribute value.

Example 30–6 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.RowSetPermission</class>
      <name>EmployeesView1</name>
      <actions>create,read,update</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.

30.8 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) 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 30.3.1, "How to Enable J2EE Container-Managed Authentication".

30.8.1 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 17.2, "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']}"
                  actionListener="#{bindings.setCurrentRowWithKey.execute}"
                  action="Edit"
                  rendered="#{userInfo.manager}">
   <af:setActionListener from="#{row.rowKeyStr}"
                         to="#{processScope.rowKeyStr}"/>
   <af:setActionListener from="#{'GlobalHome'}"
                         to="#{userState.returnNavigationRule}"/>
</af:commandButton>
```

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>
```

30.8.1.1 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 30–7 shows the key methods that the SRDemo application implements:

Example 30–7 SRDemo Application UserInfo.java Sample

```
/**
 * Constructor
 */
public UserInfo() {

        FacesContext ctx = FacesContext.getCurrentInstance();
        ExternalContext ectx = ctx.getExternalContext();

        //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;
            }
        }
}

    /**
     * @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);
    }
}
```

30.8.1.2 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 30.8.1.1, "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.

Example 30–8 shows the portion of the `faces-config.xml` file that defines the managed bean `userInfo` to hold security information for the SRDemo application.

Example 30–8   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-bean>
```

31

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 31.1, "Introduction to the Simple CSV Data Control Adapter"
- Section 31.2, "Overview of Steps to Create a Data Control Adapter"
- Section 31.3, "Implement the Abstract Adapter Class"
- Section 31.4, "Implement the Data Control Definition Class"
- Section 31.5, "Implement the Data Control Class"
- Section 31.6, "Create any Necessary Supporting Classes"
- Section 31.7, "Create an XML File to Define Your Adapter"
- Section 31.8, "Build Your Adapter"
- Section 31.9, "Package and Deploy Your Adapter to JDeveloper"
- Section 31.10, "Location of Javadoc Information"

31.1 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.

31.2 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 31.3, "Implement the Abstract Adapter Class"
   - Section 31.4, "Implement the Data Control Definition Class"
   - Section 31.5, "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 31.6, "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 31.7, "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 31.9, "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 31.7, "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.

31.3 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 31.7, "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 31.3.4, "Implementing the initialize Method"
- Section 31.3.5, "Implementing the invokeUI Method"
- Section 31.3.6, "Implementing the getDefinition Method"

31.3.1 Location of JAR Files

The abstract class `oracle.adf.model.adapter.AbstractAdapter` is located in the `JDEV_HOME/bc4j/lib/adfm.jar` file.

31.3.2 Abstract Adapter Class Outline

Example 31–1 shows an outline of a class that implements the `AbstractAdapter` class.

Example 31–1 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 31.3.4, "Implementing the initialize Method".
   }

   public boolean invokeUI()
   {
      // you need to implement this method.
      // see Section 31.3.5, "Implementing the invokeUI Method".
   }

   public AbstractDefinition getDefinition()
   {
      // you need to implement this method.
      // see Section 31.3.6, "Implementing the getDefinition Method".
   }
}
```

31.3.3 Complete Source for the SampleDCAdapter Class

Example 31–2 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 31–2 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;
}

}
```

31.3.4 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 31–3 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 31–4 initialize Method

```
public void initialize(Object sourceObj, DTContext ctx)
{
   mSrc = sourceObj;
   mJdevCtx = (Context) ctx.get(DTContext.JDEV_CONTEXT);
}
```

31.3.5 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 31–5 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 31.4, "Implement the Data Control Definition Class".

Example 31–6 shows the invokeUI method:

Example 31–6 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;
}
```

31.3.6 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 31–7 getDefinition Signature

```
public abstract AbstractDefinition getDefinition();
```

The AbstractDefinition class is the data control definition class that you created. See Section 31.4, "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 31–8 getDefinition

```
public AbstractDefinition getDefinition()
{
   return mDefinition;
}
```

31.4 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 31.4.4, "Creating a Default Constructor".
- collecting metadata from the user about the data source. See Section 31.4.5, "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 31.4.6, "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 31.4.7, "Creating an Instance of the Data Control".
- enabling the framework to load the metadata from the DCX file. See Section 31.4.8, "Setting the Metadata for Runtime".
- setting a name for your data control. See Section 31.4.9, "Setting the Name for the Data Control".

31.4.1 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.

31.4.2 Data Control Definition Class Outline

Example 31–9 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 31–9 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 31.4.4, "Creating a Default Constructor".
   }

   public Node getMetadata()
   {
      // you need to implement this method.
      // see Section 31.4.5, "Collecting Metadata from the User".
   }
```

```
    public StructureDefinition getStructure()
    {
       // you need to implement this method.
       // see Section 31.4.6, "Defining the Structure of the Data Control".
    }

    public DataControl createDataControl()
    {
       // you need to implement this method.
       // see Section 31.4.7, "Creating an Instance of the Data Control".
    }

    public void loadFromMetadata(Node node, Map params)
    {
       // you need to implement this method.
       // see Section 31.4.8, "Setting the Metadata for Runtime".
    }

    public String getDCName()
    {
       // you need to implement this method.
       // see Section 31.4.9, "Setting the Name for the Data Control".
    }
}
```

31.4.3 Complete Source for the SampleDCDef Class

Example 31–10 shows the complete source for the `SampleDCDef` class:

Example 31–10 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";
   }
}
```

31.4.4 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 31–11 SampleDCDef Default Constructor

```
public SampleDCDef()
{
}
```

The default constructor is used only during runtime. It is not used during design time.

31.4.5 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 31–12 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 31–13 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 31–14 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.

31.4.6 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 31–15 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 31–16 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;
}
```

31.4.7 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 31–17 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 31.5, "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 31–18 createDataControl Method

```
public DataControl createDataControl()
{
   SampleDataControl dcDataControl = new SampleDataControl(mCSVUrl);
   return dcDataControl;
}
```

The `SampleDataControl` class is described in more detail in Section 31.5, "Implement the Data Control Class".

31.4.8 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 31.4.5, "Collecting Metadata from the User".

The `loadFromMetadata` method has the following signature:

Example 31–19 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 31–20 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);
   }
}
```

31.4.9 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 31–21 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 31–22 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.

31.5 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 31.4.7, "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 31–1 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.)

31.5.1 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.

31.5.2 Data Control Class Outline

The following class outline for a data control class shows the methods you have to implement:

Example 31–23 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 31.5.4, "Implementing the invokeOperation Method".
   }

   public String getName()
   {
      // you need to implement this method.
      // see Section 31.5.5, "Implementing the getName Method".
   }

   public void release(int flags)
   {
      // you need to implement this method.
      // see Section 31.5.6, "Implementing the release Method".
   }

   public Object getDataProvider()
   {
      // you need to implement this method.
      // see Section 31.5.7, "Implementing the getDataProvider Method".
   }
}
```

31.5.3 Complete Source for the SampleDataControl Class

Example 31–24 shows the complete source for the `SampleDataControl` class.

Example 31–24 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;
    }

}
```

31.5.4 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 31–25   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 31–26 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.

31.5.4.1 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 31–27 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.

31.5.4.2 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.

31.5.5 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 31–28 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 31–29 getName Method

```
public String getName()
{
   return mName;
}
```

31.5.6 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 31–30 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.

31.5.7 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 31–31 getDataProvider Signature

```
public Object getDataProvider();
```

In the simple CSV data control adapter, this method just returns null.

31.6 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 31.11, "Contents of Supporting Files" for complete source listing for these classes.

31.7 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 31–32 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 31–33   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 31.11.1, "sampleDC.xsd".

31.8 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`

31.9 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 31–34 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 31–35 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 31–36 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.

31.10 Location of Javadoc Information

The JDeveloper online help provides reference information for the classes described in this chapter in Javadoc format.

Table 31–2 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

31.11 Contents of Supporting Files

This section shows the contents of the following files:

- Section 31.11.1, "sampleDC.xsd"
- Section 31.11.2, "CSVHandler Class"
- Section 31.11.3, "CSVParser"

31.11.1 sampleDC.xsd

Example 31–37 shows the contents of the `sampleDC.xsd` file.

Example 31–37 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>
```

31.11.2 CSVHandler Class

Example 31–38 shows the contents of the `CSVHandler` class.

Example 31–38 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 //////////////////////////////////

  /**
   * Creates 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;
   }

 }
```

31.11.3 CSVParser

Example 31–39 shows the contents of the `CSVParser` class.

Example 31–39 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;
      }
    }

  }
```

32

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 32.1, "Using CVS with an ADF Project"
- Section 32.2, "General Advice for Using CVS with JDeveloper"

32.1 Using CVS with an ADF Project

This section contains advice specifically for using CVS with an ADF project, for example the SRDemo application.

32.1.1 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.

32.1.2 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.

32.1.3 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.

32.1.4 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.

32.1.5 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.

32.1.5.1 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**).

32.1.6 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.

32.1.7 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.

32.2 General Advice for Using CVS with JDeveloper

This section contains advice for using CVS with JDeveloper generally.

32.2.1 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.

32.2.2 Developer-Level Activities

This section contains advice for developers working with files under CVS control.

32.2.2.1 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.

32.2.2.2 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.

32.2.2.3 Advice for Merge Conflicts in ADF Business Components Projects

You can remove one cause of merge conflicts in an multi-developer environment by disabling the use of ADF Business Components package XML files. You do this by deselecting the option **Copy Package XML Files to Class Path** on the **Business Components: General** page of the Preferences dialog. See Section 4.4.7.2, "Recommendation for Disabling Use of Package XML File"

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

After resolving merge conflicts in any ADF Business Components XML component descriptor files, the project file (`.jpr`) for an ADF Business Components project, or the corresponding business components project file (`.jpx`), Oracle recommends closing and reopening the project to ensure that you're working with latest version of the component definitions. To do this, select the project in the Application Navigator,

choose **File** > **Close** from the JDeveloper main menu, then expand the project again in the Application Navigator.

33

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 33.1, "What are Web Services"
- Section 33.2, "Creating Web Service Data Controls"
- Section 33.3, "Securing Web Service Data Controls"
- Section 33.4, "Publishing Application Modules as Web Services"
- Section 33.5, "Calling a Web Service from an Application Module"

33.1 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 business components 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.

33.1.1 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.

33.1.2 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.

33.1.3 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.

33.1.4 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.

33.2 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.

33.2.1 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 select the Create Data Control from the context menu.

33.3 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.

33.3.1 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

33.3.2 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.

33.3.2.1 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 33–1 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 33–2 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.

33.3.2.2 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 33–3 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 33–4 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.

Example 33–5 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.

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.

33.3.2.3 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 33–6 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 the example, serverencpublic.cer contains the public key certificate of the server's encryption key. To import this certificate in the client's keystore, run the following:

Example 33–7 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, as shown in the following examples:

Example 33–8 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 33–9 Command to Import the Public Key Certificate

```
keytool -import -file clientencpublic.cer -alias clientencpublic -keystore
server.jks
```

The Keytool utility will prompt you for the keystore password.

The server and client keystores are now ready to be used to configure security for the web service data control.

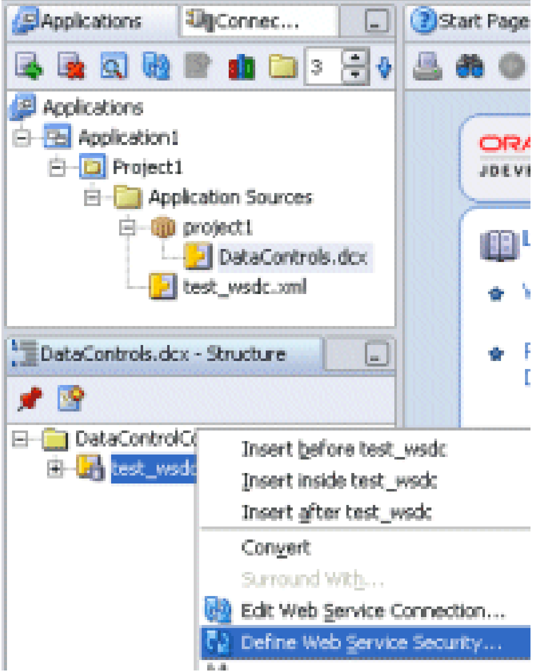
33.3.3 Defining Web Service Data Control Security

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.

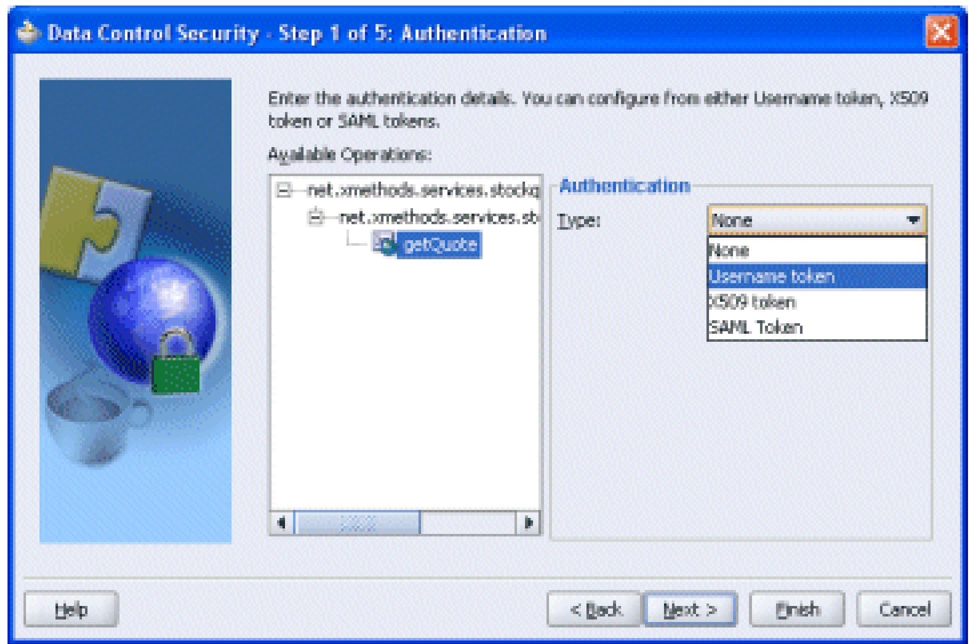
Figure 33–1 Invoking the Data Control Security Wizard



33.3.3.1 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 33–2 Select the Type of Authentication



33.3.3.1.1 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>
```

33.3.3.1.2 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.

33.3.3.1.3 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.

33.3.3.1.4 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.

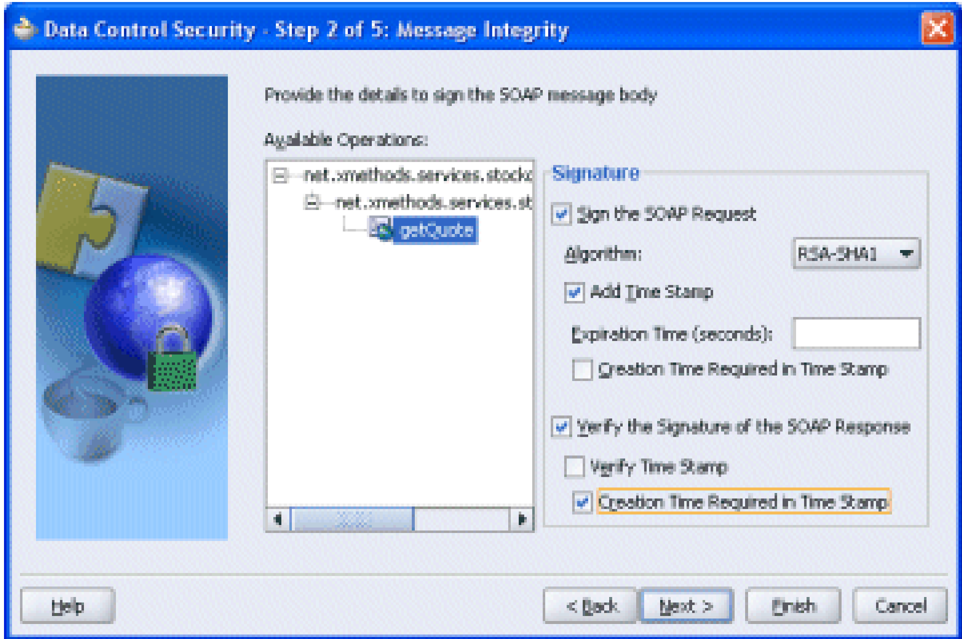
To use SAML authentication:

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.

33.3.3.2 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 33–3 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:

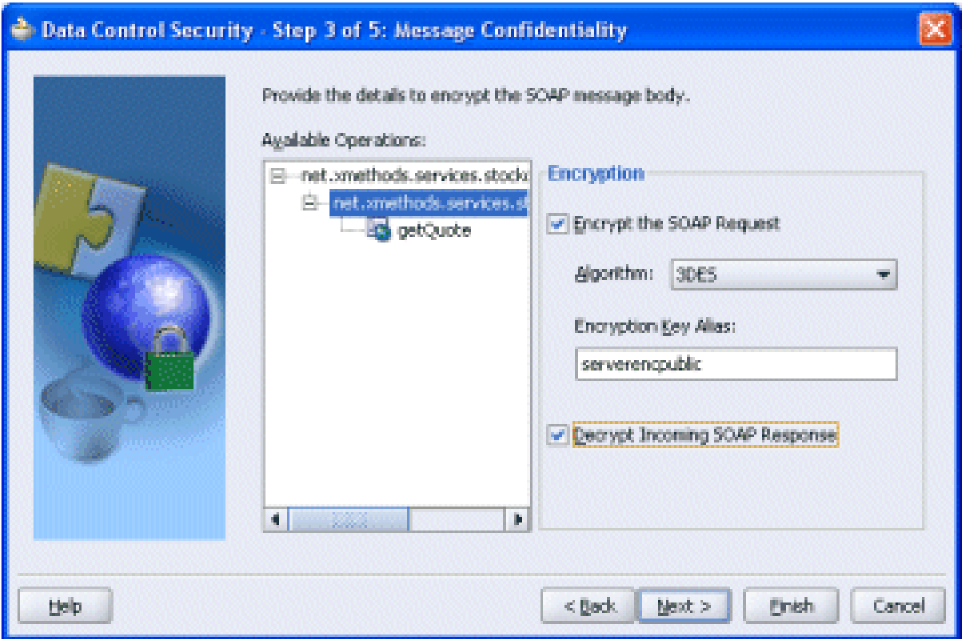
1. In the Message Integrity page of the wizard, select the appropriate options.
2. In the Keystore page of the wizard, and specify the location of the keystore file, and enter the signature key alias and password.

33.3.3.3 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 33–4 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.

33.3.3.4 How to Use a Key Store

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 33–5 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.

33.4 Publishing Application Modules as Web Services

Oracle ADF Business Components application modules offer built-in support for web services. Any custom method that you add to the client interface of your application module appears on its web service interface after you enable the optional web service deployment option.

33.4.1 How to Enable the J2EE Web Service Option for an Application Module

To enable your application module as a web service:

1. Enable a custom Java class for your application module and add to it one or more custom methods that you want to appear on the web service interface.
2. Open the Application Module editor and on the **Client Interface** tab, select one or more custom methods to appear on the client interface.
3. With the Application Module editor still open, on the **Remote** panel, select **Remoteable Application Module**, select **J2EE Web Service** in the **Available** list and shuttle it to the selected list.
4. Then click **OK** to save your changes.

33.4.2 What Happens When You Enable the J2EE Web Service Option

Once this J2EE Web Service remoteable option of your application module is enabled, every time you dismiss the Application Module editor, JDeveloper synchronizes your web service classes with any changes you made to the application module's client interface.

JDeveloper creates the generated web service class in the `server.webservice` subpackage of the package where your application module resides. For example, if you application module were named `oracle.srdemo.model.SRService`, its generated web service class would be named `SRServiceServer` in the `oracle.srdemo.model.server.webservice` package.

For each method named `someMethod` in your application module's client interface, JDeveloper adds a delegator method in the generated web service class that looks like what you see in Example 33–10. The code performs the following basic steps:

1. Acquires an application module instance from the pool.
2. Delegates the web service method call to the application module instance.
3. Returns the result if the method is non-void.
4. Releases the application module instance to the pool without removing it.

Since the web service code passes `false` to the `releaseRootApplicationModule` method of the `Configuration` object, the application module instance is not removed and remains in the pool ready to handle the next incoming stateless web service method invocation. Then the application pool grows and shrinks with load in the same way the web application will. The only difference is that here the pool is being used in a totally stateless way. For more information on tuning the application module pool, see Chapter 29, "Understanding Application Module Pooling".

Example 33–10 Generated Code for Each Method in Application Module Client Interface

```
// In YourAppModuleServer.java generated web service class
  :
  public void someMethod(int p1, String p2) {
    AppModuleImpl _am = null;
    try {
    // 1. Acquire an application module instance from the pool
      _am = (AppModuleImpl)Configuration.createRootApplicationModule(
              "com.yourcompany.yourapp.YourAppModule",
               "YourAppModuleLocal");
   // 2. Delegate a call to its someMethod() method, passing arguments
      _am.someMethod(p1, p2);
   // 3. If return of method is non-void, return the result here
    } finally {
      if (_am != null) {
        // 4. Release application module to the pool, without removing it
        Configuration.releaseRootApplicationModule(_am, false);
      }
    }
  :
  }
```

33.4.3 What You May Need to Know About Deploying an Application Module as a Web Service

JDeveloper creates a web service deployment profile as part of enabling the web service remote option on your application module. To deploy the service, just select **Deploy** from the context menu of this deployment profile.

33.4.4 What You May Need to Know About Data Types Supported for Web Service Methods

Your custom method parameters and return types can use any datatype that implements the `java.io.Serializable` interface. This *excludes* the use of any generic ADF Business Components interfaces from the `oracle.jbo` package like `Row` or `ViewObject`. While these types are allowed on the custom interface of an application module, in general, they are not supported for web services. To return collections of data through a web service interface, you will need to create custom data transfer objects that implement the `Serializable` interface and populate them from the results of your view object query results before returning them in an array.

For example, if you needed to return a single row of the `ServiceRequests` view object instance in the SRDemo application's SRService application module, you would need to create a new `ServiceRequest` bean with appropriate bean properties and corresponding getter and setter methods. This class would need to implement the Serializable interface. Since this is just a marker interface, there is no additional implementation required other than adding the `implements Serializable` keywords to the class definition. Once you have created this bean, you can populate its properties inside the custom method implementation, and then return it as the return value of a custom service method like:

```
public ServiceRequest findServiceRequest(int svrid)
```

If you need to return a collection of serializable objects, you can use any type supported by the web service infrastructure. For example, you could return an array of one or more service requests using a custom method signature like:

```
public ServiceRequest[] findServiceRequests(String status, String technician)
```

33.5 Calling a Web Service from an Application Module

In a service-oriented architecture, sometimes your application module will need to take advantage of functionality offered by a web service. JDeveloper's built-in web services wizards make this an easy task. After creating a web service proxy class using the wizard, calling the service is as simple as calling a method in a local Java object.

33.5.1 Understanding the Role of the Web Services Description Language Document

A web service can be implemented in any programming language and can reside on any server on the network. Each web service identifies the methods in its API by describing them in a standard, language-neutral XML format. This XML document, whose syntax adheres to the Web Services Description Language (WSDL) described in Section 33.1.2, "WSDL", enables tools like JDeveloper to automatically understand the names of the web service's methods as well as the data types of the parameters they might expect and their eventual return value.

In order to work with any web service, you will need to know the URL that identifies its WSDL document. This URL could be a file-based URL like:

```
file:///D:/temp/SomeService.wsdl
```

if you have received the WSDL document as an email attachment, for example, and saved it to your local hard drive. Alternatively, the URL could be an HTTP-based URL like:

```
http://somerserver.somecompany.com/SomeService/SomeService.wsdl
```

Some web services make their WSDL document available by using a special parameter to modify the actual service URL itself. So, for example, a web service that expects to receive requests at the HTTP address of:

```
http://somerserver.somecompany.com/SomeService
```

might publish the corresponding WSDL document using the same URL with an additional parameter on the end like this:

```
http://somerserver.somecompany.com/SomeService?WSDL
```

Since there is no established standard, you will just need to know what the correct URL to the WSDL document is. With that URL in hand, you can then create a web service proxy class to call the service.

33.5.2 Understanding the Role of the Web Service Proxy Class

To call a web service from an application module, you create a web service proxy class for the service you want to invoke. A web service proxy is a generated Java class that represents the web service inside your application. It encapsulates the service URL of the web service and handles the lower-level details of making the call.

The web service proxy class presents a set of Java methods that correspond to the web service's public API. By using the web service proxy class, you can call any method in the web service in the same way as you work with the methods of any other local Java class.

33.5.3 How to Call a Web Service from an Application Module

To call a web service from an application module, you perform three steps:

1. Create a web service proxy class for the web service.
2. Create an instance of the web service proxy class in your application module.
3. Invoke one or more methods on the web service proxy object.

33.5.3.1 Creating a Web Service Proxy Class for a Web Service

To create a web service proxy class for a web service you need to call, use the Create Web Service Proxy wizard. You'll find it in the **Business Tier > Web Services** category of the New Gallery. When the wizard appears, follow these steps:

1. In step 1on the **Web Service Description** page, enter the URL for the WSDL document that describes the service, then press **[Tab]**.

   For example, if the service is created to comply with the latest JAXRPC standard and deployed to an Oracle Application Server, the WSDL URL might look like this:

   ```
   http://someserver:8888/StockQuoteService/StockQuoteServiceSoapHttpPort?WSDL
   ```

2. If the wizard displays the **Next >** button enabled, then JDeveloper has recognized and validated the WSDL document. You can click it and proceed to step 3. If the button does not enable, click **Why Not?** to understand what problem JDeveloper encountered when trying to read the WSDL document. If necessary, fix the problem after verifying the URL and try step 1 again.
3. In step 5 on the **Default Mapping Options** page, choose a Java package name for the generated web service proxy class, then click **Finish**.

33.5.3.2 Understanding the Generated Web Service Proxy

JDeveloper generates the web service proxy class in the package you've indicated with a name that reflects the name of the web service port it discovered in the WSDL document. The web service port name might be a nice, human-readable name like `StockQuoteService`, or could be a less-friendly name like `StockQuoteServiceSoapHttpPort`. This port name is decided by the developer that published the web service you are using. Assuming that the port name of the service is `StockQuoteServiceSoapHttpPort`, JDeveloper will generate a web proxy class named `StockQuoteServiceSoapHttpPortClient`.

The web service proxy displays in the Application Navigator as a single, logical node called *`WebServiceName`*`Proxy`. For example, the node for the StockQuoteService web service above would appear in the navigator with the name `StockQuoteServiceProxy`. As part of generating the proxy class, in addition to the main web service proxy class that you will use to invoke the server, JDeveloper generates a number of auxiliary classes and interfaces. You can see these files in the Structure window by selecting the web service proxy node in the Application Navigator. The generated files are used as part of the lower-level implementation of invoking the web service.

The only auxiliary generated classes you will need to reference are those created to hold structured web service parameters or return types. For example, imagine that the `StockQuoteService` web service has a `quoteForSymbol()` method that accepts one String parameter and returns a floating-point value indicating the current price of the stock. If the designer of the web service chose to return a simple floating-point number, then the web service proxy class would have a corresponding method like this:

```
public float quoteForSymbol(String symbol)
```

If instead, the designer of the web service thought it useful to return *multiple* pieces of information as the result, then the service's WSDL file will include a named structure definition describing the multiple elements it contains. For example, assume the service returns both the symbol name *and* the current price as a result. To contain these two data elements, the WSDL file might define a structure named `QuoteInfo` with an element named `symbol` of String type and an element named `price` of floating-point type. In this situation, when JDeveloper generates the web service proxy class, the Java method signature will look like this instead:

```
public QuoteInfo quoteForSymbol(String symbol)
```

The `QuoteInfo` return type references one of the auxiliary classes that comprises the web service proxy implementation. It is a simple bean whose properties reflect the names and types of the structure defined in the WSDL document. In a similar way, if the web service accepts parameters whose values are structures or *arrays* of structures, then you will work with these structures in your Java code using the corresponding generated beans.

33.5.3.3 Calling a Web Service Method Using the Web Service Proxy Class

Once you've generated the web service proxy class, you can use it inside a custom method of your application module as shown in Example 33–11.

Example 33–11 Calling a Web Service Method Using the Web Service Proxy Class

```
// In YourModuleImpl.java
public void performSomeApplicationTask(String symbol) throws Exception {
  // application-specific code here
   :
  // Create an instance of the web service proxy class
  StockQuoteServiceSoapHttpPortClient svc =
            new StockQuoteServiceSoapHttpPortClient();
  // Call a method on the web service proxy class and get the result
  QuoteInfo quote = svc.quoteForSymbol(symbol);
  float currentPrice = quote.getPrice();
  // more application-specific code here
}
```

33.5.4 What Happens When You Call a Web Service from an Application Module

When you invoke a web service from an application module, the web service proxy class handles the lower-level details of using the XML-based web services protocol described in Section 33.1.1, "SOAP". In particular, it does the following:

- Creates an XML document to represent the method invocation
- Packages any method arguments in XML
- Sends the XML document to the service URL using an HTTP POST request
- Unpackages the XML-encoded response from the web service.

If the method you invoke has a return value, your code receives it as an appropriately typed object to work with in your application module code.

33.5.5 What You May Need to Know

33.5.5.1 Use a Try/Catch Block to Handle Web Service Exceptions

By using the generated web service proxy class, invoking a remote web service becomes as easy as calling a method in a local Java class. The only *real* distinction to be aware of is that the web service method call could fail if there is a problem with the HTTP request involved. The method calls that you perform against a web service proxy should anticipate the possibility that the request might fail by wrapping the call with an appropriate `try...catch` block. Example 33–12 improves on the simpler example shown above by implementing the best practice of catching the web service exception. In this case it simply rethrows the error as a `JboException`, but you could implement more appropriate error handling in your own application.

Example 33–12 Wrapping Web Service Method Calls with a Try/Catch Block

```
// In YourModuleImpl.java
public void performSomeApplicationTask(String symbol) {
  // application-specific code here
  // :
  QuoteInfo quote = null;
  try {
    // Create an instance of the web service proxy class
    StockQuoteServiceSoapHttpPortClient svc =
               new StockQuoteServiceSoapHttpPortClient();
    // Call a method on the web service proxy class and get the result
    quote = svc.quoteForSymbol(symbol);
  }
  catch (Exception ex) {
    throw new JboException(ex);
  }
  float currentPrice = quote.getPrice();
  // more application-specific code here
}
```

33.5.5.2 Web Services are Do Not Share a Transaction with the Application Module

You will use some web services to access reference information. However, other services you call may *modify* data. This data modification might be in your own company's database if the service was written by a member of your own team or another team in your company. If the web service is outside your firewall, of course the database being modified will be managed by another company. In either of these situations, it is important to understand that any data modifications performed by a web service you invoke will occur in their own distinct transaction that is unrelated to the application module's current unit of work. For example, if you have invoked a web service that modifies data and then you later call `rollback()` to cancel the pending changes in the application module's current unit of work, this has no effect on the changes performed by the web service you called in the process. You may need to invoke a corresponding web service method to perform a compensating change to account for your rollback of the application module's transaction.

33.5.5.3 Setting Browser Proxy Information

If the web service you need to call resides outside your corporate firewall, you need to ensure that you have set the appropriate Java system properties to configure the use of an HTTP proxy server. The Java system properties to configure are:

- `http.proxyHost`

  Set this to the name of the proxy server.

- `http.proxyPort`

  Set this to the HTTP port number of the proxy server (often 80).

- `http.nonProxyHosts`

  Optionally set this to a vertical-bar-separated list of servers *not* requiring the user of a proxy server (e.g. "`localhost|127.0.0.1|*.`*`yourcompany`*`.com`").

Within JDeveloper, you can configure an HTTP proxy server on the **Web Browser and Proxy** page of the IDE Preferences dialog. When you run your application, JDeveloper includes appropriate `-D` command-line options to set the above three system properties based on the settings you've indicated in this dialog.

34

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 34.1, "Introduction to Deploying ADF Applications"
- Section 34.2, "Deployment Steps"
- Section 34.3, "Deployment Techniques"
- Section 34.4, "Deploying Applications Using Ant"
- Section 34.5, "Deploying the SRDemo Application"
- Section 34.6, "Deploying to Oracle Application Server"
- Section 34.7, "Deploying to JBoss"
- Section 34.8, "Deploying to WebLogic"
- Section 34.9, "Deploying to WebSphere"
- Section 34.10, "Deploying to Tomcat"
- Section 34.11, "Deploying to Application Servers That Support JDK 1.4"
- Section 34.12, "Installing ADF Runtime Library on Third-Party Application Servers"
- Section 34.13, "Verifying Deployment and Troubleshooting"

34.1 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 34.12, "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.

34.2 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 34.12, "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 34.9, "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

   You can also select Business Components Archive, if you are using ADF Business Components.

   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`, 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 11.4.2.1, "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 11.4.2.2, "More About the faces-config.xml File" and Section 11.4.2.3, "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, which are required if you are deploying ADF Business Components, `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 12.2, "Using the Data Control Palette", check that you have the `DataBindings.cpx` file. For information about the file, see Section 12.3, "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 34.8.3, "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 34-3).

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 an URL of the following type:

- For Oracle AS: 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.

34.3 Deployment Techniques

Table 34–1 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 34–1 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.

34.4 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`.

34.5 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.

34.6 Deploying to Oracle Application Server

This section describes deployment details specific to Oracle Application Server:

- Section 34.6.1, "Oracle Application Server Versions Supported"
- Section 34.6.2, "Oracle Application Server Release 2 (10.1.2) Deployment Notes"
- Section 34.6.3, "Oracle Application Server Deployment Methods"
- Section 34.6.4, "Oracle Application Server Deployment to Test Environments ("Automatic Deployment")"
- Section 34.6.5, "Oracle Application Server Deployment to Clustered Topologies"

34.6.1 Oracle Application Server Versions Supported

Table 34–2 shows the supported versions of Oracle Application Server:

Table 34–2 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

34.6.2 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 34.11, "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 34.12.1, "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 , 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 34–3 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 34–3 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**.

34.6.3 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.

34.6.4 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*.

34.6.5 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*.

34.7 Deploying to JBoss

This section describes deployment details that are specific to JBoss.

- Section 34.7.1, "JBoss Versions Supported"
- Section 34.7.2, "JBoss Deployment Notes"
- Section 34.7.3, "JBoss Deployment Methods"

34.7.1 JBoss Versions Supported

Table 34–4 shows the supported versions of JBoss:

Table 34–4 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

34.7.2 JBoss Deployment Notes

- Before deploying applications that use ADF to JBoss, you need to install the ADF runtime libraries on JBoss. See Section 34.12, "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.

- 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.
- 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.

34.7.3 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.

34.8 Deploying to WebLogic

This section describes deployment details that are specific to WebLogic.

- Section 34.8.1, "WebLogic Versions Supported"
- Section 34.8.2, "WebLogic Versions 8.1 and 9.0 Deployment Notes"
- Section 34.8.3, "WebLogic 8.1 Deployment Notes"
- Section 34.8.5, "WebLogic Deployment Methods"

34.8.1 WebLogic Versions Supported

Table 34–5 shows the supported versions of WebLogic:

Table 34–5 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

34.8.2 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 34.12, "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.

34.8.3 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 34.11, "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.

34.8.4 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.

34.8.5 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/`).

34.9 Deploying to WebSphere

This section describes deployment details that are specific to WebSphere.

- Section 34.9.1, "WebSphere Versions Supported"
- Section 34.9.2, "WebSphere Deployment Notes"
- Section 34.9.3, "WebSphere Deployment Methods"

34.9.1 WebSphere Versions Supported

Table 34–6 shows the supported versions of WebSphere:

Table 34–6 Support Matrix for WebSphere

WebSphere version	JDK version	J2EE version
6.0.1	1.4.2	1.4

34.9.2 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 34.11, "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 34.12.2, "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 34-22.

34.9.3 WebSphere Deployment Methods

You can deploy using the WebSphere console (for example: `http://`*`<websphere_host:port>`*`/ibm/console/`).

34.10 Deploying to Tomcat

This section describes deployment details that are specific to Tomcat.

34.10.1 Tomcat Versions Supported

Table 34–7 shows the supported versions of Tomcat:

Table 34–7 Support Matrix for Tomcat

Tomcat version	JDK version	J2EE version
5.5.9	1.5	1.4

34.10.2 Tomcat Deployment Notes

- Before deploying applications that use ADF to Tomcat, you need to install the ADF runtime libraries on Tomcat. See Section 34.12, "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.

34.11 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.

34.11.1 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.

34.12 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 34.12.1, "Installing the ADF Runtime Libraries from JDeveloper".
- For WebSphere, you have to install the ADF runtime libraries manually. See Section 34.12.2, "Configuring WebSphere 6.0.1 to Run ADF Applications".
- For all application servers, you can install the ADF runtime libraries manually. See Section 34.12.3, "Installing the ADF Runtime Libraries Manually".

34.12.1 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 34.12.2, "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.

   <u>Steps for Creating Domains in WebLogic 8.1:</u>

   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`.

   <u>Steps for Creating Domains in WebLogic 9.0:</u>

   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 34-19.
   - 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 34-19.

   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 34-19.

   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.

34.12.2 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 34.12.2.1, "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 34.12.2.2, "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.pro
   cessor.TransformerFactoryImpl
   ```
7. Shut down and restart WebSphere so that it uses the new parameter.

34.12.2.1 Source for install_adflibs_1013.sh Script

Example 34–1 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 34.12.2, "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 34.12.2.2, "Source for install_adflibs_1013.cmd Script".

Example 34–1 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/


#  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
```

34.12.2.2 Source for install_adflibs_1013.cmd Script

Example 34–2 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 34.12.2, "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 34.12.2.1, "Source for install_adflibs_1013.sh Script".

Example 34–2 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 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
```

34.12.3 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 34–8 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 34–8 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 34–8 (Cont.) ADF Runtime Library Files to Copy

Copy These Files:	Notes
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:

34.12.3.1 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
   ```

34.12.4 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.

34.13 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 34.13.1, "How to Test Run Your Application"
- Section 34.13.2, ""Class Not Found" or "Method Not Found" Errors"
- Section 34.13.3, "Application Is Not Using data-sources.xml File on Target Application Server"
- Section 34.13.4, "Using jazn-data.xml with the Embedded OC4J Server"
- Section 34.13.5, "Error "JBO-30003: The application pool failed to check out an application module due to the following exception""

34.13.1 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 an URL of the following type:

- For Oracle AS: http://<host>:port/<context root>/<page>
- For Faces pages: http://<host>:port/<context root>/faces/<page>

34.13.2 "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).

34.13.3 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.

34.13.4 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).

34.13.5 Error "JBO-30003: The application pool failed to check out an application module due to the following exception"

Problem

You get the following error in the error log:

```
05/11/07 18:12:59.67 10.1.3.0.0 Started
05/11/07 18:13:05.687 id: 10.1.3.0.0 Started
05/11/07 18:13:38.224 id: Servlet error
JBO-30003: The application pool (<class_name>) failed to checkout an application
 module due to the following exception:
oracle.jbo.JboException: JBO-29000: Unexpected exception caught:
oracle.jbo.JboException, msg=JBO-29000: Unexpected exception caught:
oracle.classloader.util.AnnotatedClassFormatError, msg=<classname> (Unsupported
 major.minor version 49.0)

      Invalid class: <classname>
             Loader: webapp5.web.id:0.0.0
        Code-Source:
/C:/oc4j/j2ee/home/applications/webapp5/webapp5/WEB-INF/classes/
      Configuration: WEB-INF/classes/ in
C:\oc4j\j2ee\home\applications\webapp5\webapp5\WEB-INF\classes

    Dependent class: oracle.jbo.common.java2.JDK2ClassLoader
             Loader: adf.oracle.domain:10.1.3
        Code-Source: /C:/oc4j/BC4J/lib/adfm.jar
      Configuration: <code-source> in /C:/oc4j/j2ee/home/config/server.xml

    at
oracle.jbo.common.ampool.ApplicationPoolImpl.doCheckout(ApplicationPoolImpl.java:1
892)
```

Solution

A possible cause of this exception is that the application was unable to connect to the database for its data bindings. Check that you have set up the required database connections in your target application server environment, and that the connections are working.

Part V

Appendices

Part V contains the following appendices:

- Appendix A, "Reference ADF XML Files"
- Appendix B, "Reference ADF Binding Properties"
- Appendix C, "ADF Equivalents of Common Oracle Forms Triggers"
- Appendix D, "Most Commonly Used ADF Business Components Methods"
- Appendix E, "ADF Business Components J2EE Design Pattern Catalog"

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, "bc4j.xcfg"
- Appendix A.5, "DataBindings.cpx"
- Appendix A.6, "<pageName>PageDef.xml"
- Appendix A.7, "web.xml"
- Appendix A.8, "j2ee-logging.xml"
- Appendix A.9, "faces-config.xml"
- Appendix A.10, "adf-faces-config.xml"
- Appendix A.11, "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.

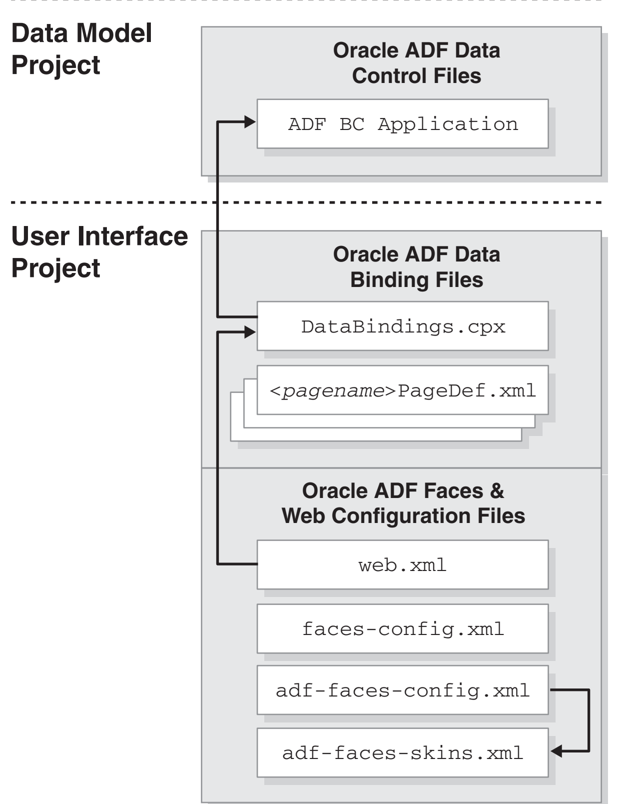
Additionally, you can view and edit the contents of any metadata file in JDeveloper's XML editor. The easiest way to work with these file is through the Structure window and Property Inspector. In the Structure window, you can select an element and in the Property Inspector, you can define attribute 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 ADF Business Components application module as a service interface to JSF web pages.

Figure A–1 Oracle ADF File Hierarchy Overview for an ADF BC-based Web Application



A.2.1 Oracle ADF Data Control Files

In an ADF Business Components application, the data control implementation files are contained within the application. The application module and view object XML component descriptor files provide the references for the data control. These files, in conjunction with the `bc4j.xcfg` file (see Section A.4) provide the necessary information for the data control.

A workspace that uses ADF Business Components in one project and a non-ADF Business Components data control in another project, like the SRDemoFAQ data

control, may have a `DataControls.dcx` file, as well as supporting *<sessionbeanname>*`.xml` and *<beanname>*`.xml` files. For more information on the non-ADF Business Components data controls, see the Oracle Application Development Framework Developers Guide 10g for J2EE Developers.

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.5, "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.6, "<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.7, "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.9, "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.

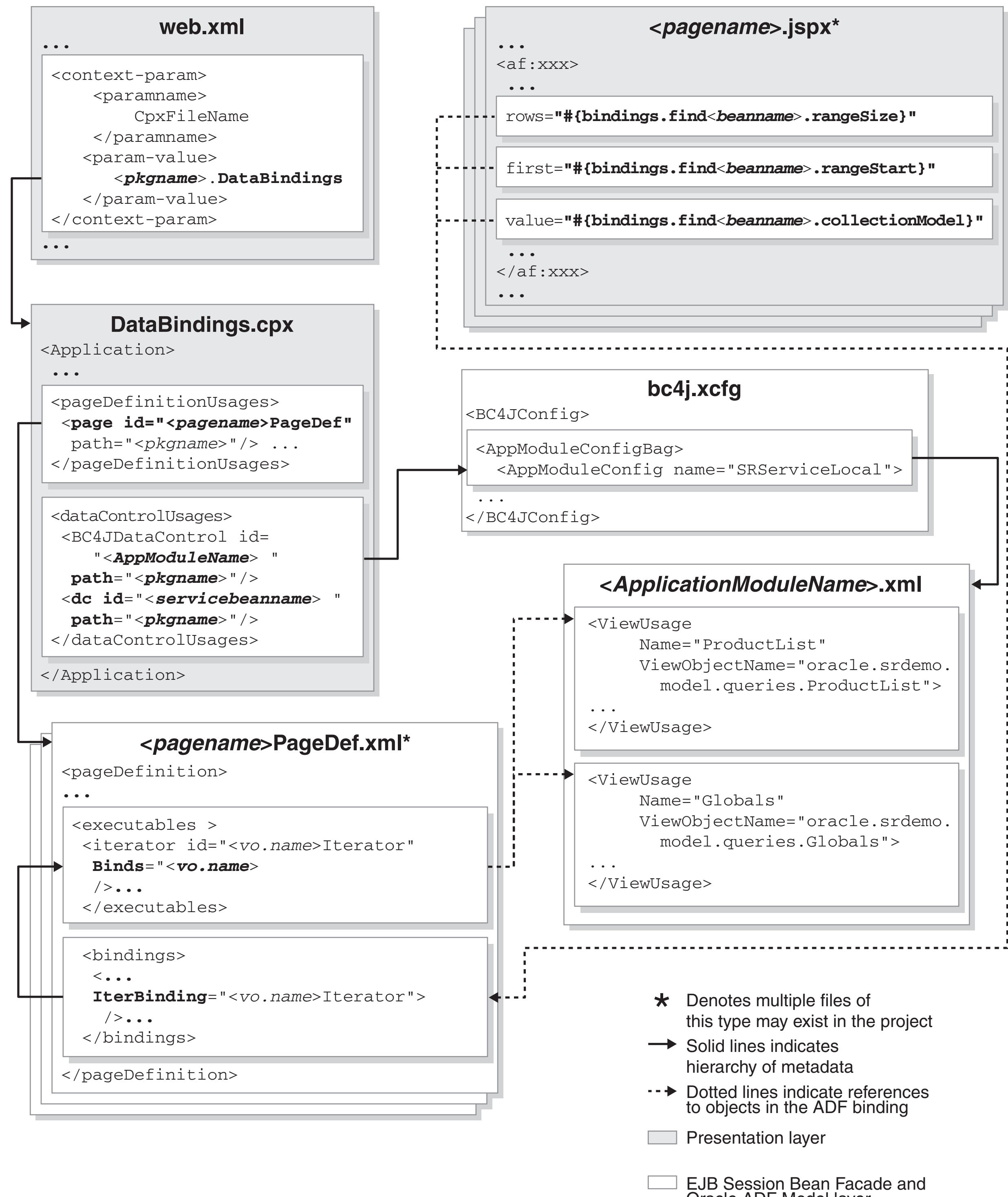
For details about ADF Faces configuration options, see Appendix A.10, "adf-faces-config.xml".

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 ADF application module as a service interface to ADF Business Components. At runtime, the objects created from these files interact in this sequence:

1. When the first request for an ADF databound web page occurs, the servlet registers the Oracle ADF servlet filter `ADFBindingFilter` named in the `web.xml` file.
2. The binding filter creates 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 application module 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 ADF BC-based Web Application



A.4 bc4j.xcfg

The `bc4j.xcfg` file contains information about application module names and the runtime parameters the user has configured. A sample `bc4j.xcfg` from the SRDemo application follows:

Example A–1 bc4j.xcfg

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<BC4JConfig>
   <AppModuleConfigBag>
      <AppModuleConfig name="SRServiceLocal">
         <DeployPlatform>LOCAL</DeployPlatform>
         <JDBCDataSource>java:comp/env/jdbc/SRDemoDS</JDBCDataSource>
         <jbo.project>DataModel</jbo.project>
         <jbo.locking.mode>optimistic</jbo.locking.mode>
         <AppModuleJndiName>oracle.srdemo.model.SRService</AppModuleJndiName>
         <jbo.security.enforce>Must</jbo.security.enforce>

<java.naming.factory.initial>oracle.jbo.common.JboInitialContextFactory</java.nami
ng.factory.initial>
         <ApplicationName>oracle.srdemo.model.SRService</ApplicationName>
         <jbo.server.internal_
connection>java:comp/env/jdbc/SRDemoCoreDS</jbo.server.internal_connection>
      </AppModuleConfig>
      <AppModuleConfig name="SRServiceLocalTesting">
         <DeployPlatform>LOCAL</DeployPlatform>
         <JDBCName>SRDemo</JDBCName>
         <jbo.project>DataModel</jbo.project>
         <AppModuleJndiName>oracle.srdemo.model.SRService</AppModuleJndiName>
         <jbo.locking.mode>optimistic</jbo.locking.mode>
         <jbo.security.enforce>Must</jbo.security.enforce>

<java.naming.factory.initial>oracle.jbo.common.JboInitialContextFactory</java.nami
ng.factory.initial>
         <ApplicationName>oracle.srdemo.model.SRService</ApplicationName>
      </AppModuleConfig>
   </AppModuleConfigBag>
   <ConnectionDefinition name="SRDemo">
      <ENTRY name="JDBC_PORT" value="1521"/>
      <ENTRY name="ConnectionType" value="JDBC"/>
      <ENTRY name="HOSTNAME" value="bordello.us.oracle.com"/>
      <ENTRY name="DeployPassword" value="true"/>
      <ENTRY name="user" value="SRDemo"/>
      <ENTRY name="ConnectionName" value="SRDemo"/>
      <ENTRY name="SID" value="bordello"/>
      <ENTRY name="password">
         <![CDATA[{904}05EF4D2067E3477EBE0CF3865966EB7C5F]]>
      </ENTRY>
      <ENTRY name="JdbcDriver" value="oracle.jdbc.OracleDriver"/>
      <ENTRY name="ORACLE_JDBC_TYPE" value="thin"/>
      <ENTRY name="DeployPassword" value="true"/>
   </ConnectionDefinition>
</BC4JConfig>
```

A.5 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.5.1 DataBindings.cpx Syntax

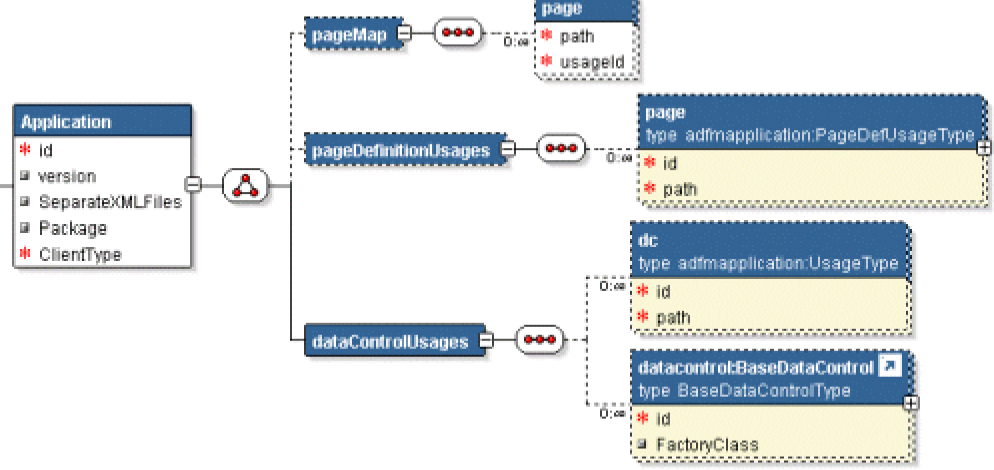
The toplevel element of the DataBindings.cpx file is <DataControlConfigs>:

```
<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–3 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–3 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–1 describes the attributes of the DataBindings.cpx elements.

Table A–1 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.5.2 DataBindings.cpx Sample

Example A–2 shows the syntax for the DataBindings.cpx file in the SR Demo application. The data controls are named by the id="<*name*>". The combination of the Package attribute and the Configuration attribute are used to locate the bc4j.xcfg file in the "./common" subdirectory of the indicated package. The configuration contains the information of the application module name and all the runtime parameters the user has configured.

Example A–2 DataBindings.cpx file

```
<dataControlUsages>
  <dc id="SRDemoFAQ" path="oracle.srdemo.faq.SRDemoFAQ"/>
  <BC4JDataControl id="SRService" Package="oracle.srdemo.model"
                   FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
                   SupportsTransactions="true" SupportsFindMode="true"
                   SupportsRangesize="true" SupportsResetState="true"
                   SupportsSortCollection="true"
                   Configuration="SRServiceLocal" syncMode="Immediate"
              xmlns="http://xmlns.oracle.com/adfm/datacontrol"/>
</dataControlUsages>
```

A.6 <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.6.1 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–3 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–3 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, view objects, and accessors) to execute during the prepareModel phase of the ADF page lifecycle. Methods to be executed are defined by <methodIterator>. The lifecycle performs the execute in the sequence listed in the <executables> section. Whether or not the method or

operation is executed depends on 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–2 describes the attributes of the toplevel <pageDefinition> element.

Table A–2 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–3 describes the attributes of the child element of <parameters>.

Table A–3 Attributes of the PageDef.xml File <parameters> Element

Element Syntax	Attributes	Attribute Description
`<parameter>`	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–4 describes the attributes of the PageDef.xml <executables> elements.

Table A–4 Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
`<accessorIterator>`	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–4 (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–4 (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–4 (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–5 describes the attributes of the PageDef.xml <bindings> element.

Table A–5 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–5 (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–5 (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–5 (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.
	IsViewObjectMethod	Set to True if the instanceName contains an instance-path relative to the associated data control's Application Module.
	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.

Table A–5 (Cont.) Attributes of the PageDef.xml File <bindings> Elements

Element Syntax	Attributes	Attribute Description
	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.6.2 PageDef.xml Sample for Attributes of a View Object

This is the page definition file that's created when you drop attributes of the LoggedInUser view object from the Data Control Palette, SRService node, into your open JSP page.

Example A–4 PageDef for Attributes of LoggedInUser View Object

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.35.65" id="app_SRCreateConfirmPageDef"
                Package="oracle.srdemo.view.pageDefs">
  <parameters/>
  <executables>
    <iterator id="GlobalsIterator" RangeSize="10" Binds="Globals"
              DataControl="SRService"/>
    <iterator id="LoggedInUserIterator" RangeSize="10" Binds="LoggedInUser"
              DataControl="SRService"/>
  </executables>
  <bindings>
    <attributeValues IterBinding="GlobalsIterator" id="ProblemDescription">
      <AttrNames>
        <Item Value="ProblemDescription"/>
      </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="GlobalsIterator" id="ProductId">
      <AttrNames>
        <Item Value="ProductId"/>
      </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="GlobalsIterator" id="ProductName">
      <AttrNames>
        <Item Value="ProductName"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="FirstName"
                     IterBinding="LoggedInUserIterator">
      <AttrNames>
        <Item Value="FirstName"/>
      </AttrNames>
    </attributeValues>
    <attributeValues id="LastName"
                     IterBinding="LoggedInUserIterator">
      <AttrNames>
        <Item Value="LastName"/>
      </AttrNames>
    </attributeValues>
    ...
```

```
  </bindings>
</pageDefinition>
```

A.6.3 PageDef.xml Sample for the Entire View Object

This is the page definition file that's created when you drop the ServiceRequestsByStatus node from the Data Control Palette, SRService node, into your open JSP page. This one produces the entire view object.

Example A–5 PageDef for the Entire View Object

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.35.65" id="app_SRListPageDef"
                Package="oracle.srdemo.view.pageDefs"
                EnableTokenValidation="false">
  <parameters/>
  <executables>
    <iterator id="ServiceRequestsByStatusIterator" RangeSize="10"
              Binds="ServiceRequestsByStatus" DataControl="SRService"/>
  </executables>
  <bindings>
    <table id="LoggedInUserServiceRequests"
           IterBinding="ServiceRequestsByStatusIterator">
      <AttrNames>
        <Item Value="SvrId"/>
        <Item Value="Status"/>
        <Item Value="RequestDate"/>
        <Item Value="ProblemDescription"/>
        <Item Value="ProdId"/>
        <Item Value="CreatedBy"/>
        <Item Value="AssignedTo"/>
        <Item Value="AssignedDate"/>
      </AttrNames>
    </table>
    <action id="setCurrentRowWithKey"
            IterBinding="ServiceRequestsByStatusIterator"
            InstanceName="SRService.ServiceRequestsByStatus"
            DataControl="SRService" RequiresUpdateModel="false" Action="96">
      <NamedData NDName="rowKey" NDValue="${row.rowKeyStr}"
                 NDType="java.lang.String"/>
    </action>
    <action id="ExecuteWithParams" IterBinding="ServiceRequestsByStatusIterator"
            InstanceName="SRService.ServiceRequestsByStatus"
            DataControl="SRService" RequiresUpdateModel="true"
            Action="95">
      <NamedData NDName="StatusCode" NDValue="${userState.listMode}"
                 NDType="java.lang.String"/>
    </action>
  </bindings>
</pageDefinition>
```

A.7 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–6 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.7.1.1, "Configuring for State Saving" through Appendix A.7.1.7, "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.7.1 Tasks Supported by the web.xml File

The following JSF and ADF Faces tasks are supported by the `web.xml` file.

A.7.1.1 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–7 shows part of a web.xml file that contains state-saving parameters.

Example A–7 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.7.1.2 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–8 shows part of a web.xml file that contains the application view caching parameter.

Example A–8 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.7.1.3 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–9 shows part of a web.xml file that contains debugging parameters.

Example A–9 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.7.1.4 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–10 shows part of a `web.xml` file that contains file upload parameters.

Example A–10 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 19.6, "Providing File Upload Capability".

A.7.1.5 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 12.3, "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–11 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 12, "Displaying Data on a Page".

Example A–11 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.7.1.6 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–12). You need to set this parameter if you use more than one JSF configuration file in your application, as described in Appendix A.9.1, "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–12 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.7.1.7 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.8 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.8.1.1.
- Redirect the log output to a location, like a file, in addition to the default Log window in JDeveloper. See Section A.8.1.2.
- Change the directory path that determines where your log file resides. See Section A.8.1.3.

A.8.1 Tasks Supported by the j2ee-logging.xml

The following JSF tasks are supported by the `j2ee-logging.xml` file.

A.8.1.1 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–13). 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 24.4.2, "Creating an Oracle ADF Debugging Configuration".

Example A–13 Changing the Logging Level in j2ee-logging.xml

```
<loggers>
   <logger name="oracle.adf" level="FINE" />
   ...
</loggers>
```

A.8.1.2 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–14). 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–14 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.8.1.3 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–15). The default directory for the log file is `../log/oc4j`.

Example A–15 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.9 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 11.2.3, "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.9.1.1, Section A.9.1.3. and Section A.9.1.4.
- Page-to-page navigation rules. See Section A.9.1.5.
- Custom validators and converters. See Section A.9.1.6.
- 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.9.1.2.

A.9.1 Tasks Supported by the faces-config.xml

The following JSF tasks are supported by the `faces-config.xml` file.

A.9.1.1 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–16).

Example A–16 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.9.1.2 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 13.2.3, "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–17 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 20.8, "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–17 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.9.1.3 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–18). The SRDemo application uses a resource properties file to hold the strings for the UI.

Example A–18 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 22.4, "Internationalizing Your Application".

A.9.1.4 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–19).

Example A–19 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.9.1.4.1 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.9.1.5 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–20). The corresponding code for the navigation case within the navigation rule for the SRList page is shown in Example A–21.

Example A–20 Action Outcome String Defined on View Button

```
<af:commandButton text="#{res['srlist.buttonbar.view']}"
                  action="View"/>
```

Example A–21 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 16, "Adding Page Navigation".

A.9.1.6 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–22). 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–22 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.9.1.7 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–23). 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–23 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.10 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–24 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–24 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.10.1 Tasks Supported by adf-faces-config.xml

The following JSF tasks are supported by the `adf-faces-config.xml` file.

A.10.1.1 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–25 Configuring an Accessibility Level

```
<!-- Set the accessibility mode to screenReader -->
<accessibility-mode>screenReader</accessibility-mode>
```

A.10.1.2 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–26 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.10.1.3 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–27 Enabling Enhanced Debugging

```
<!-- Activate the ADF Faces enhanced debugging features -->
<debug-output>true</debug-output>
```

A.10.1.4 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–28 Turning Off Client-Side Validation and Conversion

```
<!-- Disable client validation -->
<client-validation-disabled>true</client-validation-disabled>
```

A.10.1.5 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–29 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.10.1.6 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.11.1, "Tasks Supported by adf-faces-skins.xml".

For information about creating custom skins, see Section 22.3, "Using Skins to Change the Look and Feel".

Example A–30 Configuring a Skin to be Used For All Pages

```
<!-- Specify custom skin instead of Oracle skin -->
<skin-family>srdemo<skin-family>
```

A.10.1.7 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–31 Configuring an Output Mode

```
<!-- Set the output mode to printable -->
<output-mode>printable</output-mode>
```

A.10.1.8 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–32 Configuring the Number of Active ProcessScope Instances

```
<!-- Set the maximum number of processScope instances to 10 -->
<process-scope-lifetime>10</process-scope-lifetime>
```

A.10.1.9 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–33 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.10.1.10 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–34 Configuring a Custom Uploaded File Processor

```
<!-- Use my UploadFileProcessor class -->
<uploaded-file-processor>
 com.mycompany.faces.myUploadedFileProcessor
</uploaded-file-processor>
```

A.10.1.11 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–35 Configuring the Help Site URL

```
<!-- Set the help site URL -->
<oracle-help-servlet-url>mywebsite.com/project_one/help</oracle-help-servlet-url>
```

A.10.1.12 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.11 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–36 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.11.1 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.10.1.6, "Configuring the Skin Family".

For information about creating custom skins, see Section 22.3, "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 Note.

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

C

ADF Equivalents of Common Oracle Forms Triggers

This appendix provides a quick summary of how basic tasks performed using the most common Oracle Forms triggers are accomplished using Oracle ADF.

This appendix contains the following sections:

- Appendix C.1, "Validation & Defaulting (Business Logic)"
- Appendix C.2, "Query Processing"
- Appendix C.3, "Database Connection"
- Appendix C.4, "Transaction "Post" Processing (Record Cache)"
- Appendix C.5, "Error Handling"

C.1 Validation & Defaulting (Business Logic)

Table C–1 ADF Equivalents for Oracle Forms Validation and Defaulting Triggers

Forms Trigger	ADF Equivalent
`WHEN-VALIDATE-RECORD` Execute validation code at the record level	In the custom EntityImpl class for your entity object, write a public method returning `boolean` type with a method name like `validateXXXX()` and have it return `true` if the validation succeeds or `false` if the validation fails. Then, add a method validator for this validation method at the entity level on the "Validation" panel for your entity object. When doing that you can associate a validation failure message with the rule at that time.
`WHEN-VALIDATE-ITEM` Execute validation code at the field level	In the custom EntityImpl class for your entity object, write a public method returning `boolean` type and accepting a single argument of the same datatype as your attribute, having a method name like `validateXXXX()`. Have it return `true` if the validation succeeds or `false` if the validation fails. Then, add a method validator for this validation method at the entity attribute level for the appropriate attribute on the "Validation" panel for your entity object. When doing that you can associate a validation failure message with the rule at that time.

Table C–1 (Cont.) ADF Equivalents for Oracle Forms Validation and Defaulting Triggers

Forms Trigger	ADF Equivalent
`WHEN-DATABASE-RECORD` Execute code when a row in the datablock is marked for INSERT or UPDATE	Override the `addToTransactionManager()` method of your entity object. Write code after calling the super.
`WHEN-CREATE-RECORD` Execute code to populate complex default values when a new record in the data block is created, without changing the modification status of the record.	Override the `create()` method of your entity object and after calling super, use appropriate `set`*`AttrName`*`()` methods to set default values for attributes as necessary. To eagerly set a primary key attribute to the value of a sequence, construct an instance of the `SequenceImpl` helper class and call its `getSequenceNumber()` method to get the next sequence number. Assign this value to your primary key attribute. If you want to assign the sequence number lazily, but still without using a database trigger, you can use the technique above in an overridden `prepareForDML()` method in your entity object. If instead you want to assign the primary key from a sequence using your own `BEFOREINSERTFOREACHROW` database trigger, then use the special datatype called `DBSequence` for your primary key attribute instead of the regular `Number` type.
`WHEN-REMOVE-RECORD` Execute code whenever a row is removed from the data block.	Override the `remove()` method of your entity object and write code either before or after calling super.

C.2 Query Processing

Table C–2 ADF Equivalents for Oracle Forms Query Processing Triggers

Forms Trigger	ADF Equivalent
`PRE-QUERY` Execute logic before executing a query in a Data Block, typically to setup values for query-by-example criteria in the "example record".	Override `executeQueryForCollection()` on your view object class and write code before calling the super.
`ON-COUNT` Override default behavior to count the query hits for a Data Block	Override `getQueryHitCount()` in your view object and do something instead of calling the super.
`POST-QUERY` Execute logic after retrieving each row from the datasource for a data block.	Generally instead of using a `POST-QUERY` style technique to fetch descriptions from other tables based on foreign key values in the current row, in ADF it's more efficient to build a view object that has multiple participating entity objects, joining in all the information you need in the query from the main table, as well as any auxiliary/lookup-value tables. This way, in a single round-trip to the database you get all the information you need. If you still need a per-fetched-row trigger like `POST-QUERY`, override the `createInstanceFromResultSet()` method in your view object class.

Table C–2 (Cont.) ADF Equivalents for Oracle Forms Query Processing Triggers

Forms Trigger	ADF Equivalent
`ON-LOCK` Override default behavior to attempt to acquire lock on the current row in the data block.	Override the `lock()` method in your entity object class and do something instead of calling the super.

C.3 Database Connection

Table C–3 ADF Equivalents for Oracle Forms Database Connection Triggers

Forms Trigger	ADF Equivalent
`POST-LOGON` Execute logic after logging onto the Database	Override `afterConnect()` on your custom application module. Since application module instances can stay connected while serving different logical client sessions, probably what you want is to override the `prepareSession()` which is fired after initial login, as well as after any time the application module is used by a user that was different from the one that used it last time.
`PRE-LOGOUT` Execute logic before logging off from the Database	Override `beforeDisconnect()` on your custom application module class and write code.

C.4 Transaction "Post" Processing (Record Cache)

Table C–4 ADF Equivalents for Oracle Forms Transactional Triggers

Forms Trigger	ADF Equivalent
`PRE-COMMIT` Execute code before commencing processing of the changed rows in all data blocks in the transaction.	Override `commit()` method in a custom `DBTransactionImpl` class and write code before calling the super. **Note:** See this article for an overview of creating and using a custom DBTransaction implementation.
`PRE-INSERT` Execute code before NEW row in the datablock is INSERTed into the database during "post" processing.	Override `doDML()` method in your entity class and if the `operation` equals `DML_INSERT` then write code *before* calling the super.
`ON-INSERT` Override default processing for INSERTing a NEW row into the database during "post" processing.	Override `doDML()` method in your entity class and if the `operation` equals `DML_INSERT` then write code *instead of* calling the super.
`POST-INSERT` Execute code after NEW row in the datablock is INSERTed into the database during "post" processing.	Override `doDML()` method in your entity class and if the `operation` equals `DML_INSERT` then write code after calling the super.

Table C–4 (Cont.) ADF Equivalents for Oracle Forms Transactional Triggers

Forms Trigger	ADF Equivalent
`PRE-DELETE` Execute code before row removed from the datablock is DELETEd from the database during "post" processing.	Override `doDML()` method in your entity class and if the `operation` equals `DML_DELETE` then write code *before* calling the super.
`ON-DELETE` Override default processing for DELETE-ing a row removed from the datablock from the database during "post" processing.	Override `doDML()` method in your entity class and if the `operation` equals `DML_DELETE` then write code *instead of* calling the super.
`POST-DELETE` Execute code after row removed from the datablock is DELETEd from the database during "post" processing.	Override `doDML()` method in your entity class and if the `operation` equals `DML_DELETE` then write code *after* calling the super.
`PRE-UPDATE` Execute code before row changed in the datablock is UPDATEd in the database during "post" processing.	Override `doDML()` method in your entity class and if the `operation` equals `DML_UPDATE` then write code *before* calling the super.
`ON-UPDATE` Override default processing for UPDATE-ing a row changed in the datablock from the database during "post" processing.	Override `doDML()` method in your entity class and if the `operation` equals `DML_UPDATE` then write code *instead of* calling the super.
`POST-UPDATE` Execute code after row changed in the datablock is UPDATEd in the database during "post" processing.	Override `doDML()` method in your entity class and if the `operation` equals `DML_UPDATE` then write code *after* calling the super.
`POST-FORMS-COMMIT` Execute code after Forms has "posted" all necessary rows to the database, but before issuing the data COMMIT to end the transaction.	If you want a single block of code for the whole transaction, you can override the `doCommit()` method in a custom `DBTransactionImpl` object and write some code before calling super. To execute entity-specific code before commit for each affected entity in the transaction, override the `beforeCommit()` method on your entity object and write some code there.
`POST-DATABASE-COMMIT` Execute code after database transaction has been committed.	Override `commit()` method in a custom `DBTransactionImpl` class and write code *after* calling the super.

C.5 Error Handling

Table C–5 ADF Equivalents for Oracle Forms Error Handling Triggers

Forms Trigger	ADF Equivalent
`ON-ERROR` Override default behavior for handling an error.	Install a custom error handler (`DCErrorHandler`) on the ADF `BindingContext`

D

Most Commonly Used ADF Business Components Methods

This appendix highlights the most commonly used methods in the interfaces and classes of the ADF Business Components layer of Oracle ADF.

This appendix contains the following sections:

- Appendix D.1, "Most Commonly Used Methods in the Client Tier"
- Appendix D.2, "Most Commonly Used Methods In the Business Service Tier"

D.1 Most Commonly Used Methods in the Client Tier

All of the interfaces described in this section are designed for use by client-layer code and are part of the `oracle.jbo.*` package.

> **Note:** The corresponding implementation classes for these `oracle.jbo.*` interfaces are consciously designed to *not* be directly accessed by client code. As you'll see in the Section D.2, "Most Commonly Used Methods In the Business Service Tier" section below, the implementation classes live in the `oracle.jbo.server.*` package and generally have the suffix `Impl` in their name to help remind you not to using them in your client-layer code.

D.1.1 ApplicationModule Interface

The ApplicationModule is a business service component that acts as a transactional container for other ADF components and coordinates *with* them to implement a number of J2EE design patterns important to business application developers. These design pattern implementations enable your client code to work easily with updatable collections of value objects, based on fast-lane reader SQL queries that retrieve only the data needed by the client, in the way the client wants to view it. Changes made to these value objects are automatically coordinated with your persistent business domain objects in the business service tier to enforce business rules consistently and save changes back to the database.

Table D–1 ApplicationModule Interface

If you want to...	Call this ApplicationModule interface method...
Access an existing view object instance by name	`findViewObject()`
Creating a new view object instance from an existing definition	`createViewObject()`
Creating a new view object instance from a SQL Statement	`createViewObjectFromQueryStmt()` **Note:** This incurs runtime overhead to describe the "shape" of the dynamic query's `SELECT` list. Oracle recommends using this only when you cannot know the SELECT list for the query at design-time. Furthermore, if you are creating the dynamic query based on some kind of custom runtime repository, you can follow this tip to create (both read-only and updatable) dynamic view objects without the runtime-describe overhead with a little more work. If only the `WHERE` needs to be dynamic, create the view object at design time, then set the where clause dynamically as needed using `ViewObject` API's.
Access a nested application module instance by name	`findApplicationModule()`
Create a new nested application module instance from an existing definition	`createApplicationModule()`
Find a view object instance in a nested application module	`findViewObject()` **Note:** To find an instance of a view object belonging to a nested application module you use a dot notation *`nestedAMInstanceName.VOInstanceName`*
Accessing the current transaction object	`getTransaction()`

In addition to generic `ApplicationModule` access, Oracle JDeveloper 10g can generate you a custom *`YourApplicationModuleName`* interface containing service-level custom methods that you've chosen to expose to the client. You do this by visiting the **Client Interface** tab of the Application Module editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list. JDeveloper will also generate an appropriate *`YourApplicationModuleName`*`Client` client proxy implementation class that is used automatically by your remote client in the

case that you deploy your application module as an EJB Session Bean or whenever you use your application module in Batch Mode.

D.1.2 Transaction Interface

The Transaction interface exposes methods allowing the client to manage pending changes in the current transaction.

Table D–2 Transaction Interface

If you want to...	Call this Transaction interface method...
Commit pending changes	`commit()`
Rollback pending changes	`rollback()`
Execute a one-time database command or block of PL/SQL	`executeCommand()` **Note:** Commands that require retrieving `OUT` parameters, that will be executed more than once, or that could benefit by using bind variables should not use this method. Instead, expose a custom method on your application module.
Validate all pending invalid changes in the transaction	`validate()`
Change the default locking mode	`setLockingMode()` **Note:** You can set the locking mode in your configuration by setting the property `jbo.locking.mode` to one of the four supported values: `none`, `optimistic`, `pessimistic`, `optupdate`. If you don't explicitly set it, it will default to `pessimistic`. For web applications, Oracle recommends using `optimistic` or `optupdate` modes.
Decide whether to use bundled exception reporting mode or not.	`setBundledExceptionMode()` **Note:** ADF controller layer support sets this parameter to `true` automatically for web applications.
Decide whether entity caches will be cleared upon a successful commit of the transaction.	`setClearCacheOnCommit()` **Note:** Default is `false`
Decide whether entity caches will be cleared upon a rollback of the transaction.	`setClearCacheOnRollback()` **Note:** Default is `true`
Clear the entity cache for a specific entity object.	`clearEntityCache()`

D.1.3 ViewObject Interface

A ViewObject encapsulates a database query and simplifies working with the `RowSet` of results it produces. You use view objects to project, filter, join, or sort business data using SQL from one or more tables into exactly the format that the user should see it

on the page or panel. You can create "master/detail" hierarchies of any level of depth or complexity by connecting view objects together using view links. View objects can produce read-only query results, or by associating them with one ore more entity objects at design time, can be fully updatable. Updatable view objects can support insertion, modification, and deletion of rows in the result collection, with automatic delegation to the correct business domain objects.

Every `ViewObject` aggregates a "default rowset" for simplifying the 90% of use cases where you work with a single `RowSet` of results for the `ViewObject`'s query. A `ViewObject` implements all the methods on the `RowSet` interface by delegating them to this default `RowSet`. That means you can invoke any `RowSet` methods on any `ViewObject` as well.

Every `ViewObject` implements the `StructureDef` interface to provide information about the number and types of attributes in a row of its row sets. So you can call `StructureDef` methods right on any view object.

Table D–3 ViewObject Interface

If you want to...	Call this ViewObject interface method...
Set an additional runtime WHERE clause on the rowset	`setWhereClause()` **Note:** This `WHERE` clause augments any `WHERE` clause specified at design time in the base view object. It does not replace it.
Set a dynamic ORDER BY clause	`setOrderByClause()`
Create a Query-by-Example criteria collection	`createViewCriteria()` **Note:** You then create one or more `ViewCriteriaRow` objects using the `createViewCriteriaRow()` method on the ViewCriteria object you created. Then, you `add()` these view criteria rows to the view criteria collection and apply the criteria using the method below.
Apply a Query-by-Example criteria collection	`applyViewCriteria()`
Set a query optimizer hint	`setQueryOptimizerHint()`
Access the attribute definitions for the key attributes in the view object	`getKeyAttributeDefs()`
Add a dynamic attribute to rows in this view object's row sets	`addDynamicAttribute()`
Clear all row sets produced by a view object	`clearCache()`
Remove view object instance an its resources	`remove()`
Set an upper limit on the number of rows that the view object will attempt to fetch from the database.	`setMaxFetchSize()` **Note:** Default is -1 which means to impose no limit on how many rows would be retrieved from the database if you iterate through them all. By default they are fetched lazily as you iterate through them.

In addition to generic `ViewObject` access, JDeveloper 10g can generate you a custom *`YourViewObjectName`* interface containing view-object level custom methods that

you've chosen to expose to the client. You do this by visiting the **Client Interface** tab of the View Object editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list. JDeveloper will also generate an appropriate `*YourViewObjectName*Client` client proxy implementation class that is used automatically by your remote client in the case that you deploy your application module as an EJB Session Bean or whenever you use your application module in Batch Mode.

D.1.4 RowSet Interface

A RowSet is a set of rows, typically produced by executing a `ViewObject`'s query.

Every `RowSet` aggregates a "default rowset iterator" for simplifying the 90% of use cases where you only need a single iterator over the rowset. A `RowSet` implements all the methods on the `RowSetIterator` interface by delegating them to this default `RowSetIterator`. This means you can invoke any `RowSetIterator` method on any `RowSet` (or `ViewObject`, since it implements RowSet as well for its default RowSet).

Table D–4 RowSet Interface

If you want to...	Call this RowSet interface method...
Set a where clause bind variable value	`setWhereClauseParam()` **Note:** Bind variable ordinal positions are zero-based
Avoid view object row caching if data is being read only once	`setForwardOnly()`
Force a row set's query to be (re)executed	`executeQuery()`
Estimate the number of rows in a view object's query result	`getEstimatedRowCount()`
Produce XML document for rows in View Object rowset	`writeXML()`
Process all rows from an incoming XML document	`readXML()`
Set whether rowset will automatically see new rows based on the same entity object created through other rowsets	`setAssociationConsistent()`
Create secondary iterator to use for programmatic iteration	`createRowSetIterator()` **Note:** If you plan to find and use the secondary iterator by name later, then pass in a string name as the argument, otherwise pass `null` for the name and make sure to close the iterator when done iterating by calling its `closeRowSetIterator()` method.

D.1.5 RowSetIterator Interface

A RowSetIterator is an iterator over the rows in a `RowSet`. By default it allows you to iterate both forward and backward through the rows.

Table D–5 RowSetIterator Interface

If you want to...	Call this RowSetIterator interface method...
Get the first row of the iterator's rowset	`first()`

Table D–5 (Cont.) RowSetIterator Interface

If you want to...	Call this RowSetIterator interface method...
Test whether there are more rows to iterate	`hasNext()`
Get the next row of iterator's rowset	`next()`
Find row in this iterator's rowset with a given Key value	`findByKey()` **Note:** It's important that the `Key` object that you pass to `findByKey` be created using the *exact* same datatypes as the attributes that comprise the key of the rows in the view object you're working with.
Create a new row to populate for insertion	`createRow()` **Note:** The new row will already have default values set for attributes which either have a static default value supplied at the entity object or view object level, or if the values have been populated in an overridden `create()` method of the underlying entity object(s).
Create a view row with an initial set of foreign key and/or discriminator attribute values	`createAndInitRow()` **Note:** You use this method when working with view objects that can return one of a "family" of entity object subtypes. By passing in the correct discriminator attribute value in the call to create the row, the framework can create you the correct matching entity object subtype underneath.
Insert a new row into the iterator's rowset	`insertRow()` **Note:** It's a good habit to always immediately insert a newly created row into the rowset. That way you will avoid a common gotcha of creating the row but forgetting to insert it into the rowset.
Get the last row of the iterator's rowset	`last()`
Get the previous row of the iterator's rowset	`previous()`
Reset the current row pointer to the slot before the first row	`reset()`
Close an iterator when done iterating	`closeRowSetIterator()`
Set a given row to be the current row	`setCurrentRow()`
Remove the current row	`removeCurrentRow()`
Remove the current row to later insert it at a different location in the same iterator.	`removeCurrentRowAndRetain()`
Remove the current row from the current collection but do not remove it from the transaction.	`removeCurrentRowFromCollection()`
Set/change the number of rows in the range (a "page" of rows the user can see)	`setRangeSize()`
Scroll to view the Nth page of rows (1-based)	`scrollToRangePage()`

Table D–5 (Cont.) RowSetIterator Interface

If you want to...	Call this RowSetIterator interface method...
Scroll to view the range of rows starting with row number N	`scrollRangeTo()`
Set row number N in the range to be the current row	`setCurrentRowAtRangeIndex()`
Get all rows in the range as a Row array	`getAllRowsInRange()`

D.1.6 Row Interface

A Row is generic value object. It contains attributes appropriate in name and Java type for the `ViewObject` that it's related to.

Table D–6 Row Interface

If you want to...	Call this Row interface method...
Get the value of an attribute by name	`getAttribute()`
Set the value of an attribute by name	`setAttribute()`
Produce an XML document for a single row	`writeXML()`
Eagerly validate a row	`validate()`
Read row attribute values from XML	`readXML()`
Remove the row	`remove()`
Flag a newly created row as temporary (until updated again)	`setNewRowState(Row.STATUS_INITIALIZED)`
Retrieve the attribute structure definition information for a row	`getStructureDef()`
Get the Key object for a row	`getKey()`

In addition to generic `Row` access, JDeveloper 10g can generate you a custom *`YourViewObjectName`*`Row` interface containing your type-safe attribute getter and setter methods, as well as any desired row-level custom methods that you've chosen to expose to the client. You do this by visiting the **Client Row Interface** tab of the View Object editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list. JDeveloper will also generate an appropriate *`YourViewObjectName`*`RowClient` client proxy implementation class that is used automatically by your remote client in the case that you deploy your application module as an EJB Session Bean or whenever you use your application module in Batch Mode.

D.1.7 StructureDef Interface

A StructureDef is an interface that provides access to runtime metadata about the structure of a `Row`.

In addition, for convenience every `ViewObject` implements the `StructureDef` interface as well, providing access to metadata about the attributes in the resulting view rows that its query will produce.

Table D–7 StructureDef Interface

If you want to...	Call this StructureDef interface method...
Access attribute definitions for all attributes in the view object row	`getAttributeDefs()`
Find an attribute definition by name	`findAttributeDef()`
Get attribute definition by index	`getAttributeDef()`
Get number of attributes in a row	`getAttributeCount()`

D.1.8 AttributeDef Interface

An AttributeDef provides attribute definition information for any attribute of a View Object row or Entity Object instance like attribute name, Java type, and SQL type. It also provides access to custom attribute-specific metadata properties that can be inspected by generic code you write, as well as UI hints that can assist in rendering an appropriate user interface display for the attribute and its value.

Table D–8 AttributeDef Interface

If you want to...	Call this AttributeDef interface method...
Get the Java type of the attribute	`getJavaType()`
Get the SQL type of the attribute	`getSQLType()` **Note:** The `int` value corresponds to constants in the JDBC class `java.sql.Types`
Determine the kind of attribute	`getAttributeKind()` **Note:** If it's a simple attribute, it returns one of the constants `ATTR_PERSISTENT`, `ATTR_SQL_DERIVED`, `ATTR_TRANSIENT`, `ATTR_DYNAMIC`, `ATTR_ENTITY_DERIVED`. If it is an 1-to-1 or many-to-1 association/viewlink accessor it returns `ATTR_ASSOCIATED_ROW`. If it is an 1-to-many or many-to-many association/viewlink accessor it returns `ATTR_ASSOCIATED_ROWITERATOR`
Get the Java type of elements contained in an `Array`-valued attribute	`getElemJavaType()`
Get the SQL type of elements contained in an `Array`-valued attribute	`getElemSQLType()`
Get the name of the attribute	`getName()`
Get the index position of the attribute	`getIndex()`
Get the precision of a numeric attribute or the maximum length of a String attribute	`getPrecision()`
Get the scale of a numeric attribute	`getScale()`
Get the underlying column name corresponding to the attribute	`getColumnNameForQuery()`
Get attribute-specific custom property values	`getProperty()`, `getProperties()`
Get the UI `AttributeHints` object for the attribute	`getUIHelper()`

Table D–8 (Cont.) AttributeDef Interface

If you want to...	Call this AttributeDef interface method...
Test whether the attribute is mandatory	`isMandatory()`
Test whether the attribute is queriable	`isQueriable()`
Test whether the attribute is part of the primary key for the row	`isPrimaryKey()`

D.1.9 AttributeHints Interface

The AttributeHints interface related to an attribute exposes UI hint information that attribute that you can use to render an appropriate user interface display for the attribute and its value.

Table D–9 AttributeHints Interface

If you want to...	Call this AttributeHints interface method...
Get the UI label for the attribute	`getLabel()`
Get the tool tip for the attribute	`getTooltip()`
Get the formatted value of the attribute, using any format mask supplied	`getFormattedAttribute()`
Get the display hint for the attribute	`getDisplayHint()` **Note:** Will have a String value of either `Display` or `Hide`.
Get the preferred control type for the attribute	`getControlType()`
Parse a formatted string value using any format mask supplied for the attribute	`parseFormattedAttribute()`

D.2 Most Commonly Used Methods In the Business Service Tier

The implementation classes corresponding to the `oracle.jbo.*` interfaces described above are consciously designed to *not* be directly accessed by client code. They live in a different package named `oracle.jbo.server.*` and have the `Impl` suffix in their name to help remind you not to using them in your client-layer code.

In your business service tier implementation code, you can use any of the same methods that are available to clients above, but in addition you can also:

- Safely cast any `oracle.jbo.*` interface to its `oracle.jbo.server.*` package implementation class and use any methods on that `Impl` class as well.
- Override any of the base framework implementation class' `public` or `protected` methods to augment or change its default functionality by writing custom code in your component subclass before or after calling `super.`*`methodName`*`()`.

This section provides a summary of the most frequently called, written, and overridden methods for the key ADF Business Components classes.

D.2.1 Controlling Custom Java Files For Your Components

Before examining the specifics of individual classes, it's important to understand how you can control which custom Java files each of your components will use. When you don't need a customized subclass for a given component, you can just let the base framework class handle the implementation at runtime.

Each business component you create comprises a single XML component descriptor, and zero or more related custom Java implementation files. Each component that supports Java customization has a **Java** tab in its component editor in the JDeveloper 10g IDE. By checking or unchecking the different Java classes, you control which ones get created for your component. If none of the boxes is checked, then your component will be an XML-only component, which simply uses the base framework class as its Java implementation. Otherwise, tick the checkbox of the related Java classes for the current component that you need to customize. JDeveloper 10g will create you a custom *subclass* of the framework base class in which you can add your code.

> **Note:** You can setup global IDE preferences for which Java classes should be generated by default for each ADF business component type by selecting **Tools** | **Preferences...** | **Business Components** and ticking the checkboxes to indicate what you want your defaults to be.

A best practice is to *always* generate Entity Object and View Row classes, even if you don't require any custom code in them other than the automatically-generated getter and setter methods. These getter and setter methods offer you compile-time type checking that avoids discovering errors at runtime when you accidentally set an attribute to an incorrect kind of value.

D.2.2 ApplicationModuleImpl Class

The ApplicationModuleImpl class is the base class for application module components. Since the application module is the ADF component used to implement a business service, think of the application module class as the place where you can write your service-level application logic. The application module coordinates with view object instances to support updatable collections of value objects that are automatically "wired" to business domain objects. The business domain objects are implemented as ADF entity objects.

D.2.2.1 Methods You Typically Call on ApplicationModuleImpl

Table D–10 Methods You Typically Call on ApplicationModuleImpl

If you want to...	Call this method of the ApplicationModuleImpl class
Perform any of the common application module operations from inside your class, which can also be done from the client	See the Section D.1.1, "ApplicationModule Interface" section above.
Access a view object instance that you added to the application module's data model at design time	`get`*`ViewObjectInstanceName`*`()` **Note:** JDeveloper 10g generates this type-safe view object instance getter method for you to reflect each view object instance in the application module's design-time data-model.
Access the current DBTransaction object	`getDBTransaction()`

Table D–10 (Cont.) Methods You Typically Call on ApplicationModuleImpl

If you want to...	Call this method of the ApplicationModuleImpl class
Access a nested application module instance that you added to the application module at design time	`get`*`AppModuleInstanceName`*`()` **Note:** JDeveloper 10g generates this type-safe application module instance getter method for you to reflect each nested application module instance added to the current application module at design time.

D.2.2.2 Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass

Table D–11 Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass

If you want to...	Write a method like this in your custom ApplicationModuleImpl class
Invoke a database stored procedure	*`someCustomMethod`*`()` **Note:** Use appropriate method on the `DBTransaction` interface to create a JDBC `PreparedStatement`. If the stored procedure has `OUT` parameters, then create a `CallableStatement` instead. See this sample project for a robust code example of encapsulating a call to a PL/SQL stored procedure inside your application module.
Expose custom business service methods on your application module	*`someCustomMethod()`* **Note:** Select the method name on the **Client Interface** panel of the application module editor to expose it for client access if required.

JDeveloper 10g can generate you a custom *`YourApplicationModuleName`* interface containing service-level custom methods that you've chosen to expose to the client. You do this by visiting the **Client Interface** tab of the Application Module editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list.

D.2.2.3 Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

Table D–12 Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

If you want to...	Override this method of the ApplicationModuleImpl class
Perform custom setup code the first time an application module is created and each subsequent time it gets used by a different client session.	`prepareSession()` **Note:** This is the method you'd use to setup per-client context info for the current user in order to use database Oracle's Virtual Private Database (VPD) features. It can also be used to set other kinds of PL/SQL package global variables, whose values might be client-specific, on which other stored procedures might rely. This method is also useful to perform setup code that is specific to a given view object *instance* in the application module. If instead of being instance-specific you want the view object setup code to be initialized for every instance ever created of that view object component, then put the setup logic in an overridden `create()` method in your `ViewObjectImpl` subclass instead.
Perform custom setup code after the application module's transaction is associated with a database connection from the connection pool.	`afterConnect()` **Note:** Can be a useful place to write a line of code that uses `getDBTransaction().executeCommand()` to perform an `ALTER SESSION SET SQL TRACE TRUE` to enable database SQL Trace logging for the current application connection. These logs can then be processed with the TKPROF utility to study the SQL statements being performed and the query optimizer plans that are getting used.
Perform custom setup code before the application module's transaction releases its database connection back to the database connection pool.	`beforeDisconnect()` **Note:** If you have set `jbo.doconnectionpooling` to true, then the connection is released to the database connection pool each time the application module is returned to the application module pool.
Write custom application module state to the state management XML snapshot	`passivateState()`
Read and restore custom application module state from the state management XML snapshot	`activateState()`

D.2.3 DBTransactionImpl2 Class

The DBTransactionImpl2 class — which extends the base DBTransactionImpl class, and is constructed by the DatabaseTransactionFactory class — is the base class that implements the DBTransaction interface, representing the unit of pending work in the current transaction.

D.2.3.1 Methods You Typically Call on DBTransaction

Table D–13 Methods You Typically Call on DBTransaction

If you want to...	Call this method on the DBTransaction object
Commit the transaction	`commit()`
Rollback the transaction	`rollback()`
Eagerly validate any pending invalid changes in the transaction	`validate()`
Create a JDBC `PreparedStatement` using the transaction's Connection object	`createPreparedStatement()`
Create a JDBC `CallableStatement` using the transaction's Connection object	`createCallableStatement()`
Create a JDBC `Statement` using the transaction's Connection object	`createStatement()`
Add a warning to the transaction's warning list.	`addWarning()`

D.2.3.2 Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass

Table D–14 Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass

If you want to...	Override this method in your custom DBTransactionImpl2 class
Perform custom code before or after the transaction commit operation	`commit()`
Perform custom code before or after the transaction rollback operation	`rollback()`

In order to have your custom `DBTransactionImpl2` subclass get used at runtime, there are two steps you must follow:

1. Create a custom subclass of `DatabaseTransactionFactory` that overrides the create method to return an instance of your custom `DBTransactionImpl2` subclass like this:

```
package com.yourcompany.adfextensions;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.DatabaseTransactionFactory;
import com.yourcompany.adfextensions.CustomDBTransactionImpl;
public class CustomDatabaseTransactionFactory
       extends DatabaseTransactionFactory {
  /**
   * Return an instance of our custom CustomDBTransactionImpl class
   * instead of the default implementation.
   *
   * @return An instance of our custom DBTransactionImpl2 implementation.
   */
  public DBTransactionImpl2 create() {
    return new CustomDBTransactionImpl();
```

```
    }
}
```

2. Tell the framework to use your custom transaction factory class by setting the value of the `TransactionFactory` configuration property to the fully-qualified class name of your custom transaction factory. As with other configuration properties, if not supplied in the configuration XML file, it can be provided alternatively as a Java system parameter of the same name.

D.2.4 EntityImpl Class

The EntityImpl class is the base class for entity objects, which encapsulate the data, validation rules, and business behavior for your business domain objects.

D.2.4.1 Methods You Typically Call on EntityImpl

Table D–15 Methods You Typically Call on EntityImpl

If you want to...	Call this method in your EntityImpl subclass
Get the value of an attribute	`get`*`AttributeName`*`()` **Note:** Code-generated getter method calls `getAttributeInternal()` but provides compile-time type checking.
Set the value of an attribute	`set`*`AttributeName`*`()` **Note:** Code-generated setter method calls `setAttributeInternal()` but provides compile-time type checking.
Get the value of an attribute by name	`getAttributeInternal()`
Set the value of an attribute by name	`setAttributeInternal()`
Eagerly perform entity object validation	`validate()`
Refresh the entity from the database	`refresh()`
Populate the value of an attribute without *marking* it as being changed, but sending *notification* of its being changed so UI's refresh the value on the screen/page.	`populateAttributeAsChanged()`
Access the definition object for an entity	`getDefinitionObject()`
Get the Key object for an entity	`getKey()`
Determine the state of the entity instance, irrespective of whether it has already been posted in the current transaction (but not yet committed)	`getEntityState()` **Note:** Will return one of the constants `STATUS_UNMODIFIED`, `STATUS_INITIALIZED`, `STATUS_NEW`, `STATUS_MODIFIED`, `STATUS_DELETED`, or `STATUS_DEAD` indicating the status of the entity instance in the current transaction.

Table D–15 (Cont.) Methods You Typically Call on EntityImpl

If you want to...	Call this method in your EntityImpl subclass
Determine the state of the entity instance	`getPostState()` **Note:** This method is typically only relevant if you are programmatically using the `postChanges()` method to post but not yet commit entity changes to the database and need to detect the state of an entity with regard to its posting state
Get the value originally read from the database for a given attribute	`getPostedAttribute()`
Eagerly lock the database row for an entity instance	`lock()`

D.2.4.2 Methods You Typically Write in Your Custom EntityImpl Subclass

Table D–16 Methods You Typically Write in Your Custom EntityImpl Subclass

If you want to...	Write a method like this in your EntityImpl subclass
Perform attribute-specific validation	`public boolean validate`*`Something`*`(`*`AttrType`*`value)` **Note:** Register the attribute validator method by adding a "MethodValidator" on correct attribute in the **Validation** panel of the Entity Object editor. When you register the method validation
Perform entity-level validation	`public boolean validate`*`Something`*`()` **Note:** Register the entity-level validator method by adding a "MethodValidator" on the entity in the **Validation** panel of the Entity Object editor.
Calculate the value of a transient attribute	Add your calculation code to the generated `get`*`AttributeName`*`()` method.

D.2.4.3 Methods You Typically Override on EntityImpl

Table D–17 Methods You Typically Override on EntityImpl

If you want to...	Override this method in your custom EntityImpl subclass...
Set calculated default attribute values, including programmatically populating thbe primary key attribute value of a new entity instance.	`create()` **Note:** After calling `super.create()`, call the appropriate `set`*`AttrName`*`()` method(s) to set the default values for that(/those) attributes.

Table D–17 (Cont.) Methods You Typically Override on EntityImpl

If you want to...	Override this method in your custom EntityImpl subclass...
Modify attribute values before changes are posted to the database	`prepareForDML()`
Augment/change the standard INSERT, UPDATE, or DELETE DML operation that the framework will perform on your entity object's behalf to the database	`doDML()` **Note:** Check the value of the operation flag to the constants `DML_INSERT`, `DML_UPDATE`, or `DML_DELETE` to test what DML operation is being performed.
Perform complex, SQL-based validation after all entity instances have been posted to the database but before those changes are committed.	`beforeCommit()`
Insure that a related, newly-created, parent entity gets posted to the database *before* the current child entity on which it depends	`postChanges()` **Note:** If the parent entity is related to this child entity via a composition association, then the framework already handles this automatically. If they are only associated (but not composed) then you need to override `postChanges()` method to force a newly-created parent entity to post before the current, dependent child entity. See this OTN article for the code you typically write in your overridden `postChanges()` method to accomplish this.

Note: It is possible to write attribute-level validation code directly inside the appropriate `set`*`AttributeName`* method of your `EntityImpl` class, however adopting the `MethodValidator` approach suggested above results in having a single place on the **Validation** tab of the Entity Object editor to look in order to understand all of the validations in effect for your entity object, so it can result in easier to understand components.

WARNING: It is also possible to override the `validateEntity()` method to write entity-level validation code, however if you want to maintain the benefits of the ADF bundled exception mode — where the framework collects and reports a *maximal* set of validation errors back to the client user interface — it is recommended to adopt the `MethodValidator` approach suggested in the table above. This allows the framework to automatically collect all of your exceptions that your validation methods throw without your having to understand the bundled exception implementation mechanism. Overriding the `validateEntity()` method directly shifts the responsibility on your *own* code to correctly catch and bundle the exceptions like Oracle ADF would have done by default, which is non-trivial and a chore to remember and hand-code each time.

D.2.5 EntityDefImpl Class

The EntityDefImpl class is a singleton, shared metadata object for all entity objects of a given type in a single Java VM. It defines the structure of the entity instances and provides methods to create new entity instances and find existing instances by their primary key.

D.2.5.1 Methods You Typically Call on EntityDefImpl

Table D–18 Methods You Typically Call on EntityDefImpl

If you want to...	Call the EntityDefImpl method
Find an entity object of a this type by its primary key	`findByPrimaryKey()` **Note:** See this tip for getting `findByPrimaryKey()` to find entity instances of subtype entities as well.
Access the current `DBTransaction` object	`getDBTransaction()`
Find any `EntityDefImpl` object by its fully-qualified name	`findDefObject()` (static method)
Retrieve the value of an entity object's custom property	`getProperty(),getProperties()`
Set the value of an entity object's custom property	`setProperty()`
Create a new instance of an entity object	`createInstance2()` **Note:** Alternatively, you can expose custom `create`*`XXX`*`()` methods with your own expected signatures in that same custom `EntityDefImpl` subclass. See the next section for details.
Iterate over the entity instances in the cache of this entity type.	`getAllEntityInstancesIterator()`
Access `ArrayList` of entity definition objects for entities that extend the current one.	`getExtendedDefObjects()`

D.2.5.2 Methods You Typically Write on EntityDefImpl

Table D–19 Methods You Typically Write on EntityDefImpl

If you want to...	Write a method like this in your custom EntityDefImpl class
Allow other classes to create an entity instance with an initial type-safe set of attribute values or setup information.	`createXXXX(Type1arg1, ..., TypeNargN)` **Note:** Internally, this would create and populate an instance of a `NameValuePairs` object (which implements AttributeList) and call the protected method createInstance(), passing that `NameValuePairs` object. Make sure the method is `public` if other classes need to be able to call it.

D.2.5.3 Methods You Typically Override on EntityDefImpl

Table D–20 Methods You Typically Override on EntityDefImpl

If you want to...	Call the EntityDefImpl method
Perform custom metadata initialization when this singleton metaobject is loaded.	`createDef()`
Avoid using the `RETURNING INTO` clause to support refresh-on-insert or refresh-on-update attributes	`isUseReturningClause()` **Note:** Return false to disable the use of `RETURNING INTO`, necessary sometimes when your entity object is based on a view with `INSTEAD OF` triggers that doesn't support `RETURNING INTO` at the database level.
Control whether the UPDATE statements issued for this entity update only changed columns, or all columns	`isUpdateChangedColumns()` **Note:** Defaults to `true`.
Find any EntityDefImpl object by its fully-qualified name	`findDefObject()` **Note:** Static method.
Set the value of an entity object's custom property	`setProperty()`
Allow other classes to create a new instance an entity object without doing so implicitly via a view object.	`createInstance()` **Note:** If you don't write a custom create method as noted in the previous section, you'll need to override this method and widen the visibility from `protected` to `public` to allow other classes to construct an entity instance.

D.2.6 ViewObjectImpl Class

The ViewObjectImpl class the base class for view objects.

D.2.6.1 Methods You Typically Call on ViewObjectImpl

Table D–21 Methods You Typically Call on ViewObjectImpl

If you want to...	Call this ViewObjectImpl method
Perform any of the common view object, rowset, or rowset iterator operations from inside your class, which can also be done from the client	See the Section D.1.3, "ViewObject Interface", Section D.1.4, "RowSet Interface", and Section D.1.5, "RowSetIterator Interface" sections above.
Set an additional runtime WHERE clause on the default rowset	`setWhereClause()`
Defines a named bind parameter.	`defineNamedWhereClauseParam()`
Removes a named bind parameter.	`removeNamedWhereClauseParam()`
Set bind variable values on the default rowset by name. Only works when you have formally defined named bind variables on your view object.	`setNamedWhereClauseParam()`
Set bind variable values on the default rowset. Use this method for view objects with binding style of "Oracle Positional" or "JDBC Positional" when you have not formally defined named bind variables.	`setWhereClauseParam()`
Retrieved a subset of rows in a view object's row set based on evaluating an in-memory filter expression.	`getFilteredRows()`
Retrieved a subset of rows in the current range of a view object's row set based on evaluating an in-memory filter expression.	`getFilteredRowsInRange()`
Set the number of rows that will be fetched from the database per round-trip for this view object.	`setFetchSize()` **Note:** The default fetch size is a single row at a time. This is definitely not optimal if your view object intends to retrieve many rows, so you should either set the fetch size higher at design time on the **Tuning** tab of the View Object editor, or set it at runtime using this API.

D.2.6.2 Methods You Typically Write in Your Custom ViewObjectImpl Subclass

Table D–22 Methods You Typically Write in Your Custom ViewObjectImpl Subclass

If you want to...	Write a method like this in your ViewObjectImpl subclass
Provide clients with type-safe methods to set bind variable values without exposing positional details of the bind variables themselves	`someMethodName(Type1arg1, ..., TypeNargN)` **Note:** Internally, this method would call the setWhereClauseParam() API to set the correct bind variables with the values provided in the type-safe method arguments.

JDeveloper 10g can generate you a custom `YourViewObjectName` interface containing view object custom methods that you've chosen to expose to the client. You can accomplish this by visiting the **Client Interface** tab of the View Object editor, and

shuttling the methods you'd like to appear in your client interface into the **Selected** list.

D.2.6.3 Methods You Typically Override in Your Custom ViewObjectImpl Subclass

Table D–23 Methods You Typically Override in Your Custom ViewObjectImpl Subclass

If you want to...	Override this ViewObjectImpl method
Initialize custom view object class members (not row attributes!) when the view object instance is created for the first time.	`create()` **Note:** This method is useful to perform setup logic that is applicable to every instance of a view object that will ever get created, in the context of any application module. If instead of *generic* view object setup logic, you need to perform logic specific to a given view object *instance* in an application module, then override the `prepareSession()` method of your application module's `ApplicationModuleImpl` subclass and perform the logic there after calling `findViewObject()` to find the view object instance whose properties you want to set.
Write custom view object instance state to the state management XML snapshot	`passivateState()`
Read and restore custom view object instance state from the state management XML snapshot	`activateState()`
Customize the behavior of view object query execution, independent of whether the query was executed explicitly by calling `executeQuery()` or implicitly, for example, by navigating to the `first()` row when the query hadn't yet been executed.	`executeQueryForCollection()`
Change/augment the way that the `ViewCriteria` collection of `ViewCriteriaRows` is converted into a query-by-example `WHERE` clause.	`getViewCriteriaClause()`

D.2.7 ViewRowImpl Class

The ViewRowImpl class the base class for view row objects.

D.2.7.1 Methods You Typically Call on ViewRowImpl

Table D–24 Methods You Typically Call on ViewRowImpl

If you want to...	Write a method like this in your custom ViewRowImpl class
Perform any of the common view row operations from inside your class, which can also be done from the client	See the Section D.1.6, "Row Interface" section above.
Get the value of an attribute	`get`*`AttrName`*`()`
Set the value of an attribute	`set`*`AttrName`*`()`

Table D–24 (Cont.) Methods You Typically Call on ViewRowImpl

If you want to...	Write a method like this in your custom ViewRowImpl class
Access the underlying entity instance to which this view row is delegating attribute storage.	`get`*`EntityUsageAliasName`*`()` **Note:** You can change the name of the entity usage alias name on the **Entity Objects** tab of the View Object Editor

D.2.7.2 Methods You Typically Write on ViewRowImpl

Table D–25 Methods You Typically Write on ViewRowImpl

If you want to...	Write a method like this in your custom ViewRowImpl class
Calculate the value of a view object level transient attribute	`get`*`AttrName`*`()` **Note:** JDeveloper generates the skeleton of the method for you, but you need to write the custom calculation logic inside the method body.
Perform custom processing of the setting of a view row attribute	`set`*`AttrName`*`()` **Note:** JDeveloper generates the skeleton of the method for you, but you need to write the custom logic inside the method body if required.
Determine the updateability of an attribute in a conditional way.	`isAttributeUpdateable()`
Custom methods that expose logical operations on the current row, optionally callable by clients	`doSomething()` **Note:** Often these view-row level custom methods simply turn around and delegate to a method call on the underlying entity object related to the current row.

JDeveloper 10g can generate you a custom *`YourViewObjectName`*`Row` interface containing view row custom methods that you've chosen to expose to the client. You can accomplish this by visiting the **Client Row Interface** tab of the View Object editor, and shuttling the methods you'd like to appear in your client interface into the **Selected** list.

D.2.7.3 Methods You Typically Override in Your Custom ViewRowImpl Subclass

Table D–26 Methods You Typically Override in Your Custom ViewRowImpl Subclass

If you want to...	Write a method like this in your custom ViewRowImpl class
Determine the updateability of an attribute in a conditional way.	`isAttributeUpdateable()`

D.2.8 Setting Up Your Own Layer of Framework Base Classes

Before you begin to develop application specific business components, Oracle recommends creating yourself a layer of classes that extend all of the ADF Business Components framework base implementation classes described in this paper. An example of a customized framework base class for application module components might look like this:

```
package com.yourcompany.adfextensions;
import oracle.jbo.server.ApplicationModuleImpl;
public class CustomApplicationModuleImpl extends ApplicationModuleImpl {
  /*
   * We might not yet have any custom code to put here yet, but
   * the first time we need to add a generic feature that all of
   * our company's application modules need, we will be very happy
   * that we thought ahead to leave ourselves a convenient place
   * in our class hierarchy to add it so that all of the application
   * modules we have created will instantly benefit by that new feature,
   * behavior change, or even perhaps, bug workaround.
   */
}
```

A common set of customized framework base classes in a package name of your own choosing like *`com.yourcompany.adfextensions`*, each importing the `oracle.jbo.server.*` package, would consist of the following classes.

- `public class CustomEntityImpl extends EntityImpl`
- `public class CustomEntityDefImpl extends EntityDefImpl`
- `public class CustomViewObjectImpl extends ViewObjectImpl`
- `public class CustomViewRowImpl extends ViewRowImpl`
- `public class CustomApplicationModuleImpl extends ApplicationModuleImpl`
- `public class CustomDBTransactionImpl extends DBTransactionImpl2`
- `public class CustomDatabaseTransactionFactory extends DatabaseTransactionFactory`

For completeness, you may also want to create customized framework classes for the following classes as well, but overriding anything in these classes would be a fairly rare requirement.

- `public class CustomViewDefImpl extends ViewDefImpl`
- `public class CustomEntityCache extends EntityCache`
- `public class CustomApplicationModuleDefImpl extends ApplicationModuleDefImpl`

E

ADF Business Components J2EE Design Pattern Catalog

This appendix provides a brief summary of the popular J2EE design patterns that the ADF Business Components layer implements for you.

This appendix contains the following sections:

- Appendix E, "ADF Business Components J2EE Design Pattern Catalog"

E.1 J2EE Design Patterns Implemented by ADF Business Components

By using the Oracle Application Development Framework's business components building-blocks and related design-time extensions to JDeveloper, you get a prescriptive architecture for building richly-functional and cleanly layered J2EE business services with great performance. Table E–1 provides a brief overview of the numerous design patterns that the ADF Business Components layer implements for you. Some are the familiar patterns from Sun's J2EE Blueprints, and some are design patterns that ADF Business Components adds to the list.

Table E–1 J2EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF BC Implements It
Model/View/Controller Cleanly separates the roles of data and presentation, allowing multiple types of client displays to work with the same business information.	The ADF Application Module provides a generic implementation of a Model/View/Controller "application object" that simplifies exposing the application data model for any application or service, and facilitates declaratively specifying the boundaries of a logical unit of work. Additional UI-centric frameworks and tag libraries provided in JDeveloper 10g help the developer implement the View and Controller layers.
Interface / Implementation Separation Cleanly separates the API or Interface for components from their implementation class.	ADF Business Components enforce a logical separation of client-tier accessible functionality (via interfaces) and their business tier implementation. JDeveloper handles the creation of custom interfaces and client proxy classes automatically.

Table E–1 (Cont.) J2EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF BC Implements It
Service Locator Abstracts the technical details of locating a service so the client and use it more easily.	ADF application modules are looked up using a simple configuration object which hides the low-level details of finding the service instance behind the scenes. For web applications, it also hides the implementation of the application module pool usage, a lightweight pool of service components that improves application scalability.
Inversion of Control A containing component orchestrates the lifecycle of the components it contains, invoking specific methods that the developer can overrides at the appropriate times so the developer can focus more on what the code should do instead when it should get executed.	ADF components contain a number of easy-to-override methods that the framework invokes as needed during the course of application processing.
Dependency Injection Simplifies application code, and increases configuration flexibility by deferring component configuration and assembly to the container.	The ADF configures all its components from externalized XML metadata definition files. The framework automatically injects dependent objects like view object instances into your application module service component and entity objects into your view rows at runtime, implementing lazy loading. It supports runtime factory substitution of components by any customized subclass of that component to simplify on-site application customization scenarios. Much of the ADF functionality is implemented via dynamic injection of validator and listener subscriptions that coordinate the framework interactions depending on what declarative features have been configured for each component in their XML metadata.
Active Record Avoids the complexity of "anything to anything" object/relational mapping, by providing an object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.	ADF entity objects handle the database mapping functionality you use most frequently, including inheritance, association, and composition support, without having to think about object/relational mapping. They also provide a place to encapsulate both declarative business rules and one-off programmatic business domain logic.
Data Access Objects Avoids unnecessary marshalling overhead by implementing dependent objects as lightweight, persistent classes instead of each as an Enterprise Bean. Isolates persistence details into a single, easy to maintain class.	ADF view objects automate the implementation of data access for reading data using SQL statements. ADF entity objects automate persistent storage of lightweight business entities. ADF view objects and entity objects cooperate to provide a sophisticated, performant data access objects layer where any data queried through a view object can optionally be made fully updatable without writing any "application plumbing" code.
Session Facade Avoids inefficient client access of Entity Beans and inadvertent exposure of sensitive business information by wrapping Entity Beans with a Session Bean.	ADF application modules are designed to implement a coarse-grained "service facade" architecture in any of their supported deployment modes. When deployed as EJB Session Beans, they provide an implementation the Session Facade pattern automatically.

Table E–1 (Cont.) J2EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF BC Implements It
Value Object Avoids unnecessary network round-trips by creating one-off "transport" objects to group a set of related attributes needed by a client program.	ADF provides an implementation of a generic Row object, which is a metadata-driven container of any number and kind of attributes that need to be accessed by a client. The developer can work with the generic `Row` interface and do late-bound `getAttribute("Price")` and `setAttribute("Quantity")`calls, or optionally generate early-bound row interfaces like `OverdueOrdersRow`, to enable type-safe method calls like `getPrice()` and `setQuantity()`. Smarter than just a simple "bag 'o attributes" the ADF `Row` object can be introspected at runtime to describe the number, names, and types of the attributes in the row, enabling sophisticated, generic solutions to be implemented.
Page-by-Page Iterator Avoids sending unnecessary data to the client by breaking a large collection into page-sized "chunks" for display.	ADF provides an implementation of a generic RowSet interface which manages result sets produced by executing View Object SQL queries. RowSet allows the developer to set a desired page-size, for example 10 rows, and page up and down through the query results in these page-sized chunks. Since data is retrieved lazily, only data the user actually visits will ever be retrieved from the database on the backend, and in the client tier the number of rows in the page can be returned over the network in a single roundtrip.
Fast-Lane Reader Avoids unnecessary overhead for read-only data by accessing JDBC API's directly. This allows an application to retrieve only the attributes that need to be displayed, instead of finding all of the attributes by primary key when only a few attributes are required by the client. Typically, implementations of this pattern sacrifice data consistency for performance, since queries performed at the raw JDBC level do not "see" pending changes made to business information represented by Enterprise Beans.	ADF View Objects read data directly from the database for best performance, however they give developers a choice regarding data consistency. If updateability and/or consistency with pending changes is desired, the developer need only associate his/hew View Object with the appropriate Entity Objects whose business data is being presented. If consistency is not a concern, View Objects can simply perform the query with no additional overhead. In either case, the developer never has to write JDBC data access code. They only provide appropriate SQL statements in XML descriptors.
(Bean) Factory Allows runtime instantiation and configuration of an appropriate subclass of a given interface or superclass based on externally-configurable information.	All ADF component instantiation is done based on XML configuration metadata through factory classes allowing runtime substitution of specialized components to facilitate application customization.
Entity Facade Provides a restricted view of data and behavior of one or more business entities.	The ADF view object can surface any set of attributes and methods from any combination of one or more underlying entity objects to furnish the client with a single, logical value object to work with.

Table E–1 (Cont.) J2EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF BC Implements It
Value Messenger Keeps client value object attributes in sync with the middle-tier business entity information that they represent in a bidirectional fashion.	ADF's value object implementation coordinates with a client-side value object cache to batch attribute changes to the EJB tier and receive batch attribute updates which occur as a result of middle-tier business logic. The ADF Value Messenger implementation is designed to not require any kind of asynchronous messaging to achieve this effect.
Continuations Gives the developer the simplicity and productivity of a stateful programming model with the scalability of a stateless web solution.	ADF's application module pooling and state management functionality combine to deliver this developer value-add. This avoids dedicating application server tier resources to individual users, but supports a "stateless with user affinity" optimization that you can tune.

Index

Numerics

4GL development
  about, 1-4
  ADF Business Components and, 4-4

A

access keys, 11-40
accessibility support levels, 11-39
`accessorIterator` element, A-11
`Accessors` element, 15-15
Action Binding Editor, 17-5
action bindings
  about, 12-18
  debugging, 24-20
  `disabled` attribute, 12-25, 13-16
  `enabled` property, 12-26, 13-15
  for operations, 13-14
  for page navigation, 16-17
`action` element, A-14
action events, 13-16
action listeners
  in navigation operations, 13-16
  in page navigation components, 16-19, 16-20
action methods, in page navigation components, 16-16
`actionEnabled` binding property, B-1
`actionListener` attribute
  command buttons for methods, 17-6
  navigation operations, 13-16
  See also action listeners
actions
  adding ADF bindings to existing, 21-11
  in page navigation, 16-14
`actions` facet, 19-44
active data models
  about, 4-17
  benefits of, 4-18
  displayed in Data Control Palette, 10-6
  examples of, 4-17
  view link instances and view link accessors, 27-4
Add Validation Rule dialog, 6-27
ADF. *See* Oracle ADF
ADF binding context. *See* binding context
ADF binding filter. *See* binding filter
ADF bindings. *See* bindings
ADF Business Components
  4GL tools and, 4-4 to 4-8
  application module pooling. *See* pooling
  `ApplicationModule` interface, 4-11
  `ApplicationModuleImpl` class, 4-11
  archive deployment profiles for, 25-22
  checklist for basing components on framework extensions, 25-3
  client interfaces for, 4-16
  configuration properties, listing, 29-4
  configuration property scopes, 29-4
  current JDBC connection, accessing, 25-21
  custom interfaces, creating, 25-14
  custom Java classes
    about, 4-12
    for entity objects, generating, 6-38
  custom metadata properties, 25-12
  customized framework behavior, examples of, 25-10
  data types in `oracle.jbo.domain` package, 4-14
  data types supported, 4-20
  debug diagnostics output for, enabling, 5-17
  enabling security, 30-12
  error messages, customizing, 25-25
  extending base classes, 25-2
  features of, 4-2
  Forms concepts and, 4-4
  framework extension classes
    about, 25-2
    basing components on, 25-4
    creating, 25-2
    creating a layer of, 25-8
    custom Java classes, reflected in, 25-6
    entries in XML component definition file, 25-4, 25-5
    for database transactions, 25-27
    library definitions for, creating, 25-9
    packaging in JAR files, 25-9
  inheritance, using to extend individual components, 25-29
  JDeveloper design time preferences for, 4-13
  `oracle.jbo` package, 4-11
  `oracle.jbo.server` package, 4-11
  overview of, 4-1, 4-8

package naming conventions, 4-9
publishing as web services, 33-14
reusable component libraries, creating, 25-22
SQL flavors supported, 4-19
See also application modules
See also state management
ADF Business Components applications
business domain layer, creating, 2-5
business services for use cases, implementing, 2-9
database constraint error handling, customizing, 25-27
database constraint error messages, customizing, 25-27
metadata files for bindings, 2-21
overview of creating, 2-2
production, customizing on-site, 25-35
UI first development approach, 2-5
web pages, creating, 2-18
ADF Business Components projects
adding components from other directories, 25-24
component substitutions, defining global, 25-35
custom message bundles, adding, 25-26
custom validation rules, registering and using, 26-34
debug output, enabling, 5-17
default base framework extension classes, specifying preferences for, 25-7
disabling use of package XML files, 4-14
removing imported packages, 25-25
reusable component libraries
creating, 25-22
importing, 25-23
reusable framework extension classes, packaging, 25-8
ADF Command Button. *See* command buttons
ADF Controller library, 12-8
ADF Creation forms, 13-21
ADF Faces
accessibility support levels, 11-39
configuration files, A-3
converters, 20-17
dependencies and libraries, 11-14
enhanced debugging output, 11-39
file uploading, 19-47
filter and mapping configuration settings, 11-18
internationalization, 22-10
partial page rendering, 19-33
resource servlet and mapping configuration settings, 11-18
skins, 22-3
supported platforms, 11-3
tag libraries for, 11-14
validation, 20-4
validators, 20-6
ADF Faces Cache
AFC Statistics servlet, 23-7
logging, 23-6
types of content to cache, 23-2
visual diagnostics, 23-8
ADF Faces components
access keys for, 11-40
adding bindings to existing, 21-3
adding to JSF pages, 11-15
changing appearance, 22-3
creating
`commandButton` components, 17-5, 17-6
`commandButton` components for page navigation, 16-15, 16-17
`commandButton` components, for navigation operations, 13-13
`commandLink` components for page navigation, 16-15, 16-17
`commandLink` components, for navigation operations, 13-13
`inputText` components, 13-3
`outputText` components, 13-3
`selectManyShuttle` components, 19-65
`selectOneRadio` components, 19-55
`selectRangeChoiceBar` components, 14-7
`table` components, 14-3
`tableSelectMany` components, 14-18
`tableSelectOne` components, 14-4
`tree` components, 15-9
`treeTable` components, 15-17
layout and panel components, 11-29
skinning, 22-3
style properties, changing, 22-2
translating, 22-10
ADF Faces Core tag library, 11-14
ADF Faces HTML tag library, 11-14
ADF Faces lifecycle, overriding, 20-25
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, 24-7
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
benefits of JSF and, 1-8
business service technologies and, 1-3
declarative data binding and, 1-8, 10-1
exception handling, 20-23
lifecycle, 13-6
runtime, what happens at, 2-22
validation, 20-8
ADF Model Runtime library, 12-8
ADF Output Text. *See* text fields
ADF Output Text with a Label. *See* text fields
ADF phase listener, 13-6
creating custom, 20-26
registering in the `web.xml` file, 12-7

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, 34-21
  ADF Controller library, 12-8
  ADF Model Runtime library, 12-8
  adf-controller.jar file, 12-8
  ADFm.jar file, 12-8
  deleting, 34-29
  in the project properties, 12-8
  installing
    from JDeveloper, 34-18
    manually, 34-26
    on third-party servers, 34-18
  list of files, 34-26
ADF Swing, 1-2
ADF Table. *See* tables
ADF Tree. *See* tree components
ADF Tree Table. *See* treeTable components
ADFBindingFilter class, 12-10, 12-11
adf-config.xml file, 30-25
ADFContext object, 12-12
adf-controller.jar file, 12-8
adf-faces-config.xml file
  about, A-3, A-32
  accessibility, A-33
  client-side validation, 20-4, A-34
  client-validation-disabled element, 20-4
  conversion, 20-17
  currency, numbers, and decimals, A-33
  currency-code element, 22-18
  custom upload file processor, A-36
  decimal-separator element, 22-18
  editing, 11-20
  enhanced debugging output, A-34
  example of, 11-20
  help site URL, A-36
  language reading direction, A-34
  localization properties, 22-18
  number-grouping-separator element, 22-18
  output mode, A-35
  ProcessScope instances, A-35
  retrieving property values, A-36
  right-to-left element, 22-18
  skin family, A-35
  skin-family element, 22-9
  supported tasks, A-33
  time zone and year offset, A-35
  time-zone element, 22-18
adfFacesContext.postback property, 10-20
adf-faces-skins.xml file, 22-8
  about, A-37
  bundle-name element, 22-9
  family element, 22-8
  id element, 22-8
  render-kit-id element, 22-9
  skins element, 22-8
  style-sheet-name element, 22-9
  supported tasks, A-37
adfm.jar file, 12-8
AFC Statistics servlet, 23-7
AJAX-style pages, 1-16
allDetailsEnabled attribute, 14-13
allRowsInRange binding property, B-1
Application Module Editor, 5-11
application module pools
  about, 29-9
  See also pooling
application modules
  benefits of custom methods, 4-18
  bundled exception mode, 10-23
  client interfaces
    accessing in ADF Swing panels, 8-19
    accessing in custom page controllers, 8-19
    accessing in JSF backing beans, 8-17
    generating, 8-11
    local mode versus remote mode, 8-16
    programmatic access, 8-14
  client proxy classes, 8-12
  command-line test client programs for, creating, 5-19
  compared to 4GL concepts, 4-5 to 4-8
  creating, 5-7
  creation guidelines, 8-28
  custom database transaction class, configuring to use, 25-28
  custom service methods
    adding, 8-7, 8-9
    displayed in Data Control Palette, 10-5
    guidelines for, 8-9
    method signatures, 8-13
    publishing, 8-11
    testing, 6-36
  database connections
    configuring, 8-3
    types of, 8-4
  database user state, initializing, 29-18
  debugging, 8-9
  definition of, 4-3, 8-2
  editing, 5-11
  extending, 25-30, 25-34
  nested, 10-9
  operations, built-in, 10-7
  pooling and state management features of, 8-26
  publishing as web services, 33-14
  root versus nested, 8-30
  runtime configuration properties, default, 5-11
  SQL tracing, enabling, 27-15
  testing activation, 28-20
  testing tool for, 5-14
  tips and techniques for data binding, 10-12 to 10-20
  Transaction object, 7-26
  types of states in pools, 29-2
  UML diagrams, creating, 8-23
  view objects in, using, 5-6
  XML component definition files for, 5-10
  See also data models
  See also pooling

See also state management
application templates, 11-3
application view caching, 11-40
`ApplicationModule` interface, 4-11
`ApplicationModuleImpl` class
about, 4-11
built-in methods, overriding, 8-19
Apply Request Values phase, 13-8
architecture of Oracle ADF, 1-1
associations
accessor row set retention feature, using, 26-9
accessors to access related entity rows, 9-15
complex, implementing more, 26-8
composition
posting order and, 26-24
settings for, 6-25
styles of relationships, 6-11
creating manually, 6-8
database tables with no foreign key constraints and, 6-8
DBSequence-valued primary keys and, 26-27
editing accessor names, 6-10
names of, 6-5
posting order, controlling, 26-24
programmatic access, 6-30
refactoring, 6-10
XML component definition files for, 6-5, 6-11
See also entity objects
attribute bindings
about, 12-18, 13-4
EL expressions for, 13-5
setting ADF authorization grants, 30-27
attribute control hints
adding, 5-12, 6-15
Java message bundles and, 5-13
structure in Java message bundles, 5-13
`attributeDef` binding property, B-1
`attributeDefs` binding property, B-1
attribute-level validation rules, 9-3
attributes
about, 12-5
binding to text fields, 13-2
on the Data Control Palette, 12-5
Attributes Mapping wizard page, 5-4
`attributeValue` binding property, B-1
`attributeValues` bindings property, B-2
`attributeValues` element, A-14
`AttrNames` element, 15-15
authenticated users, referencing information about, 9-16
authentication
enabling ADF Security, 30-9
enabling for ADF Business Components applications, 30-12
enabling J2EE security, 30-4
authorization
ADF Security permissions, 30-24
enabling for ADF Security, 30-23
`authorizationEnforce` property, 30-14, 30-25
automatic component binding, 11-33, 16-16, 16-17
automatic form submission, 11-39
`autoSubmit` attribute
for table row selection components, 14-15
use of, 11-39

B

Back button, issues
in forms, 13-17
in page navigation, 16-11
in tables, 14-9
backing beans
ADF data controls and, 11-37
application module client interfaces, accessing, 8-17
automatic component binding and, 11-34
`binding` attribute and, 11-33
definition of, 1-7
overriding declarative methods in, 17-8
referencing in JSF pages, 11-33
registering in `faces-config.xml`, 11-31
state management release level, setting, 28-9
uses of, 11-30
using for validation method, 20-12
using in page navigation components, 16-16
`bc4j.xcfg` file, 5-11
Bind Action Property dialog, 16-17, 17-9
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, 20-12
bind variables. *See* named bind variables
`binding` attribute, 11-33
binding containers
about, 12-19
debugging, 24-16
overriding declarative methods, 17-13
runtime usage, 12-19
scope, 12-19
setting ADF authorization grants, 30-26
binding context
about, 12-8
initializing using the ADF binding filter, 12-12
binding context, debugging, 24-12
binding filter, 12-7, 12-10
binding objects
action, 12-18
attribute, 12-18
defined in page definition files, 12-13
EL properties of, B-1
`invokeAction`, 12-15

iterator, 12-15
list, 12-18, 19-58, 19-62, 19-64
method action, 12-18
referencing in EL expressions, 12-20
runtime properties, 12-27
scope, 12-19
table, 12-18
tree, 12-18
value, 12-18
binding properties
accessing in the Expression Builder, 12-21
EL reference, B-1
in EL expressions, 12-20
binding to existing components
`commandButton` components, 21-12
`commandLink` components, 21-12
`inputText` components, 21-8
`outputText` components, 21-7
selection lists, 21-12
`table` components, 21-9
`tree` components, 21-15
bindings
action for operations, 13-14
adding to existing components, 21-1
adding to UI components, 13-25
attribute, 13-4
changing, 13-26
changing for tables, 14-10
deleting for UI components, 13-25
iterator, about, 13-3
rebinding tables, 14-11
rebinding UI components, 13-26
required objects for ADF, 12-7
table, 14-5
text fields, 13-2
value, 13-4
`bindings` element, 12-18
`bindings` variable, 12-20
breakpoints
types of, 24-8
bundled exception mode, 10-23
`bundle-name` element, 22-9
Business Component Browser Connect dialog, 5-14
Business Component View Criteria dialog, 5-25
Business Components Browser
application module state management feature, using, 8-27
application modules, testing, 5-14
debug diagnostic output, example of, 5-17
features of, 5-16
illustration of, 5-15
named bind variables, inspecting, 5-28
read-only view objects, testing, 5-16
toolbar buttons, 7-16
view criteria, testing, 5-25
Business Components from Tables wizard, 6-2
business rules, programmatic, 9-1
business service clients, definition of, 4-11
business service methods
adding, 8-6
displayed in Data Control Palette, 10-5
guidelines for, 8-9
method signatures, 8-13
publishing, 8-11
testing, 6-36
business services
web services, 33-1
`button` element, A-15
buttons, command. *See* command buttons

C

caching with ADF Faces Cache, 23-1 to 23-8
calculated attributes, 7-21
automating recalculation, 26-29
entries in XML component definition file, 7-22, 7-23
character encoding, in the ADF binding filter, 12-12
`children` binding property, B-2
client-side state saving, 11-38
client-side validation
creating custom JSF, 20-15
using custom JSF, 20-15
`client-validation-disabled` element, 20-4
`CollectionModel` class, 14-6
collections, about, 12-3, 12-5
columns
attributes for, 14-6
`column` tag, 14-5
command buttons
adding ADF bindings to existing, 21-11
binding to backing beans, 16-16
binding to methods, 17-6
creating using a method, 17-5
in navigation operations, 13-13
in page navigation, 16-15, 16-17
command components
ID for, 17-7
passing parameter values with, 17-7
command links
in page navigation, 16-15, 16-17
navigation operations, 13-13
setting current row with, 14-23
`commandButton` components. *See* command buttons
`commandLink` components. *See* command links
Commit operation, 13-18
compilation errors, 24-2
components. *See* UI components
conditionally displaying components, 18-14
configuration files for JSF
creating page navigation rules in, 16-2
editing, 11-7
starter file in JDeveloper, 11-7
using multiple, 11-9
wizard for creating, 11-9
See also `faces-config.xml` file
Configuration Manager dialog, 5-11
connection pooling
pending database state and, 28-22
`Context` class, 13-7

control bindings, 10-2
control hints
  adding, 5-12, 6-15
  Java message bundles and, 5-13
  structure in Java message bundles, 5-13
conversion
  about, 20-16
  in an application, 20-2
  lifecycle, 20-2
converters
  ADF Faces, 20-17
  creating custom, 20-19
  creating custom JSF, 20-19
  using, 20-17
.cpx file. *See* DataBindings.cpx file
Create Application dialog, 11-3
Create Application Module wizard, 5-7
Create Business Components Diagram dialog, 6-12, 8-23
Create Cascading Style Sheet dialog, 22-6
Create Entity Object wizard, 6-6
Create JSF JSP wizard, 11-10
Create Managed Bean dialog, 11-31, 17-2
Create New Association wizard, 6-8
Create operation
  about, 13-20
  repeating, 13-23
Create View Link wizard, 5-36
Create View Object wizard, 5-2
Create Web Service Data Control wizard, 33-4
createPageLifecycle method, 20-26
createRootApplicationModule() method, 8-16
createRowSet() method, 27-12
createRowSetIterator() method, 27-12
creating entity objects in, 26-19
CSS style properties, 22-2
currency-code element, 22-18
current row, setting programmatically, 14-23
currentRow binding property, B-2
custom Java classes
  about, 4-12
  for entity objects, generating, 6-38
custom service methods
  adding, 8-6
  displayed in Data Control Palette, 10-5
  guidelines for, 8-9
  method signatures, 8-13
  publishing, 8-11
  testing, 6-36
custom validation rules
  creating, 26-31
  customizers for, 26-33
  registering and using in projects, 26-34
CVS
  client, 32-1
  commit comments, 32-2
  preferences, 32-1

D

data binding files
  about, A-3
data binding, about, 1-8
data collection. *See* collections
data control files
  about, A-2
Data Control Palette
  about, 12-2
  attributes, 12-5
  context menu, 12-6
  default UI component features, 12-7
  displaying, 12-2
  icons defined, 12-3
  identifying master-detail objects, 15-2
  method returns, 12-4
  objects created, 12-7
  operations, 12-5
  parameters, 12-5
  using to create UI components, 12-6
  view object collections, 12-5
Data Control Security wizard, 33-9
data control security, defining for web services, 33-9
data controls
  application module, role of, 10-3
  benefits of, 1-8
  default names, changing, 10-4
  displayed on the Data Control Palette, 12-3
  from web services, 33-4
  using to create UI components, 12-1
data models
  active master-detail coordination, enabling, 5-41
  displayed in Data Control Palette, 10-6
  multiple master view objects, setting, 27-10
  updatable
    creating, 7-1
    testing, 7-16
  view link instances and view link accessors, 27-4
  view object instances in, defining, 5-8
  view objects and, 5-6
data types
  domains, 26-2
  Oracle, list of, 4-14
  setting usage of, 4-20
database connection pools
  about, 29-9
  See also pooling
database connections, setting, 4-19
database tables, creating from entity objects, 6-6
database transaction classes, custom, 25-27
DataBindings.cpx file, A-3
  about, 12-7, 12-9, A-6
  changing a page definition filename, 12-13
  dataControlUsages element, A-8
  elements, A-8
  elements, defined, 12-10
  PageDefinitionUsages element, A-8
  pageMap element, A-8
  runtime usage, 12-19
  sample, A-8

syntax, A-7
dataControl binding property, B-2
dataControlUsages element, 12-10, A-8
data-sources.xml file, 8-5
data-sources.xml file, not including in deployment, 34-30
date format masks, 6-15, 6-17
DBSequence
  displaying, 13-23
  entity object primary key and, 6-22, 26-27
debugging
  ADF binding context, 24-12
  ADF Model in JDeveloper, 24-6
  ADF Model layer, 24-10
  binding container, 24-16
  runtime errors, 24-4
decimal-separator element, 22-18
declarative bindings, types of, 1-8
declarative development, about, 1-4
declarative validation rules
  about, 6-26
  adding, 6-26
  entries in XML component definition file, 6-27
default page navigation cases, 16-6
default range size, 27-6
defName attribute, 15-15
deploying ADF applications, 34-1
  for testing purposes, 34-1
  from JDeveloper, 34-8
  overview, 34-1
  steps for, 34-2
  techniques for, 34-8
  to EAR file, 34-8
  to JBoss, 34-12
  to Oracle Application Server, 34-9
  to Tomcat, 34-17
  to WebLogic, 34-14
  to WebSphere, 34-16
  troubleshooting, 34-29
  using Ant, 34-9
  using scripts, 34-8
deploying SRDemo application, 34-9
detailStamp facet
  about, 14-12
  DisclosureEvent event, 14-14
  used to display inline detail table, 15-20
  using, 14-12
dialog navigation rules, 19-22
digital signatures, setting for web services, 33-12
disabled attribute
  about, 12-25, 13-16
  enabled property, 12-26, 13-16
DisclosureAllEvent event, 15-20
DisclosureEvent event
  detailStamp facet, 14-14
  in tree components, 15-15
  in treeTable components, 15-20
disclosureListener attribute
  detailStamp facet, 14-14
  in tree components, 15-15
  in treeTable components, 15-20
discriminators
  selecting attributes as, 26-20
  setting default values for, 26-20
displayData binding property, B-2
displayHint binding property, B-2
displayHints binding property, B-3
DML processing, overriding, 26-10
domains
  about, 26-2
  creating
    based on Oracle object types, 26-5
    based on simple Java types, 26-2
  custom code in, 26-3
  exception messages for, creating, 26-3
  extending Oracle data type example, 26-5
  String-based example, 26-4
dropdown lists
  adding ADF bindings to existing, 21-12
  List Binding Editor, 21-13
  navigation list binding, 19-63
dynamic list of values, 19-59
dynamic menus. *See* menus
dynamic outcomes. *See* outcomes
dynamic tables, 14-3

E

Edit Form Fields dialog, 13-10
edit forms
  about, 13-18
  creating, 13-18
Edit Table Columns dialog, 14-3, 14-10, 14-16
EL expressions
  accessing results in a managed bean, 17-11
  accessing security properties, 30-29
  ADF binding properties, 12-20, 12-27
  binding attributes with, 13-5
  binding object properties reference, B-1
  bindings variable, 12-20
  creating, 12-20, 12-21
  editing, 12-21
  examples of ADF binding expressions, 12-24
  Expression Builder, using to create, 12-21
  navigation operations, 13-15
  referencing binding objects, 12-20
  syntax for ADF binding expressions, 12-20
  tracing in JDeveloper, 24-23
  using to bind to ADF data control objects, 12-20
embedded OC4J server, deploying for testing, 34-2
enabled binding property, B-3
enabled property, 12-26, 13-16
enabledString binding property, B-3
entity cache, 7-26, 7-31
entity object attributes
  accessors for custom classes, generating, 6-39
  control hints, adding, 6-15
  declarative settings for, 6-19
  type mappings, default, 6-19
  type value, 6-19

updatable, 6-20
Entity Object Editor, 6-6
entity objects
attribute control hints, 6-15
attribute-level validation rules, 9-3
attribute-level validators, 6-29
basing on PL/SQL package and database view, 26-10
business rules, programmatic, 9-1
calculated attributes, 6-48
compared to 4GL concepts, 4-5 to 4-8
composition association settings, 6-25
creating for
existing tables, 6-2
synonyms and views, 6-6
custom Java classes, generating, 6-38
custom properties, adding, 6-18
database tables with no primary key constraints and, 6-5
DBSequence-valued primary keys and composition associations, 26-27
declarative features of, 6-19
declarative runtime behavior, configuring, 6-17
default values
assigning derived values before saving, 9-9
eagerly allocating from a sequence at create, 9-9
initializing at create, 9-8
initializing at refresh, 9-9
definition of, 4-3, 6-1
deleted flags, using, 26-7
diagram of, creating, 6-12
DML processing, overriding, 26-10
DML statement syntax, setting, 4-19
domain types, using, 26-3
editing, 6-6
entity associations. *See* associations
entity definition objects, 6-30
entity rows
accessing, 6-30
creating, 6-34
updating or removing, 6-33
entity-level validation rules, 9-3
entity-level validators, 6-29
extending, 25-30, 25-33
finding by primary key, 6-30
inheritance. *See* inheritance hierarchies
names of, 6-3
pending changes, undoing, 9-11
posting order
controlling, 26-24
default, 26-24
primary key for, 6-5, 6-21
primary key value, trigger-assigned, 6-22
programmatic access, 6-30
reference entities in view objects, 7-8
RETURNING clause, disabling, 26-17
row states, 9-4
runtime metadata, accessing, 25-10
synchronizing with underlying database tables, 6-7
tips and techniques for business logic in, 9-15 to 9-20
transient attributes, adding, 6-47
trigger-assigned values, 6-21
validation rules for, 6-26
validators. *See* method validators
view objects and, 7-24
wizard for creating, 6-6
XML component definition files for, 6-4
entity references, 7-3
entity rows
accessing, 6-30
accessing using association accessors, 9-15
creating, 6-34
default values
assigning derived values before saving, 9-9
eagerly allocating from a sequence at create, 9-9
initializing at create, 9-8
initializing at refresh, 9-9
entity cache, 7-26
new, refresh flags for, 9-11
pending changes, undoing, 9-11
states of, 9-4
updating or removing, 6-33
entity usages
entries in XML component definition files, 7-11
polymorphic, 27-34
primary, 7-8
secondary, adding, 7-8
view link consistency and, 27-2
entity-level validation rules, 9-3
`error` binding property, B-3
error messages
about, 20-21
ADF Business Components, customizing, 25-25
disabling client-side, 20-23
displaying server-side, 20-22
parameters in, 20-6
`estimatedRowCount` binding property, B-3
evaluating page navigation rules, 16-10
events
action, 13-16
`DisclosureAllEvent` event, 15-20
`DisclosureEvent` event, 14-14, 15-15, 15-20
`FocusEvent` event, 15-19
`LaunchEvent` event, 19-24
`PrepareRender` event, 13-9
`RangeChangeEvent` event, 14-9
`ReturnEvent` event, 19-27
`SelectionEvent` event, 14-17
exception handling
about, 20-23
changing, 20-24
custom handler, 20-24
customizing the lifecycle, 20-25
single page, overriding for, 20-26
`executables` element, 12-15
`Execute` operation, 18-1

execute property, 13-15
ExecuteWithParams operation, 18-11
existing components. *See* binding to existing components
Expression Builder
  about, 12-21
  about the object list, 12-23
  icons used, 12-23
  using, 12-22
expression language. *See* EL expressions

F

faces-config.oxd_faces file, 11-12, 16-8
faces-config.xml file
  about, A-3, A-26
  configuring for ADF Faces, 11-19
  converters, 20-20
  custom validators and converters, A-30
  editing, 11-7
  example of, 11-7
  from-action element, 16-3
  from-outcome element, 16-3
  from-view-id element, 16-3
  locales, 22-17
  managed bean configuration, 17-2, 17-3
  managed beans, A-30
  managed-bean element, 17-3
  message resource bundle, A-28
  navigation rules and classes, A-29
  navigation-case element, 16-3
  page navigation rule elements, 16-3
  phase listener
    default, 13-6
    registering new, 20-26
  phase listener for ADF Binding, A-27
  redirect element, 16-3
  render kit for ADF Faces, A-27
  required elements for ADF, 12-7
  supported locales, A-28
  supported tasks, A-27
  to-view-id element, 16-3
  using to define page navigation rules, 16-2
  validation, 20-14
  wizard for creating, 11-9
FacesContext class, 13-6
FacesServlet class, 13-6
facets
  about, 13-11
  actions facet, 19-44
  adding or removing, 11-27
  detailStamp facet, 14-12, 15-20
  footer facet, 13-11
  in panelPage components, 11-26
  in tree components, 15-13
  in tree table components, 15-19
  location facet, 19-43
  menu, 19-2
  nodeStamp facet, 15-13, 15-19, 19-13
  pathStamp facet, 15-19
  selection facet, 14-14
Factory-Substitution-List Java system property, 25-37
family element, 22-8
file dependencies, A-2
file uploading
  context parameters for, 19-53
  custom file processor for, 19-54
  disk space and memory for, 19-53
  inputFile component, 19-50
  storage location for files, 19-49
  supporting, 19-49
filter mappings, 12-11
filter-class element, 12-11
filter-name element, 12-11
Find mode
  about, 18-1
  setting automatically on iterator, 18-7
findByKey() method, 7-25, 7-41
findByPKExtended() method, 26-23
findByPrimaryKey() method, 6-21
findMode binding property, B-3
findRowSet() method, 27-12
findRowSetIterator() method, 27-12
fixed list of values, 19-55
FocusEvent event, 15-19
FocusListener listener, 15-19
footer facet, 13-11
foreign keys, DBSequence-assigned, 26-28
foreign-key relationships, 15-1
format masks
  date, 6-15, 6-17
  number, 5-12
forms
  adding UI components, 13-25
  changing order of UI components, 13-25
  creating basic, 13-9
  creating edit, 13-18
  creating input, 13-21
  creating search, 18-1
  creation, 13-21
  deleting UI components, 13-25
  footer facet, 13-11
  modifying the default, 13-25
  navigation operations, 13-12
  parameterized, 18-11
  using the Data Control Palette to create, 13-11
  using to display master-detail objects, 15-4
  widgets for basic, 13-10
Forms development, ADF Business Components and, 4-4
from-action element, 16-3
from-outcome element, 16-3
from-view-id element, 16-3
fullName binding property, B-3

G

getAsObject method, 20-19
getAsString method, 20-19

getBundle method, 22-14
getClientConversion() method, 20-20
getClientScript() method, 20-15, 20-20
getClientValidation() method, 20-15
getContents() method, 22-16
getCurrentRow() method, 8-13
getCurrentRowIndex() method, 5-21
getEntityState() method, 9-5
getEstimatedRangePageCount() method, 27-9
getEstimatedRowCount() method, 5-19
getPostState() method, 9-5
global buttons, 19-2
global page navigation rules, 16-6
graph element, A-15

H

hierarchical menus. *See* menus
HTTP sessions, 28-3

I

id attribute, 12-13
id element, 22-8
immediate attribute, 13-8
inheritance hierarchies, 26-19
  about, 26-18
  base entity objects in, 26-20
  discriminators
    selecting attributes as, 26-20
    setting default values for, 26-20
  entity object methods, adding common, 26-22
  illustration of, 26-19
  subtype entity objects
    adding specific methods, 26-22
    creating, 26-21
    finding by primary key, 26-23
    overriding common methods, 26-22
Initialize Business Components Project dialog, 4-19
Initialize Context phase, 13-7, 13-9
init-param element, 12-12
inline tables, 15-20
input forms
  about, 13-21
  Create operation, 13-23
  creating multiple objects, 13-24
inputFile components, use of, 19-50
inputText components. *See* text fields; forms
inputValue binding property, B-4
Insert ActionListener dialog, 17-7
Insert SelectManyShuttle dialog, 19-70
internationalization
  about, 22-10
  procedures for, 22-14
  See also localizing
Invoke Application phase, 13-9
invokeAction bindings, 12-15
invokeAction element, A-13
isAttributeUpdateable() method, 9-20
isExpanded method, 15-15
iterator bindings
  about, 12-15, 13-3
  findMode and, 10-15
  range, 12-16
  rangeSize attribute, 13-14
  referencing row index and row count, 10-12
  Refresh and RefreshCondition properties, 10-19
  setting ADF authorization grants, 30-26
  tables, 14-5
iterator element, A-13
iteratorBinding binding property, B-4
iterators
  about, 12-15
  creating new, 18-8
  method, 12-15
  RowSetIterator object, 12-16
  setting Find mode on, 18-7
  variable, 12-15

J

j_security_check login method, 30-15
j2ee-logging.xml file, A-25
Java message bundle files
  about, 5-13
  entity object example, 6-16
  localized, 5-14
  validation error messages, 6-28
  view object example, 5-13
javax.faces.CONFIG_FILES context parameter, 11-9
javax.faces.STATE_SAVING_METHOD context parameter, 11-38
jazn-data.xml file, 34-31
jazn-data.xml, with web services, 33-10
JBO-30003 error, 34-31
jbo.ampool.doampooling configuration property, 28-8, 28-20
jbo.doconnectionpooling configuration parameter, 28-22
jbo.locking.mode property, 5-12, 7-30, 28-21
jbo.security.enforce property, 30-12
JBoss, deploying applications to, 34-12
jbo.viewlink.consistent configuration parameter, 27-2
JDBC datasource connections, 8-4
JDBC URL connections, 8-4
JSF
  about, 1-5
  Oracle ADF and, 1-4
JSF Configuration Editor
  launching, 11-7
  using to create managed beans, 17-2
  using to define page navigation rules, 16-2, 16-5, 16-6
JSF Core tag library, 11-14
JSF HTML tag library, 11-14
JSF lifecycle, with ADF, 13-6
JSF Navigation Case, 16-4

JSF navigation diagrams
  deleting JSF pages on, 11-13
  faces-config.oxd_faces files for, 11-12
  opening, 11-11
  renaming JSF pages on, 11-13
  using to define page navigation rules, 16-2
  See also JSF Navigation Modeler
JSF Navigation Modeler
  deleting pages, 16-13
  refreshing, 16-13
  using to define page navigation rules, 16-3, 16-13
JSF Page Flow & Configuration wizard, 11-9
JSF pages
  automatic component binding in, 11-34
  backing beans for, 11-30
  creating from the JSF navigation diagram, 11-11
  designing for ADF bindings, 21-2
  editing, 11-15
  effects of name change, 12-9
  example in XML, 11-12
  inserting UI components, 11-15
  laying out, 11-24 to 11-30
  loading a resource bundle, 22-16
  referencing backing beans in, 11-33
  ways to create, 11-10
JSF servlet and mapping configuration settings, 11-6
JSF tag libraries, 11-14
JSP documents, 11-10
  benefits of using, 2-19
JUnit extension, installing, 3-10

K

keystores
  creating, 33-6
  exporting public key, 33-8
  requesting certificates, 33-7
  using with web services data controls, 33-6

L

label attribute, 13-26
label binding property, B-4
labels binding property, B-4
labels, changing for UI components, 13-26
labelSet binding property, B-4
LaunchEvent event, 19-24
LaunchListener listener, 19-28
libraries
  ADF Controller, 12-8
  ADF Model Runtime, 12-8
  ADF runtime, 12-8
  adf-controller.jar file, 12-8
  adfm.jar, 12-8
lifecycle
  error handling, 20-2
  JSF and ADF, 13-6, 29-3
  phases in an ADF application, 13-7
lifecycle phases.*See names of individual phases*
lifecycle, page
  customizing for single page, 10-17
  customizing globally, 10-16
  JSF and ADF, illustration of, 29-2
  state management release level, setting, 28-11
links, command. *See* command links
List Binding Editor, 19-57, 19-60, 19-63, 21-13
list bindings, 12-18
list components, creating, 19-55
list element, A-16
list of values
  adding ADF bindings to existing, 21-12
  dynamic list, 19-59
  fixed list, 19-55
  List Binding Editor, 19-57, 19-60, 21-13
  list binding object, 19-58, 19-62
listeners
  DisclosureListener listener, 15-15
  FocusListener listener, 15-19
  LaunchListener listener, 19-28
  ReturnListener listener, 19-27
ListResourceBundle
  about, 22-13
  creating, 22-15
loadBundle tag
  about, 22-11
  using, 22-16
locales, registering, 22-17
localizing
  about, 22-10
  ListResourceBundle, 22-13
  property files, requirements, 22-13
  See also internationalization
location facet, 19-43
logging, A-25
  changing logging level for ADF packages, A-25
  log file location, A-26
  redirecting output, A-26
login page, 30-15
logout page, 30-19

M

managed beans
  accessing EL expression results, 17-11
  automatic component binding and, 11-34
  chaining, 19-11
  compatible scopes, 19-11
  configuring for menus, 19-4, 19-9, 19-11
  configuring for process trains, 19-39, 19-40, 19-42
  configuring in faces-config.xml, 11-31, 17-2
  creation at runtime, 17-2
  definition of, 1-6
  managed properties in, 11-36
  multiple page usage, 17-3
  overriding declarative methods in, 17-8
  scope for backing beans with method
    overrides, 17-11
  scope types, 11-31
  storing information on, 17-2, 18-15
  using in page navigation components, 16-16

validation method, 20-13
value binding expressions for chaining, 19-7
managed-bean element, 17-3
mandatory binding property, B-4
mandatory property, 20-5, 20-9
mappings
ADF binding filter, 12-11
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
master-detail coordination
combining types of, 27-5
multiple master view objects, setting, 27-10
types of, 27-4
See also master-detail objects
master-detail objects
about, 15-1
displaying in
detailStamp facet, 15-20
forms, 15-4
separate pages, 15-9
tables, 15-4
tree components, 15-9
treeTable components, 15-17
example of, 15-4
in the Data Control Palette, 15-2
managing row currency, 15-9
managing synchronization of data, 15-9
MasterTable, Inline Detail widget, 15-21
RowSetIterator objects, 15-8
treeTable components, 15-18
widgets, 15-5
Master-Details widgets, 15-5
maximum row fetch size, 27-1
MenuModel class, 19-3
menus
components for, 19-13
facets for, 19-2
managed beans for, 19-4, 19-9, 19-11
menu model creation, 19-3 to 19-11
menu tree model, 19-7
navigation rules for, 19-16
nodeStamp facet, 19-13
startDepth attribute, 19-14
ViewIdPropertyMenuModel instance, 19-9
ways to create, 19-2
message bundle files. *See* Java message bundle files
messages tag, 20-9, 20-22
messages, error
about, 20-21
ADF Business Components, customizing, 25-25
disabling client-side, 20-23
displaying server-side, 20-22
parameters in, 20-6
Metadata Commit phase, 13-9
metadata files
about, A-1
method action binding objects, 12-18
method action bindings
setting ADF authorization grants, 30-27
method iterators, 12-15
method returns, 12-4
method validators
about, 9-5
attribute-level, creating, 9-5
entity-level, creating, 9-7
entries in XML component definition files, 9-6, 9-8
error message parameter for invalid value, 9-8
using validation view objects in, example of, 9-13
methodAction element, A-16
methodIterator element, A-13
methods
adding logic to, 17-8, 17-12
binding to command components, 17-5
in page navigation components, 16-14
overriding declarative, 17-8, 17-12
populating parameters at runtime, 18-13
providing parameters when binding, 17-5
MVC architecture, 1-1

N

name binding property, B-4
named bind variables
about, 5-26
adding, 5-26, 5-30
advanced uses of, 27-14
creating search forms with, 18-11
default NULL values, 5-29
inspecting in Business Components Browser, 5-28
runtime errors, 5-28
setting values programmatically, 5-29
test client program example for, 5-31
navigation list binding, 19-63
navigation menus. *See* menus
navigation modeler. *See* JSF Navigation Modeler
navigation operations
action events, 13-16
Back button, issues, 13-17
EL expressions for, 13-15
inserting, 13-12
types, 13-15
navigation rules, page
about, 16-2
conflicts, 16-12
creating, 16-2
default cases, 16-6
deleting, 16-13
dialogs, for launching, 19-22
evaluating at runtime, 16-10
examples of, 16-8
global, 16-2, 16-6
in multiple configuration files, 16-11
menus, for, 19-16
overlapping, 16-11
pattern-based, 16-2

splitting, 16-12
navigation, page
about, 16-1
binding to a backing bean, 16-17
binding to a data control method, 16-17
default cases, 16-20
dialogs, for launching, 19-22
dynamic outcomes, 16-1, 16-16
`from-action` element, 16-3
`from-outcome` element, 16-3
`from-view-id` element, 16-3
global rules, 16-2
menus, for, 19-16
`navigation-case` element, 16-3
`NavigationHandler` handler, 16-10
`navigation-rule` element, 16-3
pattern-based rules, 16-2
`redirect` element, 16-3
rules
about, 16-2
conflicts, 16-12
creating, 16-2
default cases, 16-6
deleting, 16-13
evaluating at runtime, 16-10
examples of, 16-8
global, 16-2, 16-6
in multiple configuration files, 16-11
overlapping, 16-11
pattern-based, 16-2, 16-5
using the JSF Configuration Editor, 16-6
using the JSF Navigation Modeler, 16-3
static outcomes, 16-1, 16-14
`to-view-id` element, 16-3
using action listeners, 16-20
using outcomes, 16-1
using the JSF Navigation Modeler, 16-13
navigation, range
forms, 13-12
`row` attribute, 14-8
tables, 14-7
`navigation-case` element, 16-3
navigation-case Properties dialog, 16-5
`NavigationHandler` handler, 16-10
`navigation-rule` element, 16-3
`nodeDefinition` element, 15-15
`nodeStamp` facet, 15-13, 15-19, 19-13
number format masks, 5-12
`number-grouping-separator` element, 22-18

O

`operationEnabled` binding property, B-4
operations
accessing from the Data Control Palette, 12-5
action events for navigation, 13-16
built-in, 10-7
described, 13-20
EL expressions for navigation, 13-15
`Execute`, 18-1
`ExecuteWithParams`, 18-11
for searches, 18-5
in page navigation components, 16-14
navigation, 13-12, 13-14, 13-15
Oracle ADF
4GL development and, 1-4
architecture, 1-1
debugging the Model layer, 24-10
file syntax, A-4
Forms concepts and, 4-4
JSF and, 1-4
key components for declarative development, 1-4
security features, 30-1
supported technologies, 1-1
Oracle data types
list of, 4-14
setting usage of, 4-20
Oracle Wallet, 33-6
`oracle.adf.view.faces.CHECK_FILE_MODIFICATION` context parameter, 11-38
`oracle.adf.view.faces.DEBUG_JAVASCRIPT` context parameter, 11-38
`oracle.adf.view.faces.USE_APPLICATION_VIEW_CACHE` context parameter, 11-40
`oracle.jbo` package, 4-11, 5-18
`oracle.jbo.client` package, 8-16
`oracle.jbo.domain` package, 4-14
`oracle.jbo.rules` package, 26-31
`oracle.jbo.server` package, 4-11
`oracle.jbo.server.SessionImpl` object, 9-16
`oracle.jbo.Session` interface, 9-16
`ORDER BY` clause, 7-4
`orion-web.xml` file, 8-5
outcomes
dynamic, 16-1, 16-16
page navigation, 16-1
static, 16-1, 16-14
`outputText` components. *See* text fields; forms

P

package XML files, disabling the use of, 4-14
page controllers, 11-8
page definition files
about, 12-8, 12-12, A-3, A-8
action bindings, 12-18
at runtime, 12-19
attribute bindings, 12-18
binding containers, 12-19
binding objects, 12-13
`bindings` element, 12-18
creating, 12-12
effects of name change, 12-9, 12-12
elements, 12-13, A-10
`executables` element, 12-15
`id` attribute, 12-13
`invokeAction` bindings, 12-15
iterator bindings, 12-15
list bindings, 12-18
location, 12-12

mapped in the DataBindings.cpx file, 12-13
method bindings, 12-18
naming, 12-12
nodeDefinition element, 15-15
parameters, 12-14
parameters element, 12-14
rangeSize attribute, 12-17
refresh attribute, 12-16, 12-17
refreshCondition attribute, 12-16
renaming, A-9
sample, A-18, A-19
syntax, A-9
table bindings, 12-18
tree bindings, 12-18, 15-14
value bindings, 12-18
page element, A-14
page layouts, 11-24 to 11-30
page lifecycle
customizing for single page, 10-17
customizing globally, 10-16
JSF and ADF, illustration of, 29-2
state management release level, setting, 28-11
page navigation. *See* navigation, page
Page Properties dialog, 11-34
pageDefinition element, A-10
pageDefinitionUsages element, 12-10, A-8
PageDef.xml file. *See* page definition files
pageMap element, 12-10, A-8
paging large result sets, 27-7
panelButtonBar components, 13-15
panelPage components
facets in, 11-26
inserting into pages, 11-15
uses of, 11-14
parameter element, A-11
parameter methods, passing values to, 17-7
parameters
accessing values, 17-5
Apply Request Values phase, 13-8
bindings for, 17-5
defined in page definition file, 12-14
for messages, 20-6
for methods, 17-5
NamedData element, 17-5
on the Data Control Palette, 12-5
passing values for, 17-7
Prepare Model phase, 13-8, 13-9
providing for methods, 17-5
setting for methods, 17-5
setting on setActionListener components, 17-7
parameters element, 12-14
param-name element, 12-11
partial keys, 27-11
partial page rendering
attributes for enabling, 19-34
autoSubmit attribute and, 11-39
command components and, 19-35
panelPartialRoot tag and, 11-22
partialSubmit attribute, 19-23
partialTriggers attribute, 19-34
pathStamp facet, 15-19
pattern attribute, 20-18
pattern-based page navigation rules, 16-5
PeopleTools, ADF Business Components and, 4-6
permission grants for ADF Security, 30-24
phase listeners
creating custom, 20-26
registering in the web.xml file, 12-7
phase-listener element, 12-7
PL/SQL procedures and functions, calling from ADF custom Java classes, 25-16
pooling, application module
about, 29-1
configuration parameters
for pool behavior, 29-12
for pool cleanup, 29-13
for pool sizing, 29-13
setting as Java system parameters, 29-7
setting declaratively, 29-5
setting programmatically, 29-7
database connection and application module pools
about, 29-10
cooperation between, 29-17
multiple JVM scenario, 29-11
single JVM scenario, 29-10
type of states in pools, 29-2
types of pools, 29-9
pooling, database connection
application module and database connection pools
about, 29-10
cooperation between, 29-17
configuration parameters for, 29-15
database user state and, 29-18
multiple JVM, 29-11
single JVM, 29-10
popup dialogs
closing and returning from, 19-24
components with built-in support for, 19-33
conditions for supporting, 19-21
creating, 19-21 to 19-28
launch event, 19-24
launch listener, 19-28
launching from command components, 19-23
navigation rules for launching, 19-22
passing values into, 19-28
return event and return listener, 19-27
return value handling, 19-27
tasks for supporting, 19-22
postback property, using in refreshCondition attribute, 13-9
postChanges() method, 9-4, 9-5, 28-21
PPR. *See* partial page rendering
Prepare Model phase
about, 13-8
when navigating, 13-9
Prepare Render phase
about, 13-9
exception handling, 20-23
overriding, 20-25

prepareForDML() method, 9-9
prepareModel method, 20-25
PrepareRender event, 13-9
prepareSession() method, overriding, 8-21
primary keys, DBSequence-valued, 26-27
process trains
  page access control, 19-42
  processChoiceBar components, binding to train models, 19-44
  processTrain components, binding to train models, 19-43
  train model creation, 19-39 to 19-42
Process Validations phase, 13-8
processChoiceBar components, binding to train models, 19-44
ProcessMenuModel class, 19-39, 19-41
processScope scope, 19-28
processTrain components, binding to train models, 19-43
Project Properties dialog, 11-8
projects
  creating from WAR files, 11-4
  dependencies on, 11-8
  JSF technology in, 11-5
  properties of, 11-5, 11-12
  renaming, 11-4
  view or user interface, 11-8
property files
  creating for resource bundles, 22-15
  requirements for resource bundles, 22-13
PropertyManager class, 29-4
pseudo class, 22-6
  creating, 22-7
  referencing, 22-7
pseudo elements, 22-6

Q

query-by-example search, 18-1
query-by-example view criteria
  testing, 5-25
  using, 5-22

R

range navigation. *See* navigation, range
range paging, 27-7
range, iterator, 12-16
RANGE_PAGING_AUTO_POST access mode, 27-9
RangeChangeEvent event, 14-9
rangeSize attribute, 12-17, 13-14, 14-8
rangeSize binding property, B-5
rangeStart binding property, B-5
rebinding
  input components, 13-26
  tables, 14-11
recommended technologies for 4GL developers, 1-3
redirect element, 16-3
REF CUSOR, view objects and, 27-56
reference entities, 7-8
refresh attribute, 12-16, 12-17
  about, 13-8
  Find mode, 18-7
REFRESH_FORGET_NEW_ROWS flag, 9-9
REFRESH_REMOVE_NEW_ROWS flag, 9-9
refreshCondition attribute, 12-16
  about, 13-8
  Find mode, 18-7
releaseApplicationModule() method, 8-16
Render Response phase, 13-9
render-kit-id element, 22-9
reportErrors method, 20-23 to 20-25, 20-26
reportException method, 20-24, 20-26
required attribute
  table row selection components, 14-15
  validation, 20-4, 20-9
resetRange binding property, B-5
resource bundles
  creating as a property file, 22-15
  creating as Java classes, 22-15
  for skins
    creating, 22-8
    registering, 22-8
    using, 22-6
  ListResourceBundle, 22-13
  loading onto a JSF page, 22-16
  property files, 22-13
  property files versus Java classes, 22-13
Restore View phase, 13-7
restoreState method, 20-14
result binding property, B-5
returnActionListener tag, 19-25
ReturnEvent event, 19-27
RETURNING clause, 26-17
ReturnListener listener, 19-27
right-to-left element, 22-18
Rollback operation, 13-18
rootNodeBinding binding property, B-5
row, 12-3, 12-5
row currency
  on master-detail objects, 15-9
  setting programmatically, 14-22
row iterators, iterator mode flags and, 27-7
RowMatch objects
  in-memory filtering and, 27-32
  view link consistency and, 27-3
rows attribute
  about, 14-8
  binding to rangeSize attribute, 14-8
  first attribute, 14-8
rows, view object
  filtering with RowMatch objects, 27-32
  filtering with view criteria, 27-30
  sorting in memory, 27-27
rowsByDepth attribute, 15-20
rowset, 12-3, 12-5
RowSetIterator objects
  about, 12-16
  scope, 12-19
  used to manage master-detail objects, 15-8

rules, page navigation
about, 16-2
conflicts, 16-12
creating, 16-2
default cases, 16-6
deleting, 16-13
dialogs, for launching, 19-22
evaluating at runtime, 16-10
examples of, 16-8
global, 16-2, 16-6
in multiple configuration files, 16-11
menus, for, 19-16
pattern-based, 16-2

S

SAML assertion tokens, for web services, 33-11
saveState method, 20-14
scope, binding containers and objects, 12-19
scrollToRangePage() method, 27-9
search forms
about, 18-1
conditionally displaying results table, 18-14
EnterQuery/ExecuteQuery
about, 18-2
creating, 18-3
named bind variables
about, 18-2
using, 18-11
parameterized
about, 18-2
creating, 18-11
query-by-example, about, 18-1
results, on same page, 18-8
web
about, 18-2
creating, 18-6
security
for ADF Business Components applications, 30-12
for ADF web applications, 30-1
for web service data controls, 33-5
selectBooleanCheckbox components, in a table, 14-11
selectedValue binding property, B-6
selection facet, 14-14, 14-17
selection list components
adding ADF bindings to existing, 21-12
creating, 19-55
SelectionEvent event, 14-17
selectionState attribute, 14-17
selectItems tag, 19-58
selectManyShuttle components, creating, 19-65
selectOneChoice components
creating, 19-63
in a table, 14-11
selectOneListbox components
creating, 19-59
selectOneListbox components, in a table, 14-11
selectOneRadio components
creating, 19-55
selectOneRadio components, in a table, 14-11
selectors, 22-5
selectRangeChoiceBar components
about, 14-7
at runtime, 14-8
RangeChangeEvent event, 14-9
sequences
displaying, 13-23
service methods
adding, 8-6
displayed in Data Control Palette, 10-5
guidelines for, 8-9
method signatures, 8-13
publishing, 8-11
testing, 6-36
servlet context parameter, 12-11
setActionListener components
search pages, conditionally displaying results, 18-15
setting, 17-7
setActionListener tag, 19-18
setAssociationConsistent() method, 27-3
setCurrentRowWithKey operation
compared to setCurrentRowWithKeyValue, 10-21
setting programmatically, 14-22
setMaxFetchSize() method, 27-2
setNewRowState() method, 9-4
setQueryMode() method, 27-27
setRangeSize() method, 27-6
setSubmittedValue method, 13-8
setViewLinkAccessorRetained() method, 27-13
setWhereClause() method, 5-24
SiebelTools, ADF Business Components and, 4-7
skin element, 22-8
skin-family element, 22-9
skins
about, 22-3
alias pseudo class, 22-6
configuring an application to use, 22-9
creating, 22-6
creating a resource bundle for, 22-8
icons, for, 22-7
minimal, 22-3
Oracle, 22-3
pseudo class
about, 22-6
creating, 22-7
pseudo elements, 22-6
registering, 22-8
resource bundles
about, 22-6
creating, 22-8
registering, 22-8
rtl pseudo class, 22-7
selectors, 22-5
simple, 22-4
using, 22-5

SOAP, and web services, 33-2
SQL alias, 5-5
SQL expressions, 5-5
SQL queries
column names and view object attribute names, 5-4
expert mode, 27-18, 27-22
named bind variables, referencing, 5-27
read-only view objects, defining for, 5-3
SQL expressions in, 5-5
view criteria and, 5-24
SQL Statement wizard page, 5-3
SQL tracing, 27-15
SQL-calculated attributes, 7-21
SRDemo application
data binding in, overview of, 10-23 to 10-36
database user state, setting, 29-19
functionality, 3-11
installing, 3-4
JUnit tests, running, 3-10
overview, 3-1
refreshing data, 3-6
requirements, 3-2
running, 3-8
schema, 3-2
stack trace, reporting information in, 24-10
standalone OC4J, deploying for testing, 34-2
`startDepth` attribute, 19-14
state management
ADF, 28-4
custom user information, reading and writing, 28-17
failover mode and passivation, 28-7, 28-15
for transient view objects, 28-19
general information about, 28-1
HTTP sessions, timing out, 28-15
middle-tier savepoints and, 28-20
passivation store, types of, 28-14
passivation versus activation, 28-6
passivation, database-backed, 28-13
pending database state and, 28-21, 28-22
release levels
pending database state and, 28-22
setting at runtime, 28-9
supported, 28-8
schema, controlling, 28-13
state cleanup, automatic, 28-14
storage information, 28-13
temporary storage, cleaning up, 28-16
transactional versus non-transactional state, 28-12
state saving, 11-38
`StateHolder` interface, 20-14
static list of values, 19-55
static outcomes. *See* outcomes
stored procedures and functions, calling from ADF custom Java classes, 25-16
style properties, changing, 22-2
StyleClass dialog, 22-2
`style-sheet-name` element, 22-9
`submitForm` method, 20-4
Synchronize with Database dialog, 6-7

T

table binding objects, 12-18
`table` element, A-17
`table` tag, 14-5
tables
about, 14-2
adding ADF bindings to existing, 21-8
attributes for, 14-6
Back button, using, 14-9
bindings for, 14-5
changing default, 14-10
conditionally displaying on search page, 18-14
creating, 14-2
`detailStamp` facet
about, 14-12
using, 14-12
dynamic tables, 14-3
master table with inline detail table, 15-20
master-detail objects, displaying in, 15-4
read-only, 14-3
rebinding, 14-11
`selection` facet, 14-14
`selectRangeChoiceBar` components, 14-7
about, 14-7
`table` tag, 14-5
`var` attribute, 14-6
versus forms, 14-2
widgets for, 14-3
`tableSelectMany` components
about, 14-14
`autoSubmit` attribute, 14-15
`required` attribute, 14-15
`text` attribute, 14-15
using, 14-18
`tableSelectOne` components
about, 14-14
adding to a table, 14-4, 14-11
`autoSubmit` attribute, 14-15
`required` attribute, 14-15
`text` attribute, 14-15
using, 14-16
tag libraries for JSF and ADF Faces, 11-14
technologies recommended for 4GL developers, 1-3
technologies supported in Oracle ADF, 1-1
`text` attribute, 14-15
text fields
adding ADF bindings to existing, 21-7
binding, 13-2
creating for attributes, 13-2
input text widgets, 13-2
label widgets, 13-2
output text widgets, 13-2
using the Data Control Palette to create, 13-3
`time-zone` element, 22-18
token validation
forms, 13-17

setting, 13-17
tables, 14-9
Tomcat, deploying applications to, 34-17
tooltip binding property, B-6
to-view-id element, 16-3
Transaction object, 7-26
transactional state, 28-12
transient attributes
adding, 6-47
calculated, 7-24
calculated values, displaying, 6-48
entity-based view objects, adding to, 7-22
entries in XML component definition file, 6-48
passivation and, 28-19
Tree Binding Editor, 15-11, 15-21
tree components
about, 15-9
Accessors element, 15-15
adding ADF bindings to existing, 21-14
AttrNames element, 15-15
binding objects created for, 15-14
defName attribute, 15-15
DisclosureEvent event, 15-15
disclosureListener attribute, 15-15
example of, 15-10
facet tag, 15-13
FocusEvent event, 15-19
FocusListener listener, 15-19
isExpanded method, 15-15
nodeDefinition tag, 15-15
nodeStamp facet, 15-13
Tree Binding Editor, 15-11
treeModel property, 15-13
using to display master-detail objects, 15-9
var attribute, 15-13
tree element, A-17
TreeModel class, 15-19
treeModel property, 15-13, 15-19
treeState attribute, 15-20
treeTable components
about, 15-17
Accessors element, 15-15
adding ADF bindings to existing, 21-14
AttrNames element, 15-15
creating from Data Control Palette, 15-18
defName attribute, 15-15
DisclosureAllEvent event, 15-20
DisclosureEvent event, 15-20
disclosureListener attribute, 15-20
displaying master-detail objects, 15-17
example of, 15-17
facet tag, 15-19
nodeStamp facet, 15-19
pathStamp facet, 15-19
rowsByDepth attribute, 15-20
TreeModel class, 15-19
treeModel property, 15-19
treeState attribute, 15-20
var attribute, 15-19

U

UDDI, and web services, 33-2
UI components
adding ADF bindings to existing, 21-3
adding binding for, 13-25
adding to a form, 13-25
binding instances of, 11-33
changing labels for, 13-26
changing the display order on forms, 13-25
conditionally displaying, 18-14
creating with the Data Control Palette, 12-1, 12-6, 12-7
default ADF features, 12-7
deleting bindings for, 13-25
deleting from a form, 13-25
editing for tables, 14-10
inserting into JSF pages, 11-15
modifying, 13-25
rebinding, 13-25, 13-26
skins, 22-3
style properties, 22-2
UI control hints
adding, 5-12, 6-15
Java message bundles and, 5-13
structure in Java message bundles, 5-13
UML diagrams
about, 6-14
creating, 8-23
display options, 8-24
illustration of, 6-13, 10-4
view objects entity usages, 7-12
working with, 8-24
Update Model Values phase, 13-8
updateable binding property, B-6
username token authentication, for web services, 33-11

V

validate method, 20-10, 20-16
Validate Model Updates phase, 13-8
validation
ADF Faces, 20-4
ADF Faces attributes, 20-5
ADF Faces validators, 20-6
ADF Model validation rules
adding, 20-8
types of, 20-8
client-side custom JSF validators
creating, 20-15
using, 20-15
custom JSF validators
about, 20-12
creating, 20-13
custom validation using domains, 26-1
entity object, 6-26
lifecycle, 20-2
method, overriding, 20-12
parameters in messages, 20-6
required attribute, 20-4

runtime, 20-10
view objects for, creating, 9-11
working in an application, 20-2
validation cycle
about, 9-2
commit processing and, 9-3
infinite, 9-4
upon failures, 9-4
Validation Rules Editor dialog, 20-8
validation rules, custom
creating, 26-31
customizers for, 26-33
registering and using in projects, 26-34
validation rules, declarative
about, 6-26
adding, 6-26
attribute-level, 9-3
entity-level, 9-3
entries in XML component definition file, 6-27
error message parameter for invalid value, 9-8
error messages, 9-8
`Validator` interface, 20-12
`validator` tag, 20-9
validators
ADF Faces, 20-6
attribute-level, 6-29
custom JSF, creating, 20-14
entity-level, 6-29
See also method validators
value bindings
about, 12-18, 13-4
changing, 13-26
table, 14-5
`var` attribute
tables, 14-6
tree tables, 15-19
trees, 15-13
variable iterators
about, 12-15
`variable` element, 12-15
`variableUsage` element, 12-15
`variableIterator` element, A-14
variables
at runtime, 18-13
`variableUsage` element, 12-15
versioning
committing ADF work to CVS, 32-2
developer-level activities, 32-4
name consistency, 32-2
team-level activities, 32-3
view caching, 11-40
view criteria
named
applying, 27-25
defining, 27-24
removing, 27-25
using at runtime, 27-26
testing, 5-25
using, 5-22
using in memory, 27-30
view link accessor attributes, 27-4, 27-13
view link accessors, 27-4
view links
about, 5-38
active versus passive, 5-41
association-based, 7-13, 7-15
creating, 5-36, 7-14
master-detail coordination styles, 27-4
view link accessor attributes, exposing, 26-9
view link accessors, using, 5-39
view link consistency, using, 27-2 to 27-4
XML component definition files for, 5-38, 7-15
view object attributes
control hints, adding, 5-12
declarative settings for, 5-6
names of, 5-4
type value, 5-6
View Object Editor, opening, 5-5
view object instances
application modules and, 5-8
creating at runtime for validation, example of, 9-12
differences between view objects and, 5-8
displayed in Data Control Palette, 10-8
names of, defining, 5-8
operations, built-in, 10-8
view objects
application modules and, 5-6
client interfaces, 8-12
compared to 4GL concepts, 4-5 to 4-8
control hints, adding, 5-12
custom classes, generating, 5-43 to 5-47
data-retrieval methods, 5-18
declaratively controlling insert, update, and delete, 27-65
definition of, 4-4
difference between read-only and entity-based, 7-40
differences between view object instances and, 5-8
domain types, using, 26-3
dynamic attributes and, 27-12
entity objects and, 7-24
entity usages, multiple, 27-63
entity-based
about, 7-1, 7-6, 7-24
calculated attributes entries in XML file, 7-22, 7-23
calculated attributes, types of, 7-21
compared to read-only, 7-40
creating, 7-2, 7-5
editing, 7-6
executing query, 7-28
in multiuser environments, 7-34
partial keys, `findByKey()` and, 27-11
primary entity usage, 7-8
programmatic access examples, 7-34
programmatic, creating, 27-54
reference entities, adding, 7-8
requerying, 7-31

SQL-calculated attributes, adding, 7-21
testing, 7-16
transient attributes, adding, 7-22
view link consistency, using, 27-2 to 27-4
extending, 25-30, 25-33
filtering in memory, 27-32
forward only mode, 27-17
internally created, 27-5
Java message bundles for, 5-13
join queries and linked master-detail queries, 5-33
master-detail using association-based view links, 7-14
master-detail using view links, 5-36
maximum fetch size, 27-1
named bind variables, adding, 5-26
passivation and, 28-19
polymorphic entity usages, 27-34
polymorphic view rows, 27-38
programmatic, using, 27-54
query modes, 27-27
query results in stored procedures, working with, 27-56
query-by-example view criteria, using, 5-22
range paging, 27-7
range size and data scrolling, 27-6
read-only
compared to entity-based, 7-40
creating, 5-2 to 5-5
editing, 5-5
inline views and, 5-32
join tables in, 5-34
programmatic, creating, 27-54
testing, 5-16
reference entities, adding, 7-8
row sets and row set iterators, multiple, 27-12
`RowMatch` objects, using, 27-32
`RowSet` and `RowSetIterator`, 5-18
runtime metadata, accessing, 25-10
runtime versus design time creation, 27-17
sorting and searching in memory, 27-26 to 27-32
SQL statement syntax, setting, 4-19
static data, populating with, 27-59
tuning for performance, 27-13 to 27-17
Tuning panel options, 27-16
validation
creating at runtime, 9-11
example usage, 9-13
view row client interfaces, 8-12
XML
consuming, 27-50
generating, 27-43
XML component definition files for, 5-5, 7-6
view row attributes, modifying, 7-29
view row client interfaces, 8-13
`view.PageDefs` package, 12-12

W

Web 2.0 pages, 1-16
web configuration files, A-3
web pages. *See* JSF pages
web search forms
about, 18-2
creating, 18-6
web services
about, 33-1
authentication, 33-13
creating data controls, 33-4
defining data control security, 33-9
encrypting and decrypting, 33-13
JAZN, 33-10
keystores, 33-6, 33-13
publishing application modules, 33-14
SAML assertion tokens, 33-11
securing data controls, 33-5
setting authentication, 33-9
setting digital signatures, 33-12
SOAP, 33-2
testing authentication, 33-10
UDDI, 33-2
username token authentication, 33-11
WSDL, 33-2
WS-Security, 33-5
X509 authentication, 33-11
WebLogic, deploying applications to, 34-14
WebSphere
configuring to run ADF applications, 34-21
deploying applications to, 34-16
`web.xml` file, 12-7, A-3, A-20
ADF filter mappings, 12-11
ADF model binding, A-24
application view caching, A-22
configuring for ADF Faces, 11-18
debugging, A-22
editing, 11-6
example of, 11-5
JSF parameters, A-24
registering the ADF binding filter, 12-10
saving state, A-21
servlet context parameter, defining, 12-11
tasks supported by, A-21
uploading, A-23
`WHERE` clause
join view objects, 7-13
multiple view criteria and, 5-24
named bind variables and, 5-26
view link consistency and, 27-4
WSDL, and web services, 33-2
WS-Security, about, 33-5

X

X509 authentication, for web services, 33-11
XML component definition files
application module, 5-10
definition of, 4-11
entity object, 6-4
entries for calculated attributes, 7-22, 7-23
entries for entity usages, 7-11

entries for transient attributes, 6-48
extended components, 25-30
view object, entity-based, 7-6
view object, read-only, 5-5