Oracle® Application Development Framework

Tutorial for Forms/4GL Developers (10.1.3.1.0)

Revised, January 2008

Oracle Application Development Framework: Tutorial for Forms/4GL Developers (10.1.3.1.0)

Copyright © 2006, Oracle. All rights reserved.

Primary Authors: Jeff Gallus, Patrice Daux

Contributors: Steve Muench, Blaise Ribet

Preface

This preface outlines the contents and audience for Oracle Application Development Framework: Tutorial for Forms/4GL Developers (10.1.3.1.0).

The preface contains the following sections:

- Intended Audience
- Structure
- Related Documents

Intended Audience

This tutorial is for Java EE developers who use Oracle Application Development Framework (ADF) to build Web applications.

Structure

The tutorial consists of the following chapters:

Chapter 1: Getting Started

This chapter describes the Service Request scenario and installation of the schema.

Chapter 2: Developing Data Model Objects

This chapter describes how to build the data model object layer for your application by using Oracle ADF Business Components.

Chapter 3: Defining the Page Flow and Standards

This chapter describes how to create the skeleton pages in your JSF application and define the navigation between them.

Chapter 4: Presenting and Packaging the Data Model

This chapter describes how to build and package the data model presentation layer for your application using Oracle ADF Business Components.

Chapter 5: Developing a Simple Display Page

This chapter describes how to create a simple display page at the center of the SRDemo application that enables users to view information about service requests.

Chapter 6: Developing a Master-Detail Page

In this chapter, you develop a master-detail page that shows a service request and its Service Request History rows.

Chapter 7: Implementing Transactional Capabilities

This chapter describes how to build the pages to create a service request. The service request process involves three main pages: one to specify the product and problem, one to confirm the values, and one to commit and display the service request ID. You also create a fourth page, which displays some frequently asked questions about solving some typical product problems.

Chapter 8: Developing a Search Page

This chapter describes how to build a search page. The page contains two sections, one used to specify the query criteria and the other to display the results.

Chapter 9: Developing an Edit Page

This chapter describes how to create a page that enables managers and technicians to edit service requests.

Chapter 10: Deploying the Application to Oracle Application Server 10*g*

In this chapter, you use JDeveloper to create a deployable package that contains your application and required deployment descriptors. You then deploy the package.

Related Documents

For more information about building applications with Oracle ADF, see the following publications:

- *Oracle Application Development Framework Developer's Guide for Forms/4GL Developers 10g Release 3 (10.1.3)*
- *J2EE Application Development for Forms and Designer Developers*

1

Getting Started

This tutorial describes how to build an end-to-end Java EE Web application by using Oracle JDeveloper and Oracle ADF Business Components. The application uses various Java EE technologies, including ADF Faces and JavaServer Faces (JSF); it also uses ADF Business Components (ADF BC). In the tutorial, you learn how to use JSF for the application's user interface and for control of application navigation, as well as how to use ADF Business Components for database interaction and business logic.

This chapter contains the following sections:

Tutorial Scenario: Overview

ServiceCompany is a company that provides service support for large household appliances (dishwashers, washing machines, microwaves, and so on). It handles support for a wide variety of appliances and attempts to solve most customer issues by responding to service requests over the Web.

ServiceCompany has found that over time, customers can resolve most issues after they have the correct information. This approach has been shown to save time and money for both the company and its customers. A service request can be created at the request of a customer, technician, or manager.

Service requests opened by employees can represent any type of internal information associated with a product (examples include product recalls, specific problems with products, and so on).

The Business Problem

ServiceCompany has seen service requests increase but resolutions to them have decreased over the past two quarters. As a consequence, the company has decided to implement a more customer-friendly system and a more efficient and speedy request resolution service.

ServiceCompany wants to be able to provide the same service and response to all customers, regardless of the channel of service request placement and service request type.

Business Goals

ServiceCompany has the following goals:

- Record and track product-related service requests
- Resolve service requests smoothly, efficiently, and quickly
- Manage the assignment and record the progress of all service requests
- Completely automate the service-request process
- Enable managers to assign service requests to qualified technicians
- Enable customers and technicians to log service requests
- Track the technicians’ areas of product expertise

Business Solution

ServiceCompany has decided to implement a new, fully automated system that is built using Oracle Application Development Framework (ADF). This enables highly productive development of a standards-based Java EE application structure. The application server will be Oracle Application Server 10*g*.

The major components of the new application are:

- A customer interface to enable any user (customer, technician, or manager) to add, update, and check the status of service requests

- User interfaces with which the company can create, update, and manage service requests. This includes assigning requests to the appropriate technician and gathering cumulative history information.
- Various reporting tools to ensure timely resolution of service requests
- A user interface with which technicians can update their areas of product expertise

The following process represents the planned flow of a customer-generated service request:

1. A customer issues a request via a Web interface.
2. A manager assigns the request to a technician.
3. The technician reviews the request and then either supplies a solution or asks the customer for more information.
4. The customer checks the request and either closes the request or provides further information.
5. Managers can review an existing request for a technician and (if necessary) reassign it to another technician.
6. Technicians identify products in their area of expertise. Managers can then use this information in assigning service requests.

The technologies to be employed in building the application are as follows:

- The technology employed will be Oracle ADF.
- The data will be stored in Oracle Database 10*g*.
- The data model and business logic will be implemented by using ADF Business Components.
- Databinding (mapping between client components and the business logic) will be provided by Oracle ADF.
- The Web client layer will be built using JSF pages and ADF Faces components.
- Authorization will be based on Java EE container security.

Design Patterns and Architectural Frameworks

A good practice when developing applications is to employ design patterns. Design patterns are a convenient way of reusing object-oriented concepts between applications and between developers. The idea behind design patterns is simple: document and catalog common behavior patterns between objects. Developers can then make use of these patterns rather than re-create them.

In addition to design patterns, developers often use architectural frameworks to build applications that perform in a standard way. One of the frequently used architectural patterns is the Model-View-Controller (MVC) pattern.

In the MVC architecture, the user input, the business logic, and the visual feedback to the user are explicitly separated and handled by three types of objects. Each of these objects is specialized for a particular role in the application:

- The **view** manages the presentation of the application output to the user.
- The **controller** interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
- The **model** manages the data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).

The formal separation of these three components is a key characteristic of a good design.

Another popular design pattern for Java EE applications is the Session Facade pattern. The Session Facade pattern hides complex interactions between the application components from the client's view. It encapsulates the business logic that participates in the application workflow and therefore simplifies the interactions between application components. ADF Business Components manages the relationships between business objects. It also manages the life cycle of these participants by creating, locating (looking up), modifying, and deleting them as required by the application.

In this tutorial, you use the Session Facade design pattern in the application’s model, and you use the MVC architecture through the use of Oracle ADF and JavaServer Faces.

Using This Tutorial

The tutorial is divided into separate chapters, with each chapter building on the previous one. To get the greatest benefit out of the tutorial, you should complete each chapter in the order presented in the tutorial.

If you are only interested in the topics in a specific chapter, a helper application is available. The helper applications represent the successful completion of a chapter. Descriptions on how to use the helper applications and where to find them are available at the beginning of each chapter.

This section describes all of the prerequisite steps that you need to complete before starting to build the application itself.

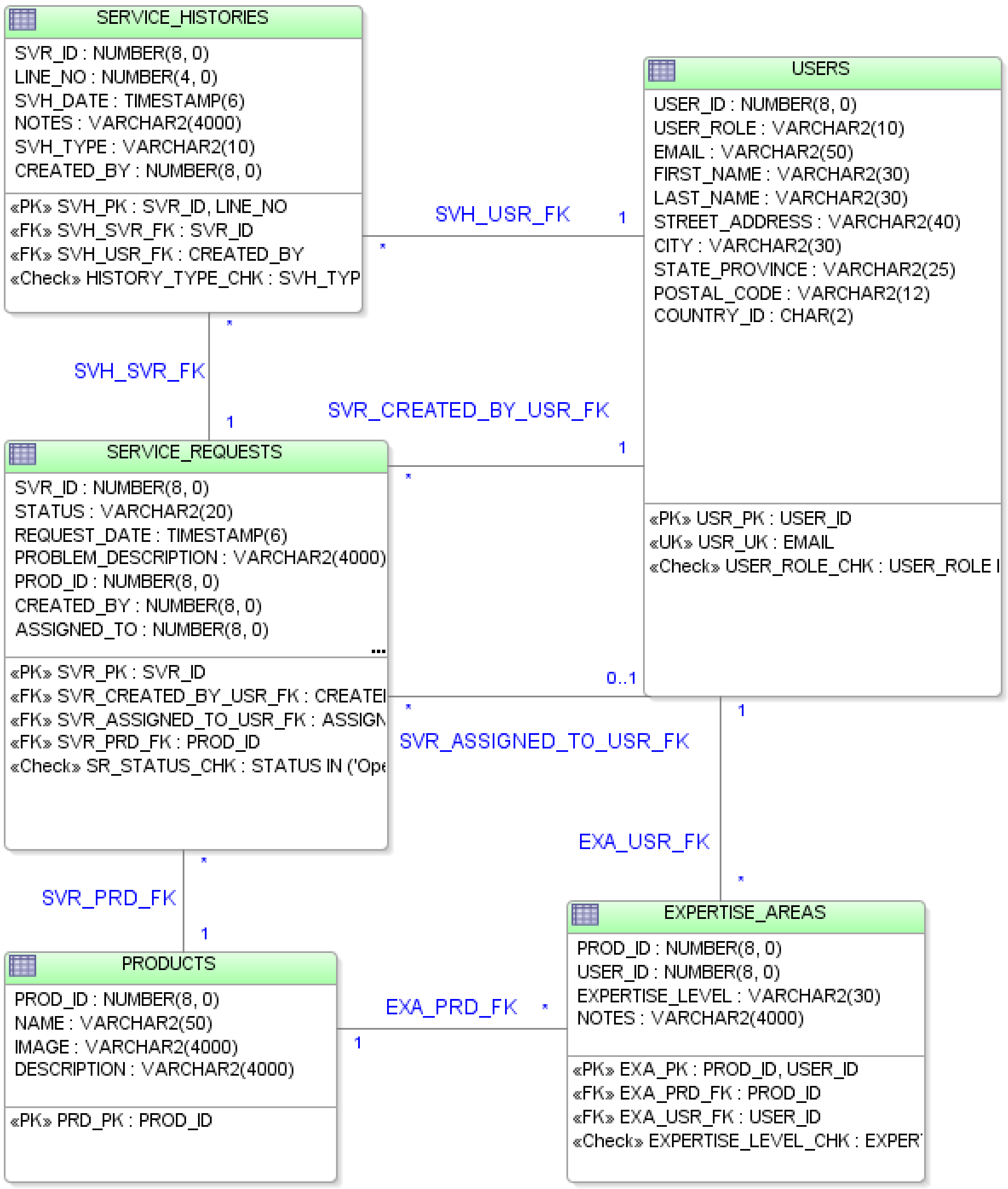
Setting Up Your Environment

You need to prepare your working environment to support the tutorial application. To do this, you perform the following key tasks:

- Prepare to install the schema
- Create the `SRDEMO` schema owner and install the Service Request schema
- Start Oracle JDeveloper 10g Release 3
- Create a JDeveloper database connection
- Define an application and projects for the tutorial

Preparing to Install the Schema

The schema consists of five tables and three database sequences. The tables are diagrammed as follows:



The five tables represent creating and assigning a service request to a qualified technician.

Tables

`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 are stored. An ID uniquely identifies a user.

`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. An artificial ID uniquely identifies all service requests.

`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. An artificial ID uniquely identifies all products.

`EXPERTISE_AREAS`: To better assign technicians to requests, the specific areas of expertise of each technician are defined.

Sequences

`USERS_SEQ`: Generates the next ID for new users

`PRODUCTS_SEQ`: Generates the next ID for new products

`SERVICE_REQUESTS_SEQ`: Generates the next ID for each new service request

Database Triggers

`ASSIGN_SVR_ID`: Assigns the next value from the **`SERVICE_REQUESTS_SEQ`** to the Id for each new service request.

`ASSIGN_PRODUCT_ID`: Assigns the next value from the **`PRODUCTS_SEQ`** to the Id for each new product.

`ASSIGN_USER_ID`: Assigns the next value from the **`USERS_SEQ`** to the Id for each new user.

Download and Extract Setup files

1. Obtain the **`ADFBCTutorialSetup.zip`** file at the following location: http://download.oracle.com/otndocs/products/jdev/10131/ADFBCTutorialSetup10131.zip
2. Unzip the file to a temporary directory (e.g., C:\temp\ADFBCTutorialSetup\) to expose the files that are used to create the three images for the Web pages. For the remainder of the tutorial, this directory is referred to as <tutorial_install>.

Installing the Service Request Schema

The `SRDEMO` user owns the data displayed in the application. Access to an Oracle `SYSTEM` user or equivalent is required to create the user account and to assign the appropriate privileges. The `createSchema.sql` file contains all the commands necessary to create the database user. The `createSchemaObjects.sql` file connects as the `SRDEMO` user and creates all the tables, constraints, and database sequences for the tutorial. The `createSequenceTriggers.sql` file creates the database triggers, which are used with the sequence generators to populate primary keys, when new records are created. Finally, the `populateSchemasTables.sql` file inserts example data into the tables for use during the tutorial.

Caution: For security reasons, it is not advisable to install the ADF tutorial schema into a production database. You may need the assistance of your DBA to access an account with the privilege to create a user.

1. Invoke SQL*Plus and log on as SYSTEM or as another DBA-level user. You may need to ask your DBA to give you an account or to run the scripts for you.
2. In the SQL*Plus window, start the build.sql script from the directory where you unzipped it. For example:

   ```
   SQLPLUS>Start <tutorial_install>\scripts\build.sql
   ```

   After control is returned to the `build.sql` script, the list of the created objects is displayed along with any potential invalid objects. Running these scripts should take less than 30 seconds. You may rerun the `build.sql` script to drop and re-create the `SRDEMO` owner and objects.

Starting Oracle JDeveloper

Follow these instructions to prepare JDeveloper Studio.

Note: If you have not already installed JDeveloper 10*g* Release 3, then do so before proceeding with the next tutorial steps. Windows Explorer

1. In Windows Explorer, navigate to the directory where JDeveloper is installed. In the root directory, double-click the **JDeveloper.exe** icon to invoke JDeveloper. If this is the first time JDeveloper has been run, a "Do you wish to migrate" window appears.
2. Click **No** to continue.
3. On startup, a "Tip of the Day" window appears. These tips are things you can do to make development more productive. Click **Close** when you've finished looking at the tips.

Creating a Database Connection to Access the `SRDEMO` Schema

Follow these instructions to create a new database connection to the Service Request schema using the `SRDEMO` user.

Note: In the tutorial, the database connection is named **`SRDemo`**. The name of the connection does not affect the ability to complete the tutorial. However, we strongly recommend using the naming conventions described in all the steps. In doing so, it is easier to follow the instructions.

1. In JDeveloper, choose **View > Connections Navigator**.
2. Right-click the **Database** node and choose **New Database Connection**.
3. Click **Next** on the Welcome page.
4. In the Connection Name field, type the connection name **`SRDemo`**. Then click **Next**.
5. On the Authentication page, enter the values as shown in the following table. Then click **Next**.

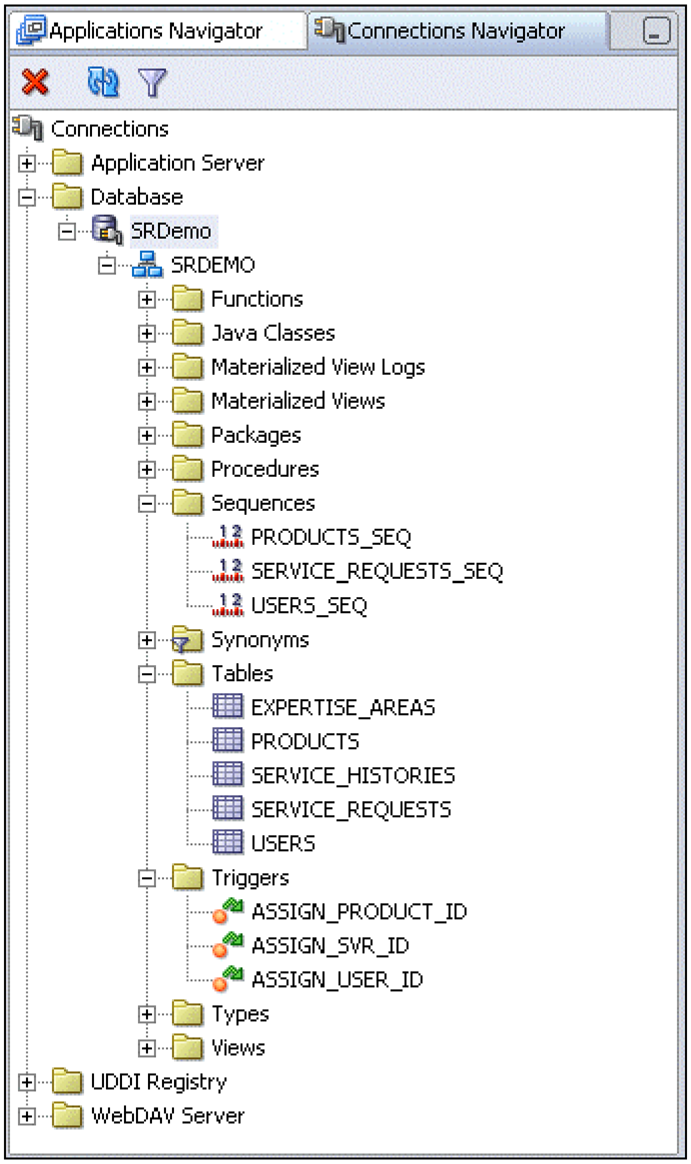
Field	Value
Username	`srdemo`
Password	`oracle`
Deploy Password	Select the check box.

6. On the Connection page, enter the following values. Then click **Next**.

Field	Value
Host Name	`localhost` This is the default host name if the database is on the same machine as JDeveloper. If the database is on anther machine, type the name (or IP address) of the computer where the database is located.
JDBC Port	`1521` This is the default value for the port used to access the database. If you do not know this value, check with your database administrator.
SID	`ORCL` This is the default value for the SID that is used to connect to the database. If you are using the Oracle Express 10*g* database, the value should be XE. If you do not know this

value, check with your database administrator.

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.
9. 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 above.



Defining an Application and Its Projects in JDeveloper

In JDeveloper, you work within projects that are contained in applications.

An application is the highest level in the control structure, serving as a collector of all the subparts of the application. When you open JDeveloper, the applications, which were opened when you last closed JDeveloper, are opened by default.

A JDeveloper project is an organization structure used to logically group related files. In a Java EE application, a project typically represents a part of the application architecture, such as the data model or a part of the client.

You can add multiple projects to your application to easily organize, access, modify, and reuse your source code.

Before you create any application components, you must first create the application and its projects. Use the following procedure to create the new SRTutorial application and its projects.

Note: Do not include special characters in the project name such as periods. If you include special characters, errors appear when you attempt to compile your project.

1. To create an application, click in the Applications Navigator, right-click the **Applications** node, and select **New Application** from the shortcut menu.
2. In the Create Application window, enter the following values:

Field	Value
Application Name	`SRTutorialADFBC`
Directory name	`<jdev_install>\jdev\mywork\SRTutorialADFBC` Keep the default value. If you used the default directory structure, then your path should match this value. The directory is created for you in the specified path, which represents the application.
Application Package Prefix	`oracle.srtutorial` This value becomes the prefix for all Java package names. You can override it later if necessary.
Application Template	`No Template [All Technologies]` In this tutorial, you access many of JDeveloper's technologies. New templates can be created and added to restrict the technologies that are available during development.

3. Click **OK**.

4. The Create Project dialog box appears. Set the values as follows:

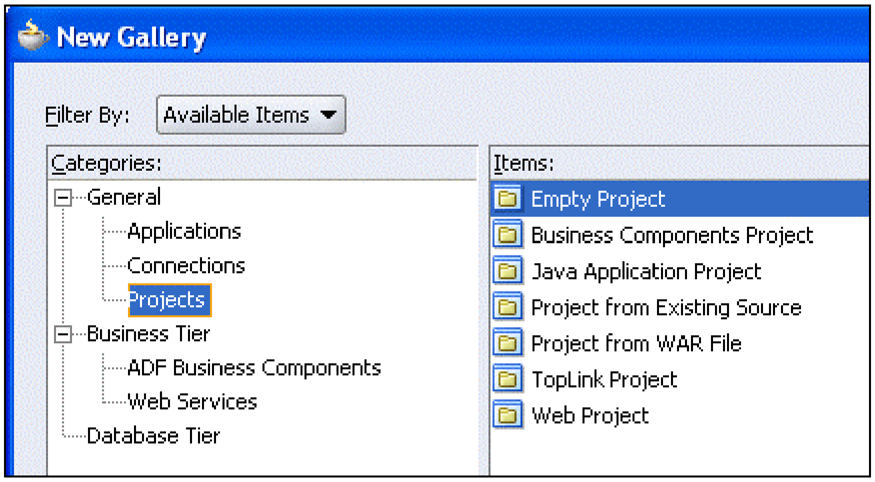
Field	Value
Project Name	`DataModel`
Directory name	`<jdev_install>\jdev\mywork\SRTutorialADFBC\DataModel` Keep the default value. If you used the recommended directory structure, then your path should match this value. The directory is created for you in the specified path to contain the project's files.

5. Click **OK**.
The DataModel project represents the data model tier of your application. You build the components of this project in the next chapter by using Oracle ADF Business Components.

6. Create another Project at the same level as the DataModel project. In the Applications Navigator, right-click the **SRTutorialADFBC** node and select **New Project**.

7. The New Gallery is invoked. Verify that **Empty Project** is selected. Then click **OK**.

   The new project becomes the child of a selected application node. To find out more about the types of projects and when they should be used, search Help for "Projects Category."



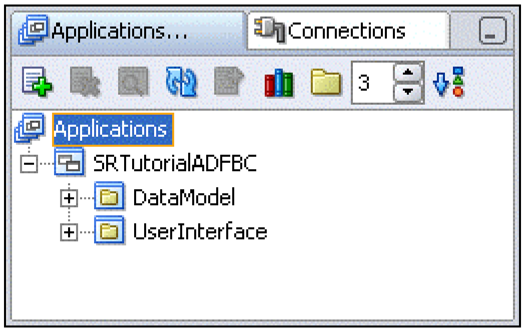
8. In the Create Project pane, set the values as follows.

Field	Value
Project Name	`UserInterface`
Directory name	`<jdev_install>\jdev\mywork\SRTutorialADFBC\UserInterface`

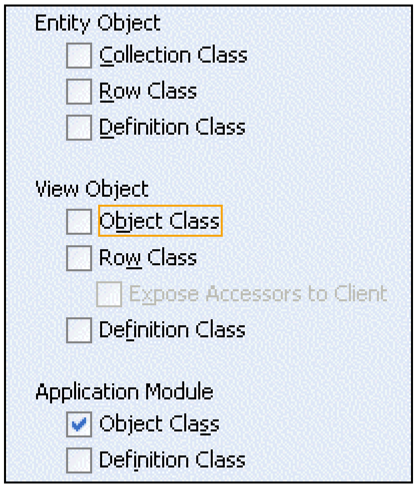
Keep the default value. If you used the recommended directory structure, your path should match this string.

9. Click **OK**. The UserInterface project represents the remainder of the application and contains the files that you create for the user interface in later chapters.
10. Double-click the new **UserInterface** project and select the **Dependencies** node.
11. Select the check box associated with the **DataModel** project. This enables the UserInterface project to access objects created in the DataModel project.
12. Select the **Project Content** node. At the bottom of the panel, set the Default Package to **`oracle.srtutorial.userinterface`**. Then press **OK**. This enables you to better manage your classes and files.
13. Double-click the **DataModel** project and select the **Project Content** node.
14. At the bottom of the **Project Content** panel, set the Default Package to **`oracle.srtutorial.datamodel`**. Then, press **OK**. This standard enables you to better manage your data model classes and files.

Your Applications Navigator should look like the following screenshot. You have one more environmental setting to change and then you are ready to create application components for the tutorial.



15. In the JDeveloper menu bar, select **Tools | Preferences**.
16. In the Preferences window, select the **Business Components** node to specify code generation preferences.
17. Deselect the **Entity Object:Row Class** and **View Object: Object Class** check boxes. These options are not necessary since we won't require any custom Java code for most components in the tutorial. With ADF Business Components, the declarative information saved in each component's XML definition file can supply all the information we'll need for most aspects of this application. Since our application will have some custom code in the application module component we create, keep the **Application Module: Object Class** check box selected. Note that these are only the *default* settings for which Java classes get automatically generated. We can choose to generate custom Java classes for any appropriate component at a later time on a component-by-component basis as needed.



Entity Object
- [ ] Collection Class
- [ ] Row Class
- [ ] Definition Class

View Object
- [ ] Object Class
- [ ] Row Class
  - [ ] Expose Accessors to Client
- [ ] Definition Class

Application Module
- [x] Object Class
- [ ] Definition Class

18. Press **OK** and **Save All** your work.

Note: As part of the tutorial setup files, we have included a zipped application for each chapter with the steps completed successfully.

If your application does not compile correctly or if it has other errors, you can use these applications as a starting point for the following chapter. There are instructions on how to use these starter applications at the beginning of each chapter.

Summary

In this chapter, you carried out all of the prerequisite steps that you need to complete before starting to build the application. You performed the following key tasks:

- Prepared the tutorial schema setup
- Installed the Service Request schema
- Started JDeveloper
- Created a JDeveloper database connection
- Defined an application and its projects in JDeveloper

2

Developing Data Model Objects

This chapter describes how to build a data model, which represents the database schema, using ADF Business Components.

The chapter contains the following sections:

- Introduction
- Creating the Data Model Using Entity Objects
- Refining the Entity Objects
- Refining the Associations Between Entities
- Incorporating Validation into the Model
- Grouping the Data
- Summary

Introduction

ADF Business Components technology provides a framework-based method for developing Java EE business services. ADF Business Components governs interaction between the rest of the application and the data stored in the data source, providing validation, specific services, and other business logic. To leverage a robust implementation of the numerous design patterns you need to:

- Declaratively design master/detail UI data model, including only data needed by client for optimum performance
- Synchronize pending data changes across multiple views of data
- Declaratively enforce required fields, primary key uniqueness, data precision/scale, and foreign key references
- Consistently apply prompts, tooltips, format masks, and error messages in any application
- Enforce best-practice interface-based programming style
- Easily capture and enforce both simple and complex business rules, programmatically or declaratively

Entity objects are the foundation of the ADF Business Components technology. Entity objects contain attributes, business rules and persistence information that apply to a specific part of your business model. Entities are used to define a logical structure of your business, such as products, service requests, service histories, etc. They can also define physical items like warehouses, users, and equipment. For example, `User` is an entity object.

- An entity object is based on a data source; this example is based on the Users table.
- An entity's attributes map to the columns of the data source; the User entity has attributes called `ID`, `Name`, `Role`, and `Email` that map to the corresponding columns of the Users table.
- You can attach validation rules to an entity object; the `Role` attribute has a validation rule that restricts status values to a list of valid values.

You perform the following key tasks in this chapter:

- Create default Business Component entity objects
- Create attributes in the entity objects
- Create relationships between entity objects
- Create validation rules

Note: If you did not successfully complete Chapter 1, you can use the end-of-chapter application that is part of the tutorial setup:

1. Create a subdirectory named `Chapter2` to hold the starter application. If you used the default settings, it should be in `<jdev-install>\jdev\mywork\Chapter2.`
2. Unzip `<tutorial-setup>\starterApplications\SRTutorialADFBC-EndOfChapter1.zip` into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRTutorialADFBC application workspace.
4. Select **File > Open**, and then select `<jdevinstall>\jdev\mywork\Chapter2\SRTutorialADFBC\SRTutorialADFBC.jws`. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 1.

Creating the Data Model Using Entity Objects

Entity objects represent your business domain objects. They map to database tables and implement data persistence. They provide hooks for validation logic, security logic, creation and deletion logic and events triggered by data modification. In this section you map the tables in your database to ADF Business Component entity objects.

Creating Entity Object Mappings for the Database Objects

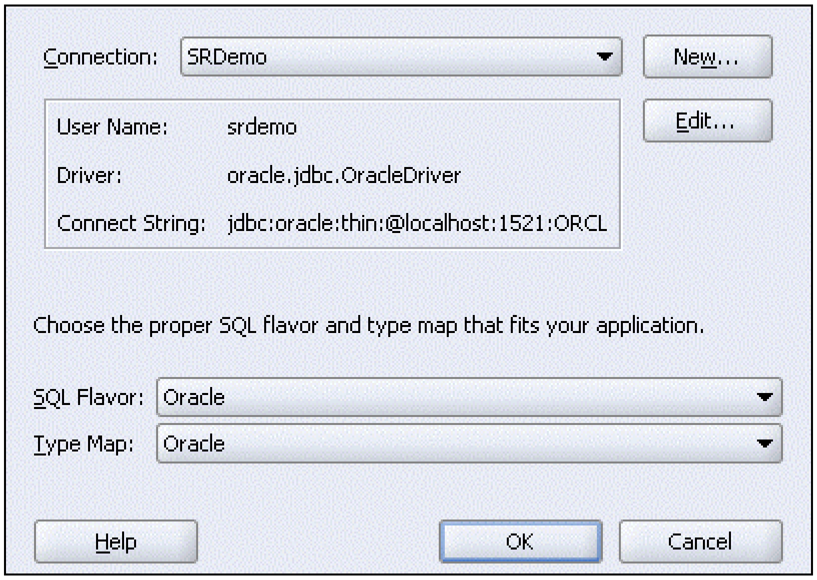
Note: Underlying database schemas sometimes change after the entity objects are created. In such cases, you can synchronize them with the database using features in Oracle JDeveloper.

In this step, you reverse-engineer entity objects from existing database tables in the `SRDEMO` schema.

1. In the Applications Navigator, right-click the **DataModel** project and select **New**.
2. In the New Gallery, expand the **Business Tier** node, select **ADF Business Components**, and choose **Business Components from Tables** in the Items list.
3. Click **OK**.

Every project using Business Components needs to be initialized. Initializing the project involves defining a database connection to be used during the creation of any business component.

4. In "Initialize the Business Component Project“ dialog ensure the connection is set to **`SRDemo`**. The username and connect string should be set to the values from the Database Connection.



Connection: SRDemo

User Name: srdemo

Driver: oracle.jdbc.OracleDriver

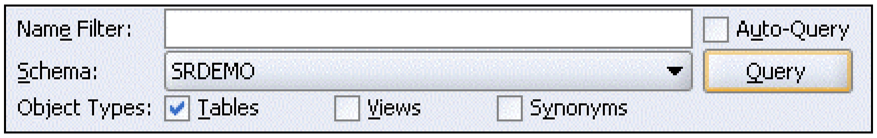
Connect String: jdbc:oracle:thin:@localhost:1521:ORCL

Choose the proper SQL flavor and type map that fits your application.

SQL Flavor: Oracle

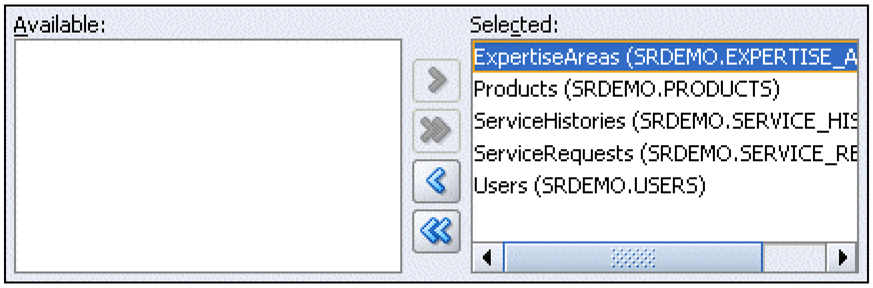
Type Map: Oracle

5. Click **OK**.
6. In the "Create Business Components from Tables – Welcome " page of the Wizard, click **Next**.
7. In step 1 of the wizard, set the values used as default during entity object creation. Set the Package property to **`oracle.srtutorial.datamodel.entities`** and ensure the Schema is set to **`SRDEMO`**.
8. Click the **Query** button next to the Schema name, to query the available tables from the database.



Name Filter:
Schema: SRDEMO
Object Types: [✓ Tables] Views Synonyms

9. In the Available list, select all the tables and press the **>** button to select them for entity creation.



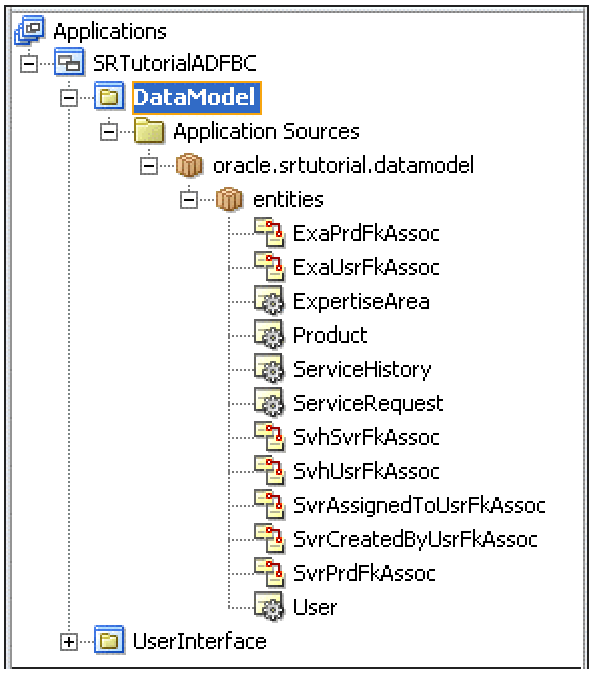
Available:

Selected:
ExpertiseAreas (SRDEMO.EXPERTISE_A
Products (SRDEMO.PRODUCTS)
ServiceHistories (SRDEMO.SERVICE_HIS
ServiceRequests (SRDEMO.SERVICE_RE
Users (SRDEMO.USERS)

10. Notice the default entity name is derived from the table name. Since each entity object represents a single logical business object, and a single row in its underlying table, it is best practice to give your entity objects singular names to reflect this. To change the entity name, highlight it in the Selected list and change the value in the Entity Name property. Set the Entity Name property to the values in the table below.

Table Name	Entity Name
EXPERTISE_AREAS	`ExpertiseArea`
PRODUCTS	`Product`
SERVICE_HISTORIES	`ServiceHistory`
SERVICE_REQUESTS	`ServiceRequest`
USERS	`User`

11. The Create Business Component Wizard can be used to create other components besides Entity Objects. For now, press **Finish** to create the entities. Other components will be created in later lessons.
12. The Applications Navigator should display all the default entities and associations and look like the following image.



Notice that in addition to seeing one entity object for each table we selected in the "Business Components for Tables" wizard, we also see components representing the referential integrity constraints between tables as corresponding associations between entity objects. While we can use the Application Navigator's "Sort by Type" toolbar toggle button to group components by type, we can further improve our application organization by Refactoring these associations into their own separate package.

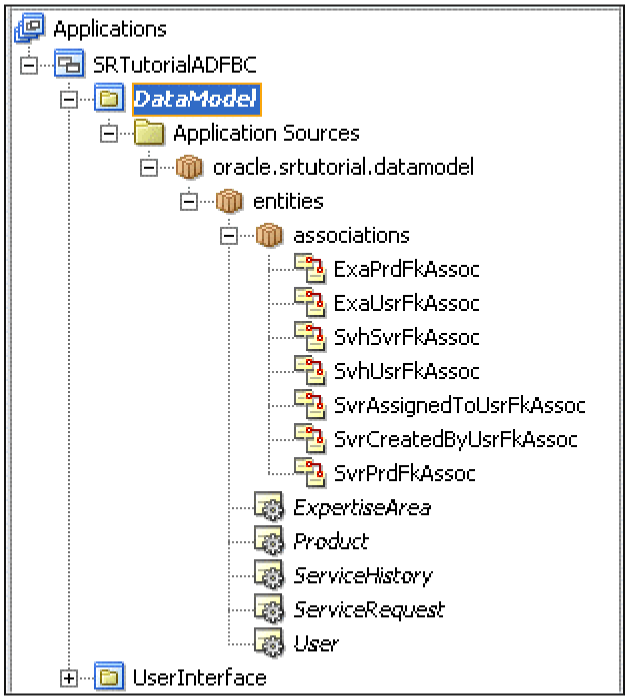
13. In the Applications Navigator, select all seven Associations.

14. Right click and from the context menu, select **Refactor → Move**.

15. In the Move business Components dialog, append to the Package property `.associations` . The complete package property should be `oracle.srtutorial.datamodel.entities.associations.`

16. When prompted, create the new package.

The resulting hierarchy in the Applications Navigator should look like the following image.



Refining the Entity Objects

In this section, you refine the definitions of the default entities you created above. Many features of entity objects are controlled through declarative settings that you modify using the ADF Business Components editors in JDeveloper. This includes information about the names and datatypes of the entity object's data properties, as well as other settings that control the runtime behavior. When required, you can generate related Java files to hold custom code for an entity object that gets "triggered" at the appropriate time to perform programmatic defaulting, validation, or other application-specific business functionality. By encapsulating behavior into the entity object layer, all pages you build that interact with the same business domain objects will have consistent runtime behavior and formatting.

Entity object attributes (entity attributes, for short) are named properties that describe the values that instances of an entity object might hold, such as:

- The Java type of the attribute
- Constraints, such as PRIMARY KEY, NOT NULL or UNIQUE
- Miscellaneous simple functionality, such as static default values

Refine the Attribute Definitions

You can use the Entity Object Editor to modify the attribute definitions. You can modify the general attribute properties, and the attribute's control hints.

In the next few steps, you modify the Attributes of the `ServiceRequest` object:

1. In the Applications Navigator, double click the **ServiceRequest** entity object.
2. Select the **Status** attribute and set the Default property to **Open**. This causes the value of the Status attribute in newly-created service requests to default to the string `Open`.

3. Select the **RequestDate** and select the History Column check box.
4. In the dropdown, select **created on**. These two steps indicate that the `RequestDate` attribute should default to the database date when the service request is first created.

Set the Control hints for the ServiceRequest attributes. The values set here are automatically leveraged in the presentation layer for any page that presents service request information.

5. Select the Control Hints tab and set the **Label Text** to property to the values in the table below.

Attribute	Label Text Value
`SvrId`	`Request Id`
`Status`	`Status`
`RequestDate`	`Requested On`
`ProblemDescription`	`Problem`
`ProdId`	`Product`
`CreatedBy`	`Created By`
`AssignedTo`	`Assigned To`
`AssignedDate`	`Assigned On`

6. Set the Type and Format properties for the following attributes to the values in the table below.

Attribute	Format Type	Format Value
`RequestDate`	`Simple Date`	`MM/dd/yyyy HH:mm`
`AssignedDate`	`Simple Date`	`MM/dd/yyyy HH:mm`

Note: Date and number format masks in the Java EE world follow the existing Java standards**.**

7. Set the ProblemDescription display width to **60**.
8. Click the **OK** button to accept all the modifications to the `ServiceRequest` entity.

Set attribute properties and control hints for `ServiceHistory`'s attribute.

9. Select the Control Hints tab and set the **Label Text** to property to the values in the table below.

Attribute	Label Text Value
`SvrId`	`Request Id`

`LineNo`	`Line #`
`SvhDate`	`Date Created`
`Notes`	`Comments`
`SvhType`	`Type`
`CreatedBy`	`Created By`

10. Define Control Hints format for the `SvhDate` attribute. Set the Format Type to **Simple Date** and the Format to **MM/dd/yyyy HH:mm**.
    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.
11. In the SvhDate Entity Attribute tab, select the **History Column** check box and select the **created on** value in the drop down.
12. Click **OK**.
13. **Save All** your work.

All the basic data model entity and attribute definitions are now created. The next step is to refine the associations between the entity objects.

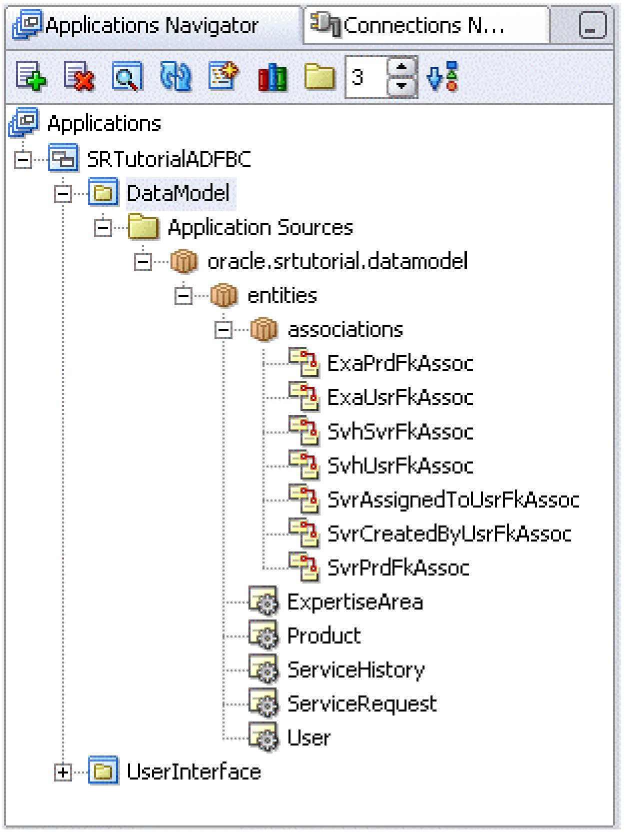
Refining the Associations Between Entities

Oracle ADF associations are business components that define a relationship between two Oracle ADF entity object definitions (the "source" and "destination" entity objects) based on sets of entity attributes (the "source" and "destination" attributes) from each. These can range from simple one-to-many relationships based on foreign keys to complex many-to-many relationships. For example, associations can represent:

- The one-to-many relationship between a user and all service request placed by that user
- The one-to-one relationship between a product and its extended description (if these are represented by separate entity objects)
- The many-to-many relationship between products and the users that currently possess skills with that product. (In our application, that many-to-many relationship is resolved by the creation of the `EXPERTISE_AREAS` table.

In the tutorial, you update the already created association names to something more descriptive.

1. In the Applications Navigator, expand **oracle.srtutorial.datamodel → entities → associations** to expose all the default associations created by the wizard.



The default naming convention for inherited reverse-engineered associations follows the names of the referential integrity constraints.

Recalling the database schema diagram from Chapter 1, we see that the constraint between the `PRODUCTS` and `EXPERTISE_AREAS` table is named `EXA_PRD_FK`. The default association name is created by converting this underscore-separated, uppercase name into a more Java-friendly mixed-case like `ExaPrdFk`, and then adding the `Assoc` suffix.

- Short name of the source entity. In the above image `Exa` represents the ExpertiseArea entity.
- Short name of the destination entity. In the first association in the list `Prd` represents the Product entity.
- The key phrase `FkAssoc` to represent the association supports a foreign key

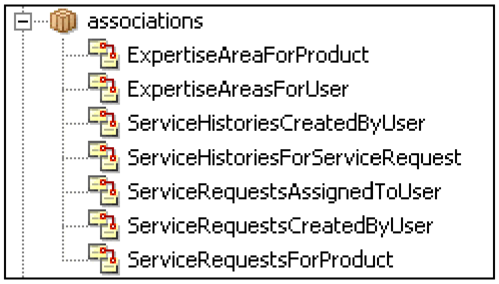
While there is a structure to the association names, it's good practice to rename them to something more descriptive.

2. Right click the first association **ExaPrdFkAssoc** and from the context menu select **Refactor → Rename**.
3. In the Rename Association dialog change the name to `ExpertiseAreasForProduct`.
4. Press **OK**.
5. Modify the remaining associations to more descriptive names using the table below.

Old Name	New Name

ExaUsrFkAssoc	`ExpertiseAreasForUser`
SvhSvrFkAssoc	`ServiceHistoriesForServiceRequest`
SvhUsrFkAssoc	`ServiceHistoriesCreatedByUser`
SvrAssignedToUsrFkAssoc	`ServiceRequestsAssignedToUser`
SvrCreatedByUsrFkAssoc	`ServiceRequestsCreatedByUser`
SvrPrdFkAssoc	`ServiceRequestsForProduct`

6. When compete, the associations should look like the image below.



```
associations
    ExpertiseAreaForProduct
    ExpertiseAreasForUser
    ServiceHistoriesCreatedByUser
    ServiceHistoriesForServiceRequest
    ServiceRequestsAssignedToUser
    ServiceRequestsCreatedByUser
    ServiceRequestsForProduct
```

Incorporating Validation into the Model

Oracle ADF Business Components provides a number of predefined validation rule classes that allow you to add business logic to entity objects without writing a single line of code. These built-in validation rules are implemented declaratively (in the Entity Object Editor) and each time you make use of one, an entity object's XML file records that validate rule "usage".

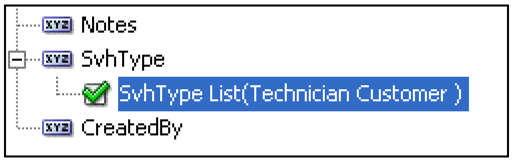
You can add a validation rule to an entity attribute by selecting the Validation node in the Entity Object Editor. There are several types of validation rules, allowing you to validate your attribute against a single value, a list of valid values, or a minimum, maximum, or range of values; alternatively, you can write a method to specify your own validation rules.

In the next steps you incorporate a validation list into the data model. The List Validator is designed for validating an attribute against a relatively small set of values. First, `ServiceHistory` has a `SvhType` attribute, which should be restricted to two values: `Technician` or `Customer`. The type attribute can then be used to restrict what users see what records.

1. In the Applications Navigator, double click the **ServiceHistory** entity object.
2. In the Entity Object Editor, select the **Validation** node.
3. In the Declared Validation Rules: pane, select the **SvhType** attribute.
4. Click the **New** button.
5. In the Edit Validation Rule dialog, set the following properties using the table below.

Property	Value
`Rule`	`List Validator`
`Attribute`	`SvhType`
`Operator`	`In`
`List`	Literal values
`List of Values`	`Technician, Customer` Each literal value should be on its own line.
`Error Message`	The value for Note Type must be `Technician` or `Customer`

6. When complete, press **OK**. A nested validation rule displays within the `SvhType` attribute. OK should be bold so should be the one in step 7.



7. Click **OK** to complete adding validation to the `ServiceHistory` entity object.

A final step is to ensure a default-generated value is placed in the Service Request identifier as a primary key. In the next couple of steps, add a sequence generator to the SvrId attribute.

8. In the Applications Navigator, double click the **ServiceRequest** entity object.
9. In the Entity Object Editor, expand the **Attributes** node and select the **SvrId** attribute.
10. Set the Type property to **DBSequence**. This indicates that the id of each service request will be populated by a database sequence (`SERVICE_REQUESTS_SEQ`) using a database trigger (`ASSIGN_SVR_ID`).
11. Save your work.

Grouping the Data

Exposing the data model to the client application requires an application module. An application module performs a specific application task—for example, handling online orders or processing customer information. An application module is automatically exposed as data control and has these main characteristics:

- Contains a set of optionally master/detail related queries representing the application data model.
- Tailored to the presentation needs of a client interface (such as a sequence of web pages allowing the user to accomplish a task).

- It provides custom service methods, accessible by local and remote clients, which implement the application module behavior.
- It manages the connection to the database and keeps track of all changes that affect data in the database.
- It can be easily reused in the business logic tiers of other applications.

In the SRTutorialADFBC application there is only one application module, which is available to everyone using the application. In the tutorial, you will iteratively enhance the data model for this Application Module as the pages are developed.

In the next steps, create the SRPublicService application module.

1. In the Applications Navigator, right click **oracle.srtutorial.datamodel** and from the context menu, select **New Application Module**.
2. In Step 1: Name, of the wizard, use the values form the table below to complete the step.

Property	Value
`Package`	`oracle.srtutorial.datamodel`
`Name`	`SRPublicService`

3. Press **Next** then **Finish** to complete the Application Module.

Summary

In this chapter, you built the data model for your application. To accomplish this, you performed the following key tasks:

- Created the data model using entity objects
- Refined the associations between entities
- Refined properties for attributes
- Incorporated validation in the data model
- Created a container to hold the data model

3

Defining Page Flow and Standards

This chapter describes how to create outline definitions for each of the pages in your JSF application and specify the navigation between them. You then create and employ standards for the user interface.

The chapter contains the following sections:

- Introduction
- Creating a JSF Navigation Model
- Creating Pages on the Diagram
- Linking the Pages Together
- Providing for the Translation of the User Interface
- Creating a Standard Look and Feel
- Summary

Introduction

In the previous chapter, you built the entity objects and application module for the Service Request application. Ensure that you have successfully built the data model, as described in that chapter, before you start to work on this one.

In this chapter, you start to work on the user interface. You use the JSF Navigation Modeler in JDeveloper to diagrammatically plan and create your application's pages and the navigation between them. You perform the following key tasks:

- Creating a page-flow diagram
- Defining the navigation rules and navigation cases between the pages
- Providing for translation of the User Interface
- Creating a Standard Look and Feel

Note: If you did not successfully complete Chapter 2, you can use the end-of-chapter application that is part of the tutorial setup:

1. Create a subdirectory named **`Chapter3`** to hold the starter application. If you used the default settings, it should be in `<jdev-install>\jdev\mywork\Chapter3`.
2. Unzip **`<tutorial-setup>\starterApplications\SRTutorialADFBC-EndOfChapter2.zip`** into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRTutorialADFBC application workspace.
4. Select **File > Open**, and then select **`<jdevinstall>\jdev\mywork\Chapter3\SRTutorialADFBC\SRTutorialADFBC.jws`**. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 2.

Creating a JSF Navigation Model

With the JSF Navigation Modeler, you can visually design your application from a bird's-eye view. In the following steps, you open a new empty diagram and add a title to it:

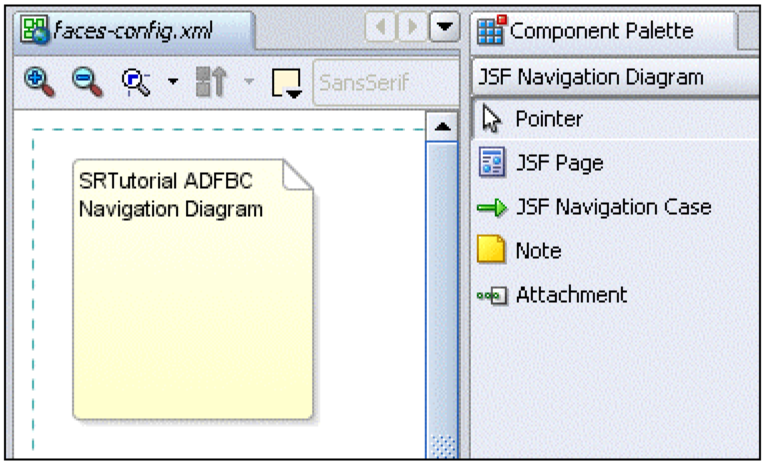
1. In the Applications Navigator, right-click the **UserInterface** node, and select **New** from the short cut menu.
2. In the New Gallery, expand the **Web Tier** node and select **JSF** in the Categories pane.
3. Choose **JSF Page Flow & Configuration (faces-config.xml)** from the Items pane. Then click **OK**.

   This choice adds a JSF configuration file to your project. You edit the JSF configuration file by using the JSF Navigation Modeler to specify navigation rules between the pages of your application.
4. In the Create JSF Configuration File dialog box, accept the default name **`faces-config.xml`** and create it in the default directory location. Click **OK** to continue.

   The empty diagram opens. Notice the Component Palette to the right of the diagram editor. You use this to create components for the JSF Navigation Model.

   Notice also the four tabs at the bottom of the diagram editor screen. The default view is the Diagram view, where you can model and create the pages in your application. Clicking the Overview tab shows a console-type interface that enables you to register any and all types of configurations into your `faces-config` file, including managed beans, navigation rules, and other items such as custom validators, converters, and so on. The Source tab enables you to edit the generated XML code directly. JDeveloper automatically synchronizes the different views of your JSF navigation. Finally, the History tab shows a history of recent changes.
5. Add a title to your diagram. Select **Note** from the Component Palette, where the JSF Navigation Diagram choices are listed, and then drag it to the upper-left portion of your diagram.

   It is good practice to provide annotation to a diagram, and the general Note component is useful for this purpose.
6. Type **`SRTutorial ADFBC Navigation Diagram`** in the box on the diagram, and then click elsewhere in the diagram to create the note. The note serves as your diagram's title.



Note: The steps in this section showed you how to explicitly create a new page flow diagram. You need to do this because you are just starting to build the Web application and have not yet created any pages. If you don't create a `faces-config.xml` file explicitly, JDeveloper creates one automatically when you add a JSF page to your project. Then you simply need to double-click it in the Navigator to open it in the diagram editor.

Creating Pages on the Diagram

There are two ways to add a page to a page flow diagram. If you have already created some JSF pages, you can drag them from the Navigator to the diagram. Alternatively, you can create the pages directly in the diagram.

You have not yet created any pages for the SRTutorialADFBC application. In the following steps, you create placeholders for pages in the application. (You define the pages fully in subsequent chapters of the tutorial.)

1. Select **JSF Page** in the Component Palette, and click where you want the page to appear in the diagram.

   An icon for the page is displayed on the diagram. At this stage, the icon initially has a yellow warning over it to remind you that it is simply a placeholder rather than a fully defined page.

2. Click the icon label and type **`/app/SRList.jspx`** as the page name.

   Note: The JSP standard supports creating pages that are well-formed XML documents. These can be especially handy because they allow the page's source to be easily parsed using XML tools, and they encourage the best practice of not mixing code into your page. JDeveloper supports working with either *.jsp pages or these XML-based *.jspx pages.

The SRList page is at the center of the application. It is a page where all users (customers, technicians, and managers) can browse existing service requests.

The page name requires an initial slash so that it can be run. If you remove the slash when you type the name, it is reinstated. Prefixing the name of the page by **`/app`** indicates the directory location where the page is to be stored.

You can create the detailed pages either iteratively (as you develop the page flow diagram) or at a later stage.

For the purposes of the tutorial, you create placeholders for all the pages in the application at this point and build the detailed pages in later chapters.

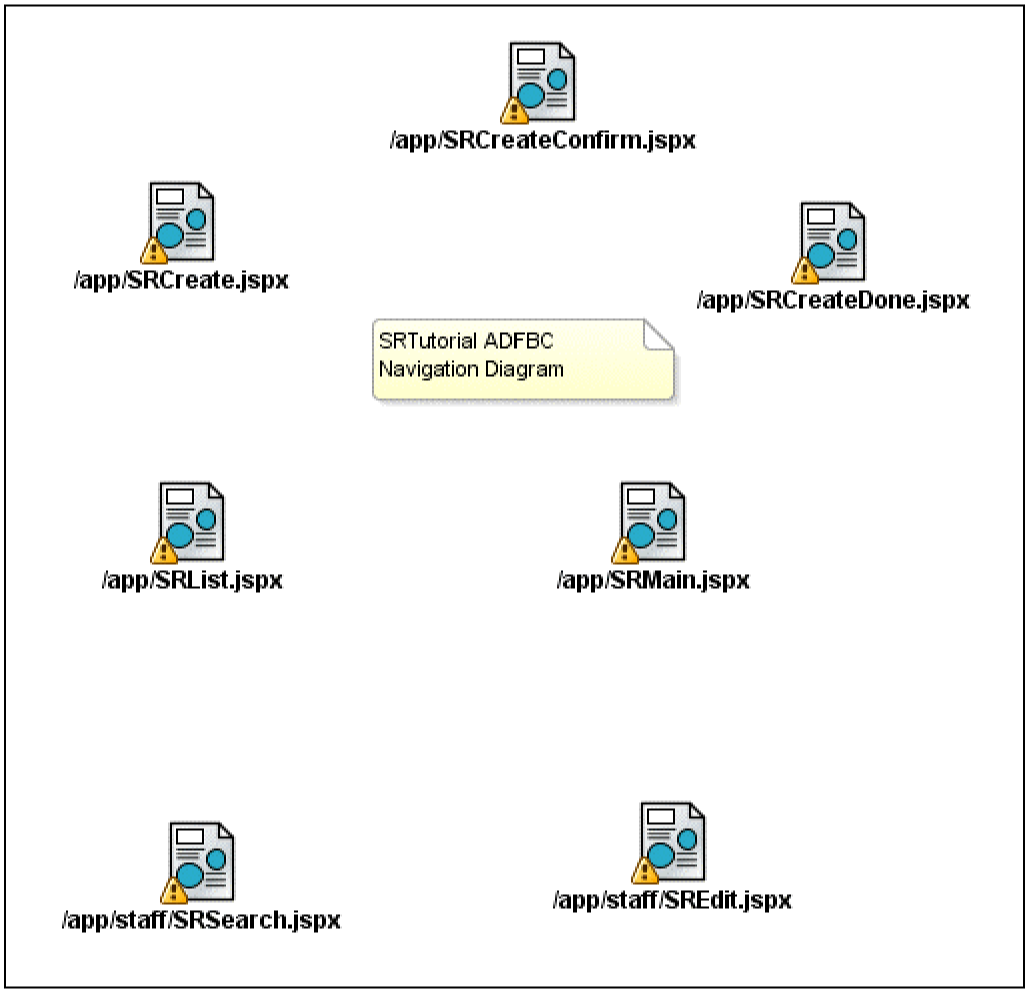
3. Repeat steps 1 and 2 to create five more page placeholders in the `/app` directory:

Page Name	**Page Purpose**
`/app/SRMain.jspx`	Enables users to add extra information to an existing service request
`/app/SRCreate.jspx`	Enables users to create a new service request
`/app/SRCreateConfirm.jspx`	Confirmation screen for new requests
`/app/SRCreateDone.jspx`	Final screen in the creation of a new request

4. Then create the following two pages that only staff-members will have access to (once we enable Java EE security in a later chapter) in the `/app/staff` directory:

Page Name	**Page Purpose**
`/app/staff/SREdit.jspx`	Enables managers and technicians to amend service requests
`/app/staff/SRSearch.jspx`	Enables users to search for a service request

These screens make up the SRTutorialADFBC application. When you have finished, your diagram should look like the following screenshot. If it does not, you can drag the pages to make your diagram similar.



Linking the Pages Together

Now that you have created placeholders for the application's pages, you need to define how users navigate between them. JSF navigation is defined by a set of rules for choosing the next page to be displayed when a user clicks a UI component (for example, a command button). These rules are defined in the JSF configuration file, which is maintained automatically for you as you work with the JSF navigation diagram. There may be several ways in which a user can navigate from one page, and each of these is represented by a different navigation case.

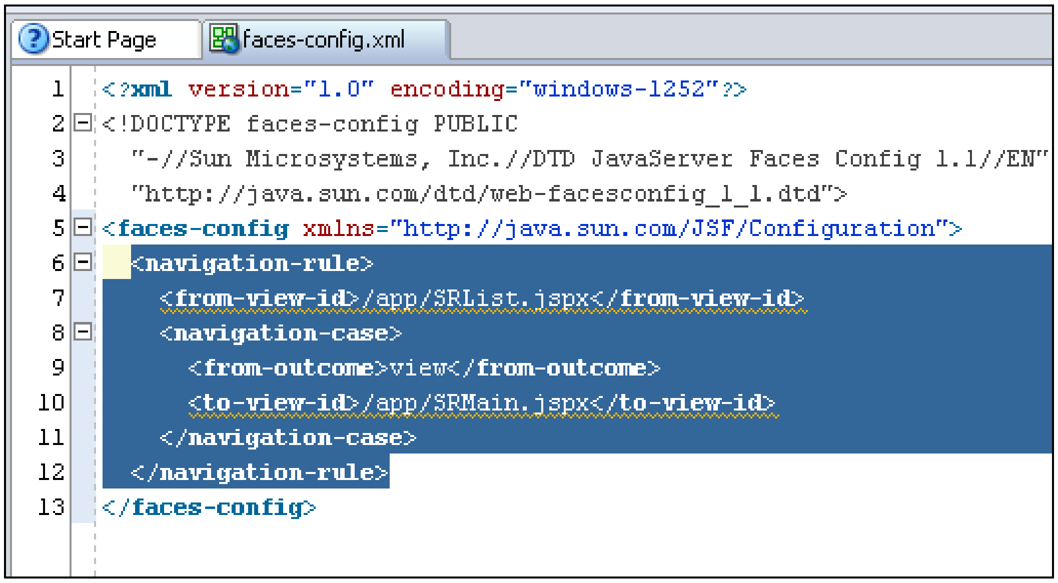
For the SRTutorialADFBC application, you start by drawing simple navigation cases on the page flow diagram.

Add a link to enable a user to navigate from the SRList page to the SRMain page:

1. Select **JSF Navigation Case** in the Component Palette.
2. Click the icon for the source JSF page (**SRList**), and then click the icon for the destination JSF page (**SRMain**) for the navigation case.

   The navigation case is shown as a solid line on the diagram, and a default label ("success") is shown as the name of the navigation case.

3. Modify the default label by clicking it and typing **view** over it. We'll see later how to declaratively tie the (View) button to the "view" navigation rule so that clicking on the (View) button takes the user to the SRMain page as reflected by the navigation line in the diagram.
4. Click the **Overview** tab at the bottom of the screen. Click **Navigation Rules** in the left table. Notice that the rule you just created in the diagram is listed in the table.
5. To understand the syntax of the navigation rule you created, click the **Source** tab to see the XML code for the rule. The following images show you the rule in the source:



```
<?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">
  <navigation-rule>
    <from-view-id>/app/SRList.jspx</from-view-id>
    <navigation-case>
      <from-outcome>view</from-outcome>
      <to-view-id>/app/SRMain.jspx</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

The `<from-view-id>` tag identifies the source page; the `<to-view-id>` tag identifies the destination page; the `<from-outcome>` tag identified the logical name of the navigation rule. The wavy lines under the page names remind you that the pages have not yet been created.

6. Repeat the steps to create more navigation cases on your diagram, as defined in the following table.

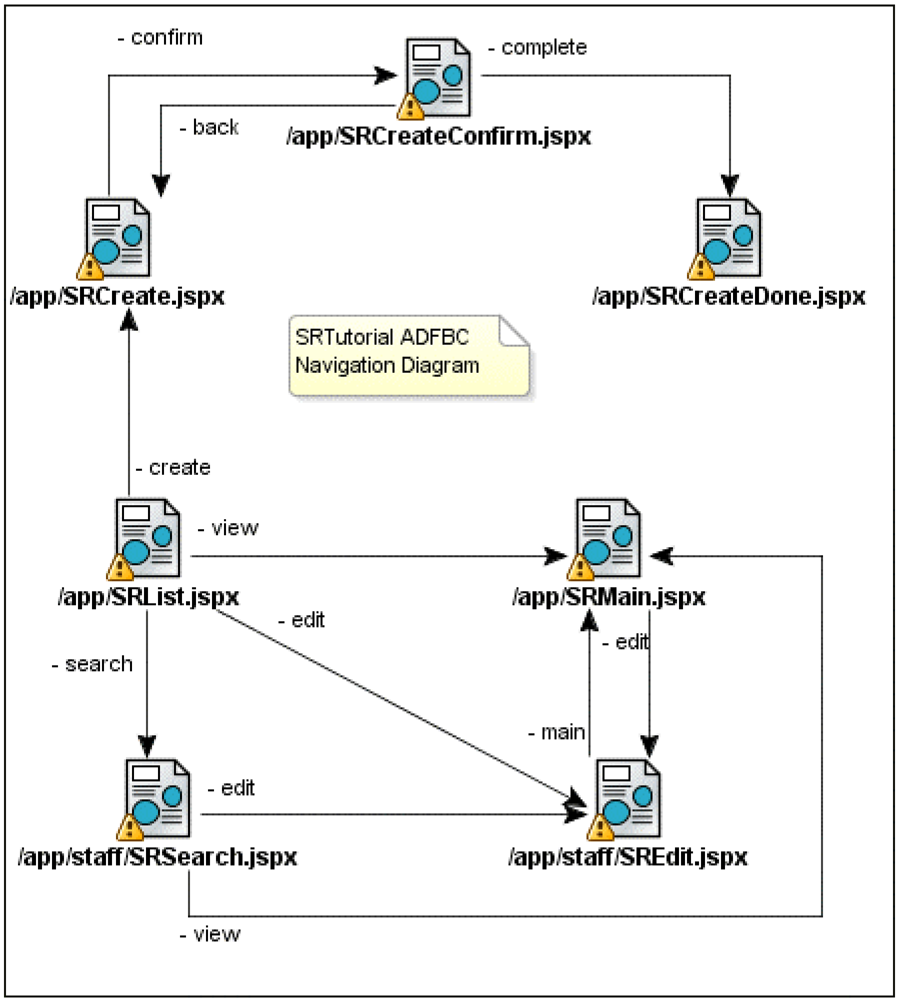
   The table shows the paths by which users navigate between the pages shown. For example, a user clicks a View button on the SRList page to navigate to the SRMain page, and clicks an Edit button to navigate to the SREdit page.

Source	Destination	Outcome
SRList	SREdit	edit
SRList	SRSearch	search
SRMain	SREdit	edit
SREdit	SRMain	main
SRSearch	SRMain	view
SRSearch	SREdit	edit
SRList	SRCreate	create

Source	Destination	Outcome
`SRCreate`	`SRCreateConfirm`	`confirm`
`SRCreateConfirm`	`SRCreate`	`back`
`SRCreateConfirm`	`SRCreateDone`	`complete`

Note: To create a "dogleg" in the line representing the navigation case, click once at the point where you want the change of direction to occur. You can add any number of doglegs to a navigation case. To add a point in the middle of a navigation line you've already created, Shift-click on the line where you want the point. Shift-click on an existing midpoint to remove it.

7. Click **Save** to save the diagram. Your diagram should look like the following screenshot:



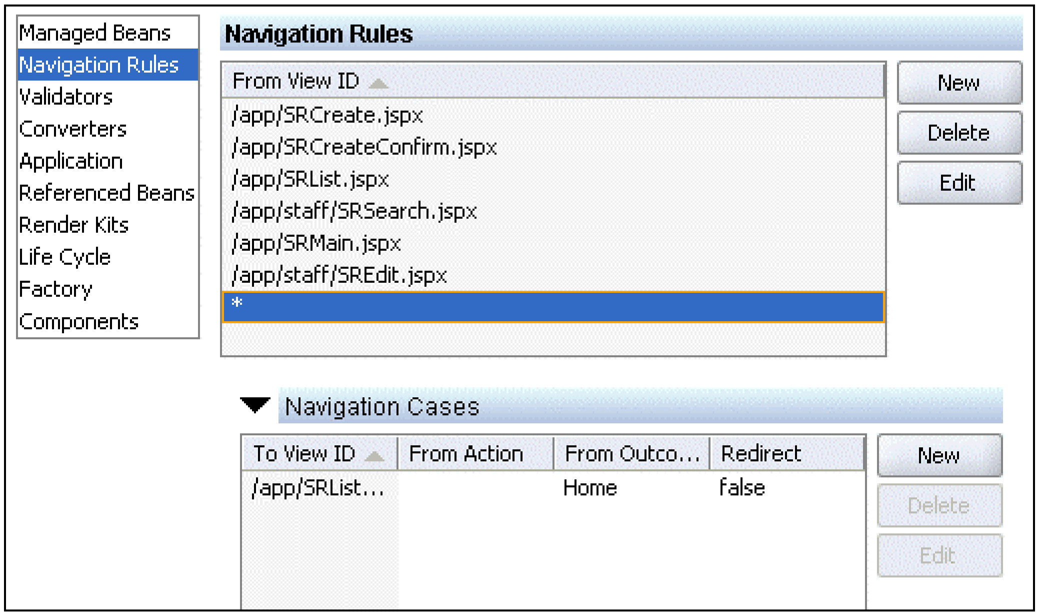
Defining Global Navigation Rules

Global navigation rules define the navigation paths that are available from all pages (typically through the use of Help buttons and Logout icons). You don't use the diagram to define these paths; you use the Overview page (or Configuration Editor) instead.

To define global navigation rules, perform these steps:

1. Click the **Overview** tab at the bottom of the diagram page.
2. Select **Navigation Rules** on the top left of the page. The Navigation Rules box displays the rules you just created in the diagram.
3. Click the **New** button on the right.
4. In the Create Navigation Rule dialog box, choose "*****" from the drop down list, and then click **OK**.
5. Click the **New** button to the right of the Navigation Cases box.
6. Type **`/app/SRList.jspx`** in the To View ID field (the drop-down list is empty because you have not yet created any pages). Type **`Home`** in the From Outcome field. Then click **OK**. This identifies the SRList page as the home page, to which users can return from any page in the application.
7. **Save** your work. In later chapters you create JSF pages and employ this rule.

The completed Navigation Rules page should look like the following screenshot:



From View ID
/app/SRCreate.jspx
/app/SRCreateConfirm.jspx
/app/SRList.jspx
/app/staff/SRSearch.jspx
/app/SRMain.jspx
/app/staff/SREdit.jspx
*

To View ID	From Action	From Outco...	Redirect
/app/SRList...		Home	false

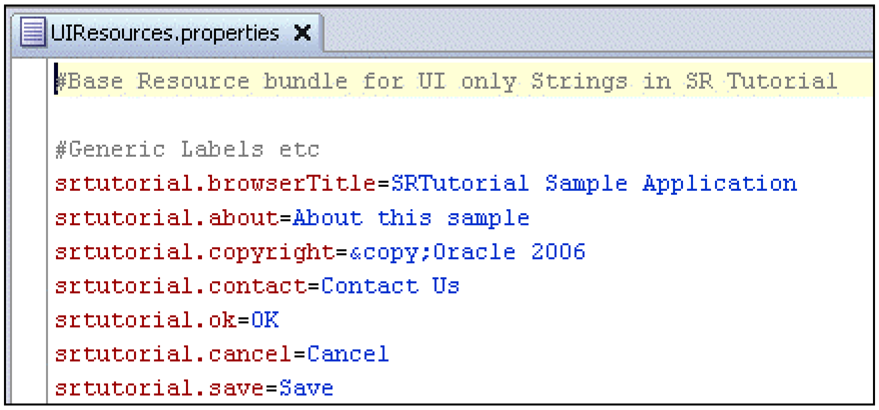
Providing for Translation of the User Interface

JavaServer Faces (JSF) makes the process of translating the user interface into multiple languages relatively straightforward. The strings used in the user interface of tutorial pages, which do not come directly from the presentation hints of our business components, are defined in a single file, `UIResources.properties`. This is a flat text file containing name=value pairs, where the name is an abstract identifier of a resource and the value is the actual value that is displayed at run time. For example, `srtutorial.browserTitle`=SRTutorialSample Application.

This kind of properties file makes maintenance of the UI simpler and also helps greatly with translation. To translate the application, all you have to do is duplicate the properties file by selecting File > Save As and saving with a file name with the appropriate suffix (for example, `UIResources_de` for the German version). You then translate the values of the messages that need to be different in German (leaving the names of the resources the same). In the translated version of the resource file, you can delete any lines that don't need to be different in German. JSF inspects the browser locale information and automatically loads the correct bundle at run time. If a resource is not found in the locale-specific bundle, JSF uses the base bundle instead.

For convenience, the `UIResources` file is supplied with the tutorial. Perform the following steps to import the file into your JDeveloper project:

1. In the Navigator, select the **UserInterface** project, and then choose **New** from the context menu.
2. In the New Gallery, select the **General** node in the Categories pane (if it is not already selected), and then choose **File** in the Items pane. Click **OK.**
3. Name the file **`UIResources.properties`** and click **OK**.
4. JDeveloper creates a Resources node under the `UserInterface` project and places the file there.
5. In Windows Explorer (or the equivalent in your operating system), navigate to the directory where you unzipped the setup files. Open the **`UIResources.properties`** `<tutorial_setup>\files` directory), select all the text in the file, and copy it to the clipboard.
6. In JDeveloper, paste the contents of the clipboard into the **`UIResources.properties`** file and save it.
7. Examine some of the name=value pairs described above. You use them throughout your pages. The following screenshot shows some examples:



```
#Base Resource bundle for UI only Strings in SR Tutorial

#Generic Labels etc
srtutorial.browserTitle=SRTutorial Sample Application
srtutorial.about=About this sample
srtutorial.copyright=&copy;Oracle 2006
srtutorial.contact=Contact Us
srtutorial.ok=OK
srtutorial.cancel=Cancel
srtutorial.save=Save
```

Now the file containing all the reference labels is included in the UserInterface project, you can include it into a set of standard components.

Creating a Standard Look and Feel

When developing Web applications, you can provide a consistent user experience by maintaining a common look and feel (same use of color, same screen design and layout) and—more importantly—by creating similar interaction behaviors. You can create a simple template that can be used as the basis for each of the pages in the application. This can increase your productivity as a developer because you do not need to “reinvent the wheel” for each page you build.

Perform the following steps to create a template that serves as the basis for developing all the pages in the SRTutorialADFBC application:

1. In the Navigator, select the **UserInterface** project, and then choose **New** from the context menu.
2. In the New Gallery, expand the **Web Tier** node in the Categories pane (if it is not already expanded) and choose **JSF**.
3. In the Items pane, choose **JSF JSP** and then click **OK**. The Create JSF JSP Wizard launches.
4. Complete the first three steps of the wizard using the following values:

Wizard Step 1: JSP File

Field	Value
File Name	`SRTutorialTemplate.jspx`
Directory Name	This is the location where the file is stored. Ensure that you create the page in the <jdev_install>`\jdev\mywork\SRTutorialADFBC\UserInterface\public_html\Template` folder.
Type	JSP Document (*.jspx)
Mobile	Clear the check box.

5. Click **Next**.

Wizard Step 2: Component Binding

Field	Value
Do Not Automatically Expose UI Components in a Managed Bean	Ensure that this option is selected. (Because this is just the outline for the pages you are going to develop later, you choose not to create a managed bean for the UI components at this point. Later, when you create the detailed individual pages, you can create managed beans for

	the each pages' UI components.)

6. Click **Next**.

Wizard Step 3: Tag Libraries

Field	Value

Selected Libraries	`ADF Faces Components`
	`ADF Faces HTML`
	`JSF Core`
	`JSF HTML`

7. Click **Finish**

   The empty page appears in the Visual Editor.

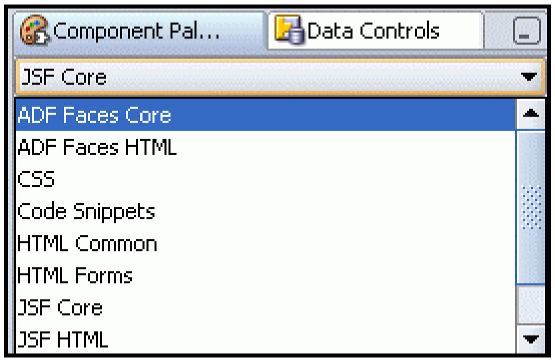
Adding Components to the Page

The next steps show you how to add components to the template page. The Component Palette is used for creating components in the different visual editors.

The Component Palette comprises a number of palette pages. Each palette page contains a logical grouping of components. For example, when you created the JSF navigation diagram, the palette page contained JSF Page, JSF Navigation Case, Note, and Attachment. Those are the items that are appropriate for a JSF navigation diagram.

In some cases there are multiple sets of components available. When that is the case, you select the group of components that you want by clicking the list of palette pages on the component palette.

The following screenshot shows some of the palette pages you use to create your JSF pages:



In the next series of steps, you include ADF components, which then are used as the basic elements of all the application pages. If the Component Palette is not visible, select **View > Component Palette**.

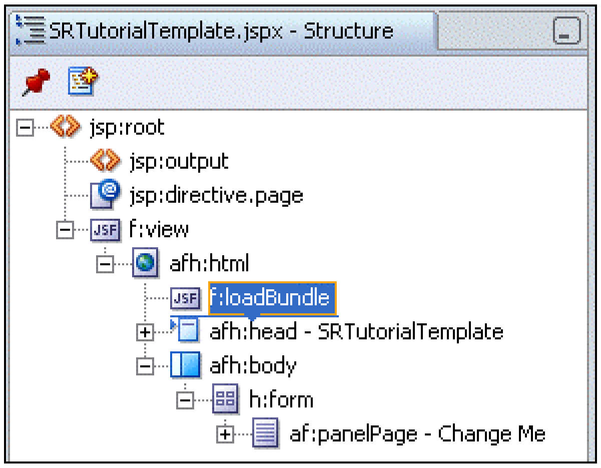
1. Select the **ADF Faces Core** page in the Component Palette, and then locate the **PanelPage** component in the list.

2. Drag the **PanelPage** to the template in the Visual Editor.

3. In the Property Inspector, type **`Change me`** in the Title property. This title will be modified appropriately for the pages you subsequently develop from the template page.
4. Add branding to the page as follows: Locate a directory called **`images`** in the `<ADFBCTutorialSetup>\files` directory. Using Windows Explorer (or the equivalent in your operating system), copy the entire directory to `<jdev_install>\jdev\mywork\SRTutorialADFBC\UserInterface\public_html`.
5. In the Application Navigator, select the **Web Content** folder and choose **Refresh** from the View menu.
6. Locate the **`images`** node, expand it if it is not already expanded, and drag **`SRBranding.gif`** to the "branding" facet at the upper left of the template page. In the pop-up window, choose **`GraphicImage`** as the type of component to create.
7. In the Property Inspector, ensure the resulting URL property is **/images/SRBranding.gif**.

In the next steps, you add a `LoadBundle` tag, which is used to help in the translation process when internationalizing applications. The `loadBundle` tag identifies the resource bundle that is used in the jspx pages of the application (that is, the `UIResources.properties` file you copied in the previous section).

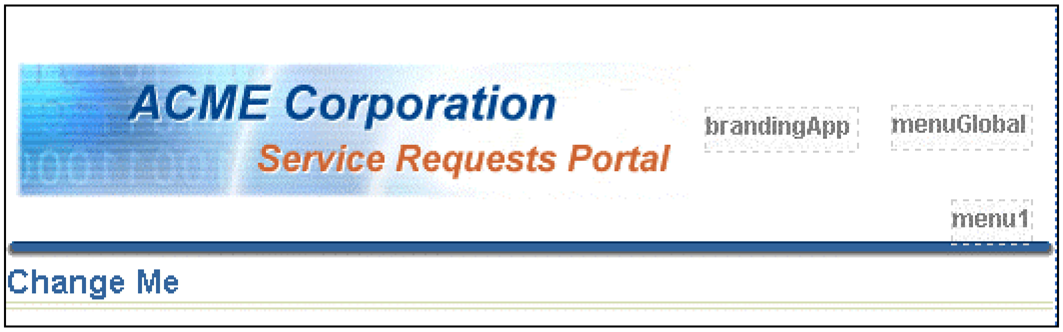
8. Select the **SRTutorialTemplate.jspx** node in the Applications Navigator and open the Structure window (View | Structure)
9. Select the **JSF Core** Component Palette page. Drag the **`LoadBundle`** component to the Structure window, and drop it above **`<>afh:head`**. Make sure it is right underneath the **`afh:html`** tag as shown on the image.



Important Note: The f:loadBundle *must* be first component in the list of children. If it is not first, the Expression Language Editor (EL) will not show the reference to the UIResource.properties file. You must complete the next step to see the LoadBundle component in the Structure Pane.

10. In the Insert LoadBundle pop-up window, set the Basename to **`UIResources`** (or use **[…]**) and click the Properties File radio button to browse for the properties file. Remember that the file is in `<jdev_install>\jdev\mywork\SRTutorial\UserInterface\`. Set the `Var` property to **`res`** (`res` is the page-scoped alias for the resource bundle, which can then be used in Expression Language throughout the pages). Click **OK.**

11. **Save All** your work.
12. Your template page should look like the following screenshot:



Summary

In this chapter, you created a page-flow diagram showing the pages of your application and the links needed for users to navigate between them. To accomplish this, you performed the following key tasks:

- Created outline pages on a page-flow diagram
- Created navigation links between the pages
- Defined global navigation rules
- Created a resources file to facilitate translation of the application's UI
- Created a template page to provide a standard look and feel for application pages

4 Presenting and Packaging the Data Model

This chapter describes how to define the SQL queries that comprise the data model of the application using ADF view objects. A view object is a component that simplifies all aspects of working with rows of data for the user interface.

The chapter contains the following sections:

- Introduction
- Creating View Objects
- Linking the Viewed Data
- Exposing Linked Views in an Application Module
- Querying Information About the Logged-In User
- Summary

Introduction

In Chapter 2, you built the first part of an ADF Business Components application focusing on designing your business objects layer with entity objects and using them to encapsulate some basic application business logic. You also created the SRPublicService application module to contain our data model. Now you must focus on how to present the data in a useful and logical way so the end user can accomplish the required tasks your application must support, using the pages you've defined. In this chapter we'll begin to flesh out the queries that comprise the data model.

There can be any number of ways you may want to look at the data in your application. Some pages require summarized information that might join data from multiple related tables, including only a small number of the total available columns, filtering the data to display only certain rows, and ordering the data in the way that makes the most sense for the task at hand. Other pages might require showing and editing more detailed information, perhaps master and related details on the same page, or other combinations.

A view object uses a SQL query to sort and filter data. It encapsulates the SQL query along with information about which entity objects the queried data relates to (if any). This information enables the framework to automatically enforce your business rules, keep related data in sync when foreign keys change, and to apply any changes made to the view object's rows back to the database with no code. In addition, you can relate two view objects using a view link to achieve automatic master/detail coordination. We'll see that no matter what your application requires, you can retrieve exactly the data you need for the task at hand using one or more view objects.

In this chapter you create three main view objects, two are used in the SRList page of the service request application. In later chapters you create additional view objects to support specific functionality used in other pages.

You perform the following key tasks:

- Create view objects
- Define a view object's entities and attributes
- Modify the SQL statement
- Create bind variables

Note: If you did not successfully complete Chapter 3, you can use the end-of-chapter application that is part of the tutorial setup:

1. Create a subdirectory named **`Chapter4`** to hold the starter application. If you used the default settings, it should be in `<jdev-install>\jdev\mywork\Chapter4`.
2. Unzip **`<tutorial-setup>\starterApplications\SRTutorialADFBC-EndOfChapter3.zip`** into this new directory. Using a new, separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRTutorialADFBC application workspace.
4. Select **File > Open**, and then select **`<jdev-install>\jdev\mywork\Chapter4\SRTutorialADFBC\SRTutorialADFBC.jws`**. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 3.

Creating View Objects

In this chapter you create three major data components used by pages in the service request application. Each view object supports a different bit of functionality, retrieving information about the user who is logged on, all service requests, and all service histories. In this section you create the view objects to represent these three data areas.

Defining the Logged In User View

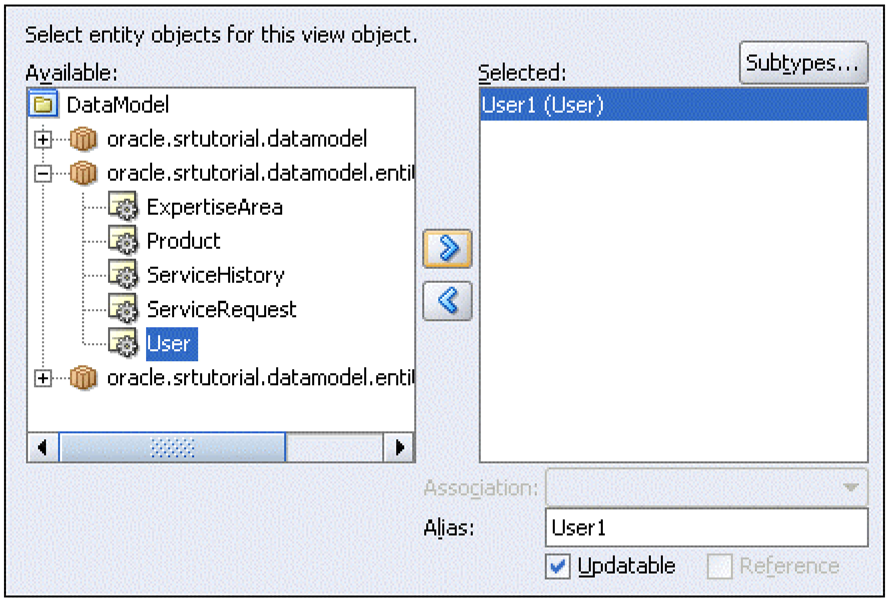
Create the view object representing the user who is logged in. The view uses the Id, Email, First and Last Names from the User entity object. A bind variable passes the email address into the SQL Where clause to retrieve the logged in user information.

1. In the Applications Navigator, right-click the **`oracle.srtutorial.datamodel`** node, and select **New View Object** from the shortcut menu.
2. Click **Next** on the Welcome page.
3. In the first step of the Create View Object Wizard, set the properties using the values from the following table. When done, click **Next**. If the property is not listed below, then leave the default value.

Property	**Value**
`Package`	`oracle.srtutorial.datamodel.queries`
`Name`	`LoggedInUser`

Extends	`<null>`

4. In Step 2: Entity Objects, expand the **`oracle.srtutorial.datamodel.entities`** node and shuttle the **User** entity to the selected side.



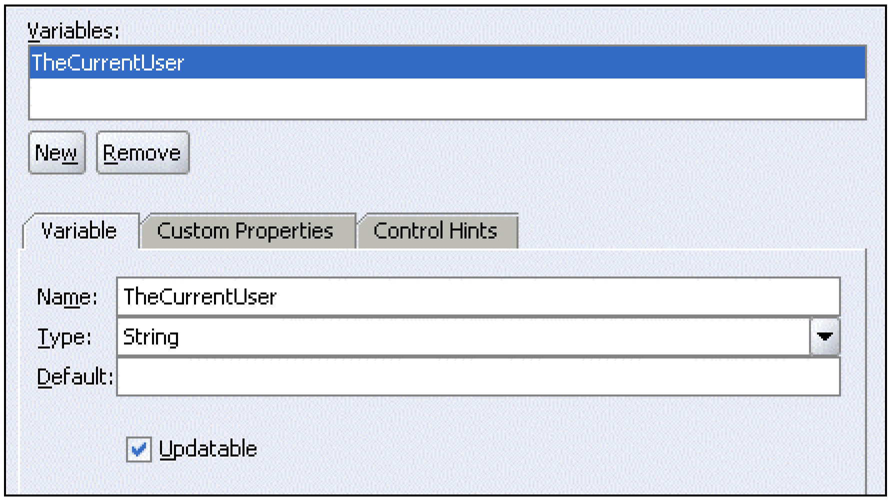
Alias: User1

5. Change the Alias property value to **Users** and click **Next**.
6. In Step 3: Attributes, shuttle the following four attributes to the selected side.
   - `UserId`
   - `Email`
   - `FirstName`
   - `LastName`

   Using these four attributes you can find and display all the important information about a user.
7. Click **Next** to continue.
8. In Step 4: Attribute Settings, leave the values at their defaults, and click **Next**.
9. In Step 5: SQL Statement, notice that all four attributes are included in the select statement.
10. Add a Where clause to be used to receive a value from a variable. Set the Where clause as follows.

    ```
    Email=lower(:TheCurrentUser)
    ```
    This will allow us to retrieve the information for a particular logged-in user based on the value of this bind variable that we'll formally define in the next step.
11. Click **Next**.

12. In Step 6: click the **New** button and create a bind variable to pass in the e-mail address. Name the bind variable **TheCurrentUser** and set the type to **String**.



Name: TheCurrentUser
Type: String
Default:
✓ Updatable

13. Click **Finish** to complete the creation of the view.

This view object should be used anytime a page returns service requests for a specific logged on user. Security is then responsible for passing the value from the logon screen to the LoggedInUser view object.

Defining the Service Request View

In the next steps, create the view object representing service request data. This view object can be used to retrieve service requests depending on the request's status. The Status of a Service Request may be: Open, Closed, or Pending. Any page can use this view object to return service requests depending on the status. A bind variable passes the status from a page to the view object which is used in a Where clause.

1. In the Applications Navigator, right-click the **`oracle.srtutorial.datamodel > queries`** node, and select **New View Object** from the short cut menu.
2. Click **Next** on the Welcome page.
3. In the first step of the Create View Object Wizard, set the properties using the values from the table below. If the property is not listed below, then leave it at its default value.

Property	**Value**
`Package`	`oracle.srtutorial.datamodel.queries`
`Name`	`ServiceRequests`

4. Click **Next**.
5. In Step 2: Entity Objects, expand the **`oracle.srtutorial.datamodel.entities`** node and shuttle the **ServiceRequest** entity to the selected side.
6. Click **Next.**
7. In Step 3: Attributes, shuttle all the **ServiceRequest** attributes to the **Selected** side.

8. Click **Next** twice, to continue.
9. In Step 5: SQL Statement, add a where condition to retrieve all service requests with a status like the value of a bind variable.

   ```
   STATUS LIKE NVL(:TheStatus,'%')
   ```
10. Click **Next**.
11. In Step 6: Click the **New** button and create a bind variable to pass in the Status. Name the bind variable `TheStatus` and set the type to `String`.
12. Click **Finish** to create the view.

In a later chapter the ServiceRequest view object is used by the SRList page. The page displays service requests, based on the value of Status. An Enduser can select the status value from a menu, which passes the value to the view object's bind variable. The bind variable is used in the Where clause to restrict the records returned.

Defining the Service Histories View

In the next steps, create a view object representing all the service history data.

1. In the Applications Navigator, right-click the `oracle.srtutorial.datamodel > queries` node, and select **New View Object** from the short cut menu.
2. Click **Next** on the Welcome page.
3. In the first step of the Create View Object Wizard, set the properties using the values from the table below. If the property is not listed below, then leave it at its default value.

Property	Value
Package	oracle.srtutorial.datamodel.queries
Name	ServiceHistories

4. Click **Next**.
5. In Step 2: Entity Objects, expand the `oracle.srtutorial.datamodel.entities` node and shuttle the **ServiceHistory** entity to the selected side.
6. Click **Next** to continue.
7. In Step 3: Attributes, shuttle all the **ServiceHistory** attributes to the **Selected** side.
8. Click **Next** to continue.

   The reminder of the view object properties can be left at their default. Some like the label and format will get inherited at runtime from their underlying source entity objects.
9. In Step 4: Attribute Settings, do not change any values and click **Finish to** create the **ServiceHistories** view object.

Three of the view objects needed for the application are complete. These view objects can now be used in the JSF application pages. As pages are developed, additional view objects may be needed to support specific business functionality. In later chapters you create additional view objects to support additional application pages.

Linking the Viewed Data

The view objects receive data from a single entity, queried from a single table. In many cases, your pages require displaying a coordinated view of master and detail information. The relationship between the source and destination is the view link. The source end is also referred to as the master end; the destination is referred to as the detail end.

In the SRTutorialADFBC application, two main view links are needed to connect the view objects you have already created. You need to create two view links: one to show what user created a service request, the other to show service histories for a service request.

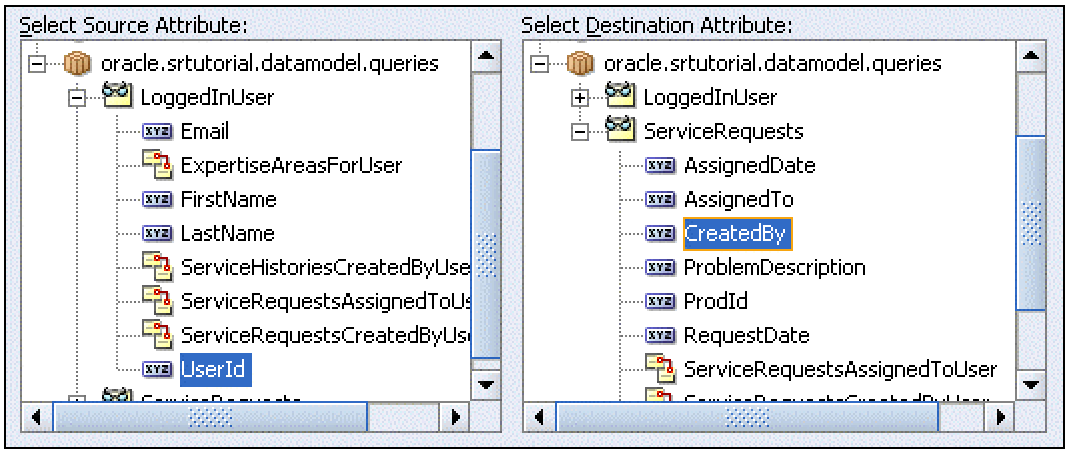
Linking the User with Service Requests

Since we want to display information coming from both the ServiceRequest and the User business objects, we'll involve both of these view objects to create a view link for associating the view objects.

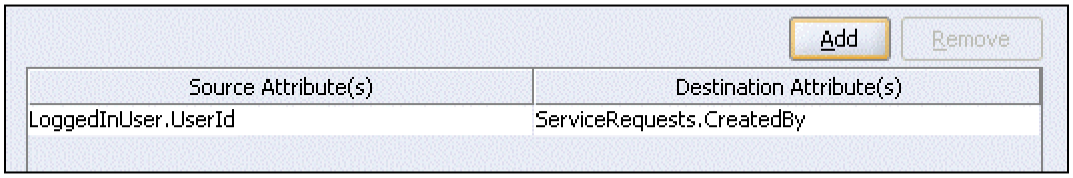
1. In the Applications Navigator, expand the **`oracle.srtutorial.datamodel.queries`** nodes.
2. Select the queries node, and from the context menu select **New View Link**.
3. Click **Next** in the Welcome page, to continue.
4. In the first step of the Create View Link Wizard, set the properties using the values from the table below. If the property is not listed below, then leave the default value.

Property	Value
`Package`	`oracle.srtutorial.datamodel.queries.viewlinks`
`Name`	`RequestCreatedByUser`

5. Click **Next**.
6. For Step 2: View Objects, in the Source Attribute pane expand **`oracle.srtutorial.datamodel.queries`** and then the **`LoggedInUser`** entity.
7. Scroll down and highlight the **`UserId`** attribute.
8. In the Destination Attribute pane, expand **`oracle.srtutorial.datamodel.queries`** and then the **`ServiceRequest`** entity.
9. Select the **`CreatedBy`** attribute.


Select Source Attribute:
oracle.srtutorial.datamodel.queries
LoggedInUser
Email
ExpertiseAreasForUser
FirstName
LastName
ServiceHistoriesCreatedByUse
ServiceRequestsAssignedToUs
ServiceRequestsCreatedByUs
UserId
Select Destination Attribute:
oracle.srtutorial.datamodel.queries
LoggedInUser
ServiceRequests
AssignedDate
AssignedTo
CreatedBy
ProblemDescription
ProdId
RequestDate
ServiceRequestsAssignedToUser

10. With both attributes selected, click the **Add** button. The link will appear in the lower pane of the step.



Source Attribute(s)	Destination Attribute(s)
LoggedInUser.UserId	ServiceRequests.CreatedBy

11. Click **Next** to continue.

The link is created using the CreatedBy attribute but our application requires that this detail query show the service requests that were *either* created by *or* assigned to the currently logged-in user. To incorporate this functionality, alter the default Where condition.

12. Replace the default Where clause with the one below, to also return records that are assigned to a user.

```
(:Bind_UserId = ServiceRequest.CREATED_BY OR :Bind_UserId =
ServiceRequest.ASSIGNED_TO)
```

13. Click the **Test** button to confirm that the query is valid.

14. Click **Finish** to complete creating the view link.

The view link connects the LoggedInUser and ServiceRequests view objects, connecting them by user id. This allows a user to login to the application, select a status value and then see all their service requests for that value.

Exposing Linked Views in an Application Module

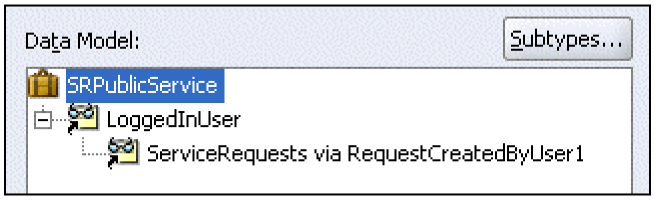
Any time you add new view objects or links to the project, you can include them in a new or existing application module. Now that you have three view objects and one view link, you include them in the SRPublicService application module. Once the view objects are added to the application module, they are available for use in JSF pages.

Note: In this tutorial you will not need to, but in general you can reuse view objects and view links and use them as part of the data model for any number of application modules, as required. In a similar vein, you can define any number of view objects that reference the same underlying entity objects that you have built, presenting their data filtered, joined, and sorted in any way to meet your requirements.

Include the User and their Service Requests

In the next steps, include the two view objects representing a logged on user and their service requests in the SRPublicService application module. The application module then can be used in a page to display data. Only those view objects included in an application module can be used in JSF pages.

1. In the Applications Navigator, open the **`oracle.srtutorial.datamodel`** node and double-click the **SRPublicService** application module
2. In the Application Module editor, select the **Data Model** node.
3. In the Available pane, expand the **`oracle.srtutorial.datamodel.queries`** node.
4. Select the **`LoggedInUser`** node, change the name from **LoggedInUser1** to **LoggedInUser** and shuttle it to the Data Model pane.
5. In the Data Model pane, select the **LoggedInUser**. Then in the Available pane, select the **`ServiceRequests via RequestCreatedByUser`** view object that is nested below the LoggedInUser view object (its name appears as ServiceRequests1**)** and shuttle it to the Data Model pane.
6. Change the Instance Name from **ServiceRequests1** to **ServiceRequests**. The Data Model pane should look like the image below.



7. Click **OK** to complete the application module.
8. Use **Save All** to save your work.

The views needed to track who is logged in and to retrieve service requests are now able to be used while creating pages.

Querying Information About the Logged-In User

Now that the view objects are added to the application module, you want to be able to see records returned to the page. For testing purposes only, override the `prepareSession` method and hard code a user ID to populate the `TheCurrentUser` bind variable. This variable is used to determine what service requests to retrieve.

Note: In a later step of the tutorial, after enabling Java EE security, you will remove this hard coded user ID value and replace it with a call to a function that returns the name of the currently logged-in user.

---

1. In the Applications Navigator, click the **SRPublicService** application module.
2. In the Structure pane, expand the **Sources** node, and double-click the **`SRPublicServiceImpl.java`**. This will invoke the code in the editor.
3. Select **Source > Override Methods** from the menu bar.
4. Scroll down the Override Methods dialog box, and click the check box next to the **prepareSession(oracle.jbo.Session)** method.
5. Click **OK**.
6. In the code editor, replace the existing `prepareSession ()` method `(oracle.jbo.Session)` with the code below.

```
protected void prepareSession(Session session) {
  super.prepareSession(session);
  String currentUser = "sking";
// Replace later with getUserPrincipalName()

getLoggedInUser().setNamedWhereClauseParam("TheCurrentUser",currentUser);
getLoggedInUser().executeQuery();
// Save the current user id in the session user data so entities can
// refer to it
getDBTransaction().getSession().getUserData().put("CurrentUserId",
getLoggedInUser().first().getAttribute("UserId"));
}
```

If needed, press [Alt] + [Enter] to include the (`oracle.jbo.Session`) import statement.

We see at the top of our `SRPublicServiceImpl.java` file that our class extends a base ADF framework class for application modules:

```
public class SRPublicServiceImpl extends ApplicationModuleImpl
```

The `prepareSession()` method, inherited from the base class for ADF application modules, is invoked, or "triggered," by the framework when a user begins using the application module component for the first time in a session. By writing a method in your custom application module class with the same name and signature as a method in the class from which it extends, you can override the default behavior to augment it with some additional logic before or after performing. The line `super.prepareSession(session)` is the Java code that says "go perform the functionality the superclass would have performed had I not overridden this method." In this way, the code you added after this line augments the default behavior.

Using the automatically-generated `getLoggedInUser()` method, we access the `LoggedInUser` view object instance and set the value of the named Where clause parameter, `TheCurrentUser`, to the value of the `currentUser` variable and then execute the query to retrieve the information from the underlying `USERS` table for that user. The next line accesses the "user data" hashtable from the current session, and puts an entry named `CurrentUserId` containing the value of the UserId attribute of the first row of the `LoggedInUser` view object. This will be the numerical user ID of the currently logged-in user. You will make use of this numerical ID in a later step of the tutorial.

Next to the overridden `prepareSession()` method that we added, the JDeveloper 10*g* code editor gives us a positive confirmation that the method is overriding a base class method by showing the symbol. If you hover your mouse over this symbol, you will see the tooltip that says “Overrides method in oracle.jbo.server.ApplicationModuleImpl”. It’s important to see this symbol next to your overridden methods because if you have typed the method name incorrectly, or have provided a set of parameters that is different from the ones in the superclass, then your method will not be overriding the base class method correctly and will not get automatically triggered by the framework

You now have the view objects needed for the SRList page and made them available while developing client applications. In the next chapter you create the SRList page and use the ADF Business Components you just created.

Summary

In this chapter, you created three view objects representing data used in JSF pages. You created two view links, allowing more complex data to be displayed. Finally, you added the view objects and links to the application module, making them available as data controls in the pages. To accomplish this, you performed the following key tasks:

- Created view objects
- Customized attribute definitions
- Linked view objects
- Exposed view objects and links in an application module
- Queried information about the logged-in user

5

Developing a Simple Display Page

This chapter describes how to create the SRList page, a simple display page at the center of the SRTutorial application that enables users to view information about service requests.

The chapter contains the following sections:

Introduction

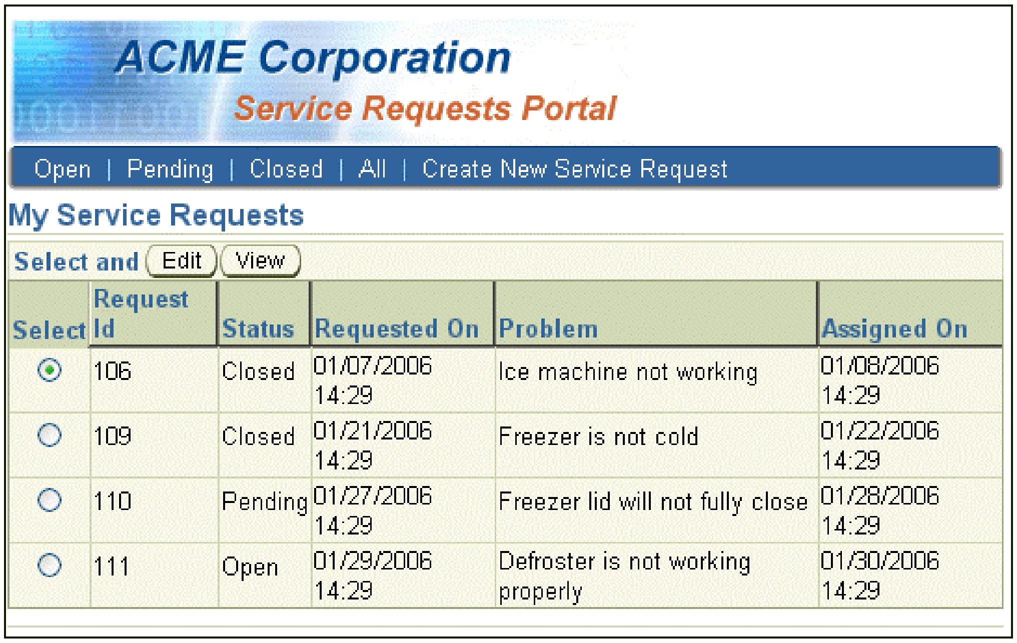
The SRList page is at the center of the SRTutorialADFBC application. It is the first page that users see after logging in, and they can navigate from this page to all the other pages in the application.

Note: If you did not successfully complete Chapter 4, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named **`Chapter5`** to hold the starter application. If you used the default settings, it should be in `<jdev-install>\jdev\mywork\Chapter5`.
2. Unzip **`<tutorial-setup>\starterApplications\SRTutorialADFBC-EndOfChapter4.zip`** into this new directory. Using a new, separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRTutorialADFBC application workspace.
4. Select **File > Open**, and then select **`<jdev-install>\jdev\mywork\Chapter5\SRTutorialADFBC\SRTutorialADFBC.jws`**. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 4.

Right-click the **UserInterface** node in the Navigator and select the **Open JSF Navigation** option from the context menu (alternatively, you can double-click the **`faces-config.xml`** file in the Navigator) to revisit the page-flow diagram that you created and see how the SRList page relates to the other pages. The screen shot shows the completed SRList page.



Select	Request Id	Status	Requested On	Problem	Assigned On
●	106	Closed	01/07/2006 14:29	Ice machine not working	01/08/2006 14:29
○	109	Closed	01/21/2006 14:29	Freezer is not cold	01/22/2006 14:29
○	110	Pending	01/27/2006 14:29	Freezer lid will not fully close	01/28/2006 14:29
○	111	Open	01/29/2006 14:29	Defroster is not working properly	01/30/2006 14:29

Here are some points to note about the SRList page:

- The page displays each existing service request created by or assigned to the currently logged-on user, including the service request's status.
- Any user (customer, technician, or manager) can access the page.
- When a customer logs in, all service requests created by that customer are displayed. A View button is available.
- When a technician logs in, all service requests assigned to that technician are displayed. View and Edit buttons are available.
- The list of displayed requests can be filtered by clicking the second-level menu to show one of the following: all service requests currently open, only requests with a status of closed, or all requests irrespective of status. The default is requests that are open.
- Any user can select a service request and, by clicking the View button, go to the SRMain page to update the history of that request. In addition, technicians can click the Edit button to visit the SREdit page, where they can edit the request.
- To add a new request, any user can navigate to the SRCreate page by choosing the Create New Service Request menu option.
- Technicians can navigate to the Search page (SRSearch) by using the Advanced Search menu tab.
- Managers can see all service requests. All menu options are available to managers.

To create the SRList page with the functionality and look-and-feel described in the preceding list and screenshot, you now perform the following key tasks:

- Creating the page outline, based on the template page you created in Chapter 3
- Adding user interface components to the page
- Connecting up the View and Edit buttons
- Defining Refresh behavior
- Creating a menu bar to enable users to view service requests with different statuses or to create a new service request
- Creating drilldown functionality to enable users to select a row in the table and navigate to the SRMain page to add information for the selected service request

Creating the Page Outline

In this section, you create the SRList page and add the template to apply the appropriate look and feel.

1. If it is not already open, double-click the **`faces-config.xml`** file to view the Page Flow Diagram. (If `faces-config.xml` is open but not showing the diagram, click the **Diagram** tab at the bottom of the visual editor window).
2. Double-click the **SRList** page to invoke the JSF Page Wizard.

3. Complete the first three steps of the wizard using the following values:

Wizard Step 1: JSP File

Field	**Value**
File Name	`SRList.jspx`
Directory Name	This is the location where the file is stored. Ensure that you create the page in the `\SRTutorialADFBC\UserInterface\public_html\app` folder.
Type	`JSP Document`
Mobile	Ensure that the **Add Mobile Support** check box is clear.

4. Click **Next**.

Wizard Step 2: Component Binding

Field	**Value**
Do Not Automatically Expose UI Components in a New Managed Bean	Ensure that the option is selected.
Name	`disabled`
Class	`disabled`
Package	`disabled`

5. Click **Finish** to create the page details. The new SRList page is displayed in the Visual Editor.
6. Expand **`UserInterface > Web Content`** nodes to open the **`SRTutorialTemplate`** file if it is not already open.
7. In the Structure window, collapse the **`afh:html`** node and select it. From the context menu, choose **Copy**.
8. Click the tab to return to the SRList page, and in the Structure window expand the **`f:view`** node.
9. Delete the **html** node, and then right-click **`f:view`** and choose **Paste** from the context menu. The look and feel that you created earlier is now applied to the new page.

Adding User Interface Components to the Page

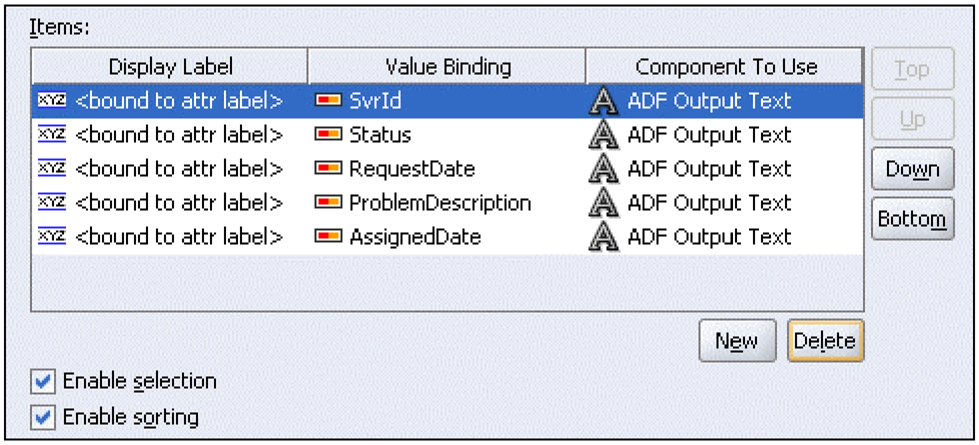
Perform the following steps to add some user-interface elements to the page:

1. Add a title to your page as follows: Click the page in the Visual Editor to select it. (Alternatively, you can select **`af:panelPage`** in the Structure window.) In the Structure window, choose **Properties** from the context menu.

2. In the PanelPage Properties dialog box, click **Bind** in the Title property.
3. In the Bind to Data dialog box, expand the **JSP Objects** node and then the **res** node in the Variables tree.
4. Scroll through to locate **`srlist.pageTitle`**. Select it and shuttle it into the Expression pane. Click **OK**, and then click **OK** again. The EL expression `#{res['srlist.pageTitle']}` refers to the resource string from the resource bundle named `res`—loaded using the `loadBundle` tag from our page template—corresponding to the string key of `srlist.pageTitle`.
5. This is the name of the page title resource as defined in the `UIResources.properties` file that you created in the preceding chapter. The value displayed at run time is the actual title of the page.
6. Add a header to display in the browser title when you run the page, as follows:
   In the Structure pane, select **`afh:head`**. In the Property Inspector, set the Title property to **`#{res['srtutorial.browserTitle']}`**.

Note: You can set these values as follows. Invoke the Properties pane, select the Title value field, and then click the dotted button that appears at the far right of the value field. The "Bind to Data" dialog box opens. Pick from the Variables tree as you did in the preceding step. (Alternatively, you can type the value in the Property Inspector.)

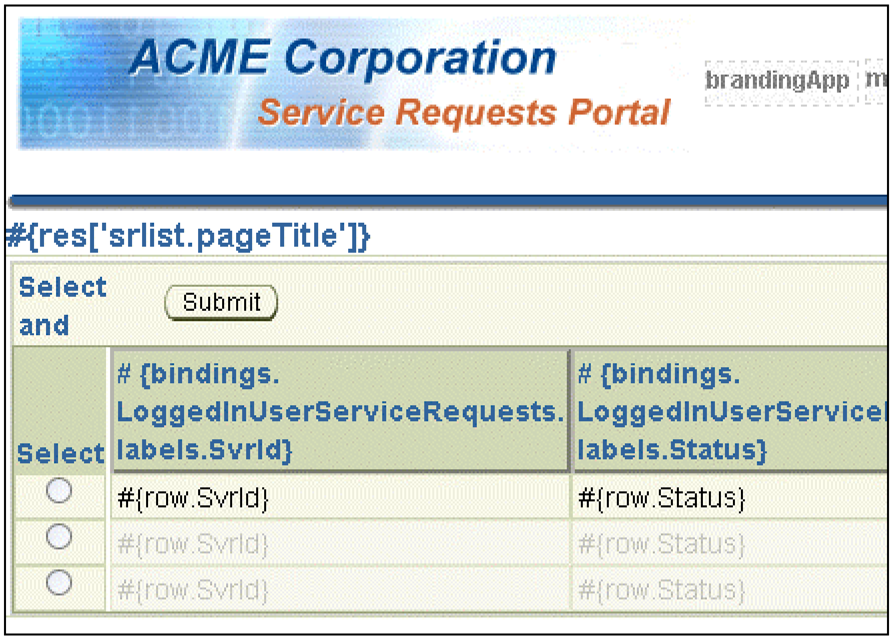
7. You now add data to the page. The data is in the form of a read-only table bound to the **ServiceRequests** view object. First, ensure that the Data Control Palette is visible to the right of the Visual Editor. If it is not, choose **Data Control Palette** from the View menu.
8. Expand the **`SRPublicServiceDataControl`** node, and then select **`LoggedInUser`** → **`ServiceRequests`** from the list of data collections.
   The fact that the `ServiceRequests` data collection is nested inside the `LoggedInUser` collection in the data control palette reflects the fact that it is a detail collection of only those service requests for the current row in the `LoggedInUser` collection.
9. Drag **ServiceRequests** to the structure window and drop it on **`af:panelPage`**. From the Create context menu, choose **`Tables > ADF Read-only Table`**.
10. In the Edit Table Columns dialog box, reorder the columns so that **SvrId** is at the top of the list, followed by **Status, RequestDate, ProblemDescription**, and **AssignedDate**. This determines the order of the columns on the page. Use the **Delete** button to remove all other fields. Make sure that the "Component to Use" column has **ADF Output Text** set for every column. The ADF Output Text is a JSF control to display a read-only field.
11. Select the **Enable Selection** and **Enable Sorting** check boxes to add an option selection column to the table, and then click **OK**.


Items:

Display Label	Value Binding	Component To Use
<bound to attr label>	SvrId	ADF Output Text
<bound to attr label>	Status	ADF Output Text
<bound to attr label>	RequestDate	ADF Output Text
<bound to attr label>	ProblemDescription	ADF Output Text
<bound to attr label>	AssignedDate	ADF Output Text

☑ Enable selection
☑ Enable sorting

At this point, your page should look like the following screenshot:



#{res['srlist.pageTitle']}

Select and

Select	#{bindings.LoggedInUserServiceRequests.labels.SvrId}	#{bindings.LoggedInUserService labels.Status}
○	#{row.SvrId}	#{row.Status}
○	#{row.SvrId}	#{row.Status}
○	#{row.SvrId}	#{row.Status}

Creating the Edit and View Buttons

The SRList page provides a starting point for customers and technicians. From this page, users can navigate either to an edit page or to a view page. The Edit button is available only when a technician or manager logs in. Clicking this button takes users to the SREdit page, where details of the currently selected service request can be modified.

The View button is available to all users of the application, enabling them to navigate to the SRMain page to update the history of a selected request.

Perform the following steps to change the default submit button to the Edit button.

1. Click the **Submit** button. In the Property Inspector, change the Text property to `#{res['srlist.buttonbar.edit']}`.

This is the name of the button resource as defined in the `UIResources.properties` file. The value displayed at run time is "Edit."

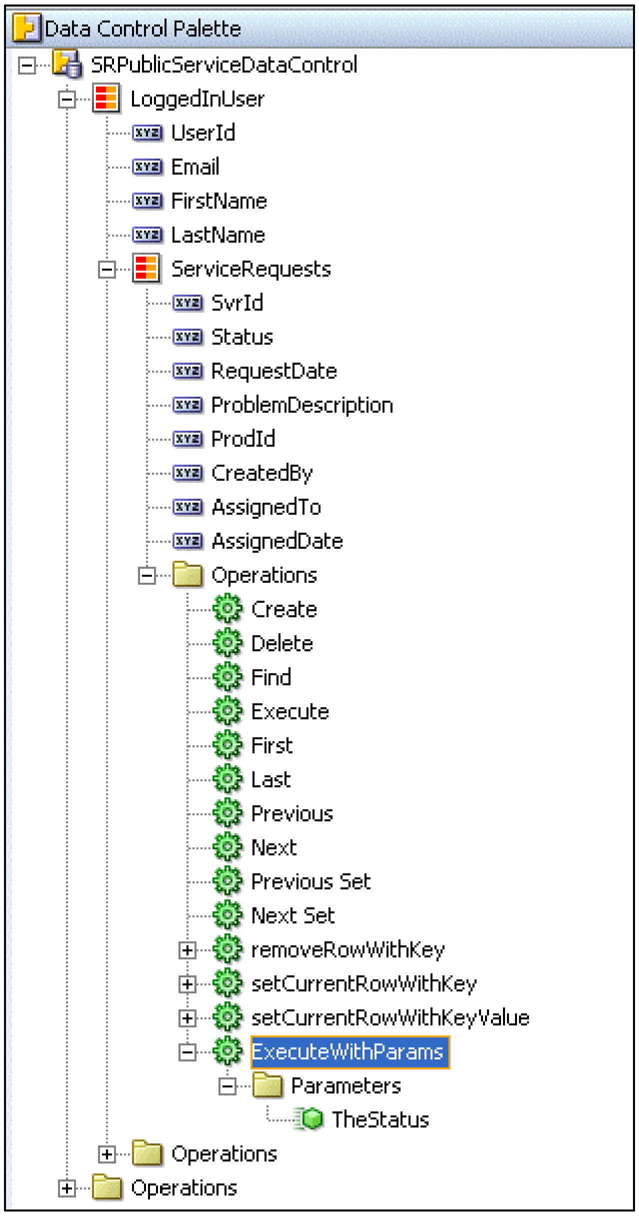
2. Also in the Property Inspector, set the Action property to **edit**. This action outcome string you are able to pick from a drop-down list comes from the fact that you already defined the navigation rules in the JSF diagram. Then, clicking this button follows the navigation rule corresponding to the "edit" outcome; that means (based on the diagram you created) it will navigate you to the SREdit page.
3. Select the **ADF Faces Core** page in the Component Palette and choose the **Command Button** component.
4. Drag and drop the **Command Button** next to the Edit button you just created.
5. In the Property Inspector, set the Text property to **`#{res['srlist.buttonbar.view']}`** and the Action property to **view**.
6. Use **Save All** to save your work.

Adding a Menu Bar to the Page

The SRList page has a set of menu options so that users can choose to view service requests with a status of Open, Closed, or Pending or they can view All requests regardless of status.

Perform the following steps to add these menu options to the page:

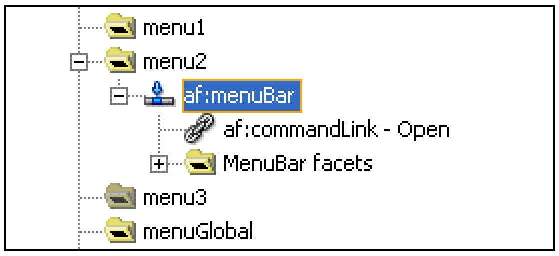
1. You first create a second-level menu bar to hold the menu options. In the Structure window, expand the **PanelPage facets** node, and scroll down to **menu2**. Right-click, and from the menu choose **`Insert inside menu2 > MenuBar`**.
2. In the **Data Control Palette** expand the **`SRPublicServiceDataControl > LoggedInUser > ServiceRequests > Operations`** node and select **`ExecuteWithParams`**. If you expand the Parameters node, you should see the **`TheStatus`** parameter that you created as a bind variable in Chapter 4. Make sure you do not use the nested Operations node under LoggedInUser. If you do, then later you will not be able to bind to TheStatus parameter. The only parameter you will have access to is TheCurrentUser.



3. Drag **`ExecuteWithParams`** to the Structure window onto **`af:menuBar`**. In the Create popup menu, choose **`Operations > ADF Command Link`**.
4. In the Property Inspector, type **`Open`** in the Text property.

At this point you create only one **`commandLink`** for the Open status. Later you will use Copy and Paste to create three additional command links for Pending, Closed, and All options.

The Structure pane should look like the image below.



5. Add a `commandMenuItem` to the menu bar to enable users to create a new service request, as follows: Select the **ADF Faces Core** page in the Component Palette, and then drag a **`CommandMenuItem`** to the menu bar.

6. In the Property Inspector, set the Text property to **`#{res['srlist.menubar.newLink']}`** and set the Action property to **`create`**.

7. **Save** the page.

You now have a command link menu item component displayed on the page. Next add an action listener to the command link menu item, to set the value used by the bind variable. In turn, the bind variable then uses the passed value to return records to the page.

Setting the Menu Parameters

There are four menu items that when selected, must pass a value to the bind variable named TheStatus in the ServiceRequests view object. An Action Listener can be used to set the value passed to the bind variable in the view object. Since the menu item was created using the ServiceRequest view object's ExecuteWithParam built-in action, the bind variable will have its value set to any EL expression that you indicate should be caused to fire by clicking the link.

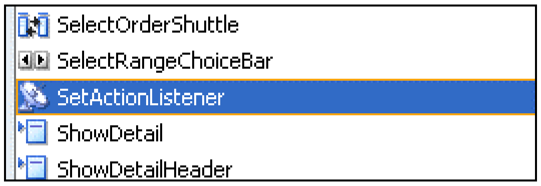
You will use action listeners to set the value of a request-scoped attribute named `Status` to values like `Open`, `Closed`, `Pending`, and `null` depending on which menu link the user clicks. Then you will configure the ExecuteWithParams action to refer to this `Status` attribute using the EL expression of `#{requestScope.Status}` for the value of its bind variable named TheStatus.

Perform the following steps to define a SetActionListener:

1. In the Structure pane, select the **Open commandLink**.

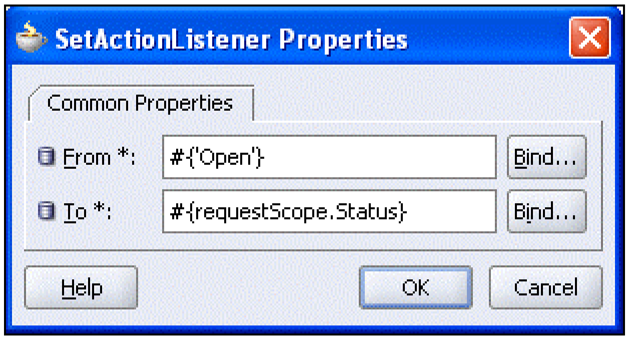
2. Right-click and, from the context menu, select **`Insert inside af:commandLink Open > ADF Faces Core`**.

3. In the dialog box, scroll down and select **`SetActionListener`**.



4. Click **OK**.

5. Set the **From** property to **`#{'Open'}`** and the **To** property to **`#{requestScope.Status}`**. When the CommandLink associated with this action listener is clicked the listener assigns the literal string value "Open" to the requestScope.Status variable.



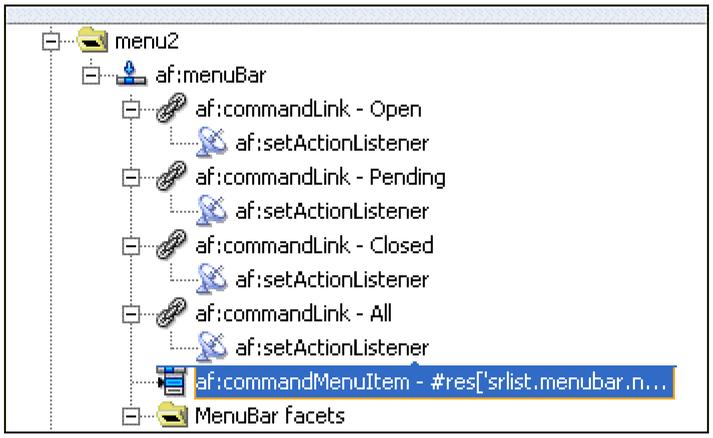
From *: #{'Open'}
To *: #{requestScope.Status}

6. Click **OK**.

Next, create three more command links and Action Listeners as follows. To make quick work of the task, you can leverage the copy-and-paste support in the Structure pane.

7. In the Structure pane, right-click the **Open commandLink** and select **Copy** from the context menu.
8. Right-click the `menuBar` node and select **Paste** from the context menu.
9. Repeat the above step two more times, creating a total of four menu options. Set the Text property for each of the new command links as follows:
   - `Pending`
   - `Closed`
   - `All`

   The Structure pane should now look like this:

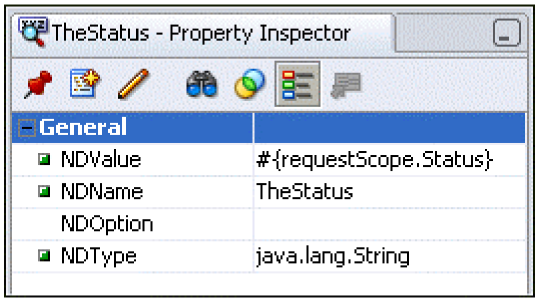


```
menu2
  af:menuBar
    af:commandLink - Open
      af:setActionListener
    af:commandLink - Pending
      af:setActionListener
    af:commandLink - Closed
      af:setActionListener
    af:commandLink - All
      af:setActionListener
    af:commandMenuItem - #res['srlist.menubar.n...
    MenuBar facets
```

10. In the `setActionListener` of the Pending commandLink, set the **From** property to `#{'Pending'}` and the **To** property to `#{requestScope.Status}`
11. In the `setActionListener` of the Closed commandLink, set the **From** property to `#{'Closed'}` and the **To** property to `#{requestScope.Status}`
12. In the `setActionListener` of the All commandLink, set the **From** property to `#{null}` and the **To** property to `#{requestScope.Status}`

Now that the listener is set to assign an appropriate value to the requestScope.Status variable, you will configure the ExecuteWithParams action to use the requestScope.Status variable as the value of its **TheStatus** bind variable. This is done in the declarative data-binding information that Oracle ADF stores in the companion Page Definition XML file for this JSP page.

13. Right-click anywhere on the SRList page and select **Go To Page Definition**.
14. Most of the work in the page definition is completed using the Structure pane. Expand the **`bindings > ExecuteWithParams`** nodes, exposing **TheStatus** parameter.
    Note: If you see an empty structure window, click into the `app_SRListPageDef.xml` file in the code editor and the structure window for the page definition file will appear.
15. In the Property Inspector, override the binding for the NDValue to **`#{requestScope.Status}`**. This defines the value that will be passed to the bind variable named TheStatus when the ExecuteWithParams action is executed. By making it the same EL expression as we used in the To expression of the SetActionListener, we ensure that the value used for the bind variable will be the value set by the action listener.



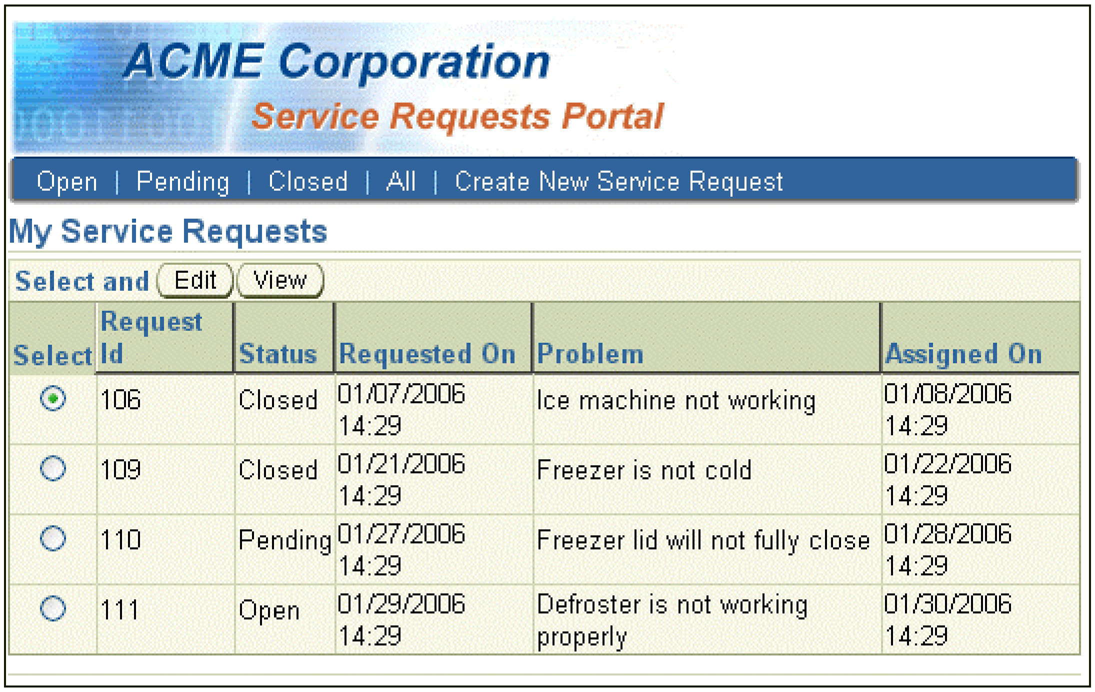
General	
NDValue	#{requestScope.Status}
NDName	TheStatus
NDOption	
NDType	java.lang.String

16. Use **Save All** to save your work.

Running the Page

Now run the page. With the SRList page open in the Visual Editor, right-click and select **Run**.

JDeveloper will compile the application, start the embedded Oracle OC4J application server, and launch your default browser to show you the running page. When the page appears, it should look like the image below.



My Service Requests

Select and Edit View

Select	Request Id	Status	Requested On	Problem	Assigned On
●	106	Closed	01/07/2006 14:29	Ice machine not working	01/08/2006 14:29
○	109	Closed	01/21/2006 14:29	Freezer is not cold	01/22/2006 14:29
○	110	Pending	01/27/2006 14:29	Freezer lid will not fully close	01/28/2006 14:29
○	111	Open	01/29/2006 14:29	Defroster is not working properly	01/30/2006 14:29

By default the table shows all rows (open or closed ones).

Notice the various UI components that you added to the page; note the values picked up from the `UIResources` file for the page title and the button label text. Notice that the label text control hints that we defined for our ServiceRequest entity object are automatically getting used for the table column titles.

The menu tabs that you defined are displayed in the menu bar along the top of the page. You added highlighted text to the selected request tab. By default all service requests are displayed. Test the menu option and the navigation link to the next page of records.

The View and Edit buttons will not work, since the destination pages have not been created yet.

Summary

In this chapter, you created a display page to enable users of the SRTutorial application to view information about service requests. To accomplish this, you performed the following key tasks:

- Created an outline page based on a template page
- Added user interface components to the page to display service request information
- Added View and Edit buttons for navigating to other pages in the application
- Added menus to enable users to select a service request by status
- Created Action Listeners to assign values to request-scope attributes
- Run and tested the page

6

Developing a Master-Detail Page

In this chapter, you develop a master-detail page that shows a service request and its Service Request History rows. From this page, users can see the scope and history of a service request. They can also add detailed notes to the service request.

The chapter contains the following sections:

Introduction

Each of the next chapters involves creating different JSF pages. Not only will you create the page, including the UI and navigation components, but also you build the dependent data model component which the page uses.

The SRMain page provides a master-detail view of Service Requests and their Service Request History rows. The page contains three component areas: the read-only Service Request form, the Notes input area, and the Service Request History table.

You perform the following key tasks in this chapter:

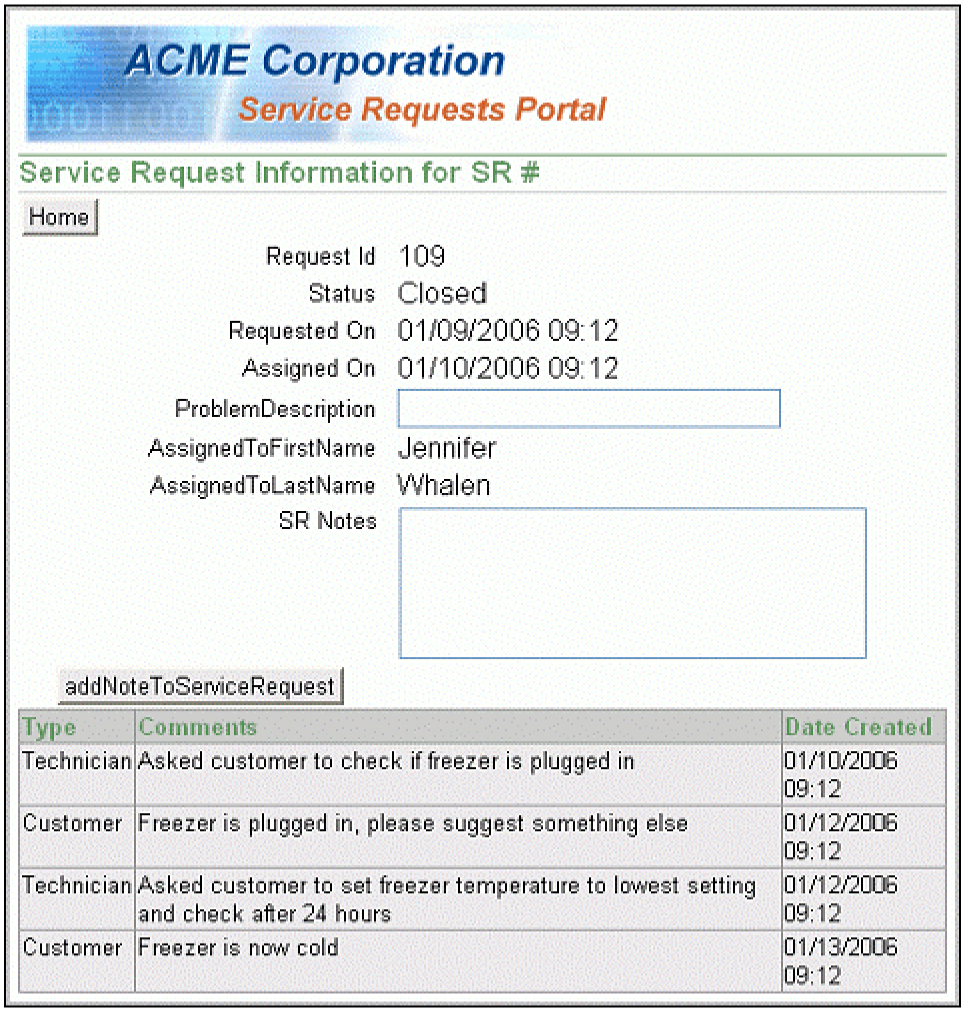
- Customize the ServiceRequest and ServiceHistory entity objects
- Create the ServiceRequest and ServiceHistories view objects
- Create the Service Request read-only form
- Create the Notes input form and add code to programmatically set data values from parameters
- Add the Service Request History table

Note: If you did not successfully complete Chapter 5, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named `Chapter6` to hold the starter application. If you used the default settings, it should be in `<jdev_install>\jdev\mywork\Chapter6`.
2. Unzip `<tutorial_setup>\starterApplications\SRTutorialADFBC-EndOfChapter5.zip` into this new directory. Using a new, separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRTutorial application workspace.
4. Select **File > Open**, and then select `<jdev_install>\jdev\mywork\Chapter6\SRDemo\SRTutorialADFBC.jws`. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 5.

Right-click the **UserInterface** node in the Navigator and select the **Open JSF Navigation** option from the context menu (alternatively, you can double-click the `faces-config.xml` file in the Navigator) to revisit the page-flow diagram that you created and see how the SRMain page relates to the other pages. The screen shot shows the completed SRMain page.



Service Request Information for SR #

Request Id	109
Status	Closed
Requested On	01/09/2006 09:12
Assigned On	01/10/2006 09:12
ProblemDescription	
AssignedToFirstName	Jennifer
AssignedToLastName	Whalen
SR Notes	

Type	Comments	Date Created
Technician	Asked customer to check if freezer is plugged in	01/10/2006 09:12
Customer	Freezer is plugged in, please suggest something else	01/12/2006 09:12
Technician	Asked customer to set freezer temperature to lowest setting and check after 24 hours	01/12/2006 09:12
Customer	Freezer is now cold	01/13/2006 09:12

Here are some points to note about the SRMain page:

- Template applied for standard look and feel
- A read-only form for the master record, with one of the fields converted to an input text field, and with another input text field added
- A parameter form with a button that enables users to add a note (detail record) to the service request (master) by executing a custom method added to the application module. The button action is augmented with custom code to clear the note input text field. An entity object must be modified to enable this functionality.
- A read-only table to display the detail records (service histories); the entity object

Other features of the page include:

- UI components exposed as a managed bean
- An action binding on the page to set the current row view object based on a parameter set by a previous page, which is done by creating an action listener on the previous page that places the key of its current row into a request-scoped variable.
- An executable binding on the page to invoke the action binding as part of page rendering.

Refining the Data Model

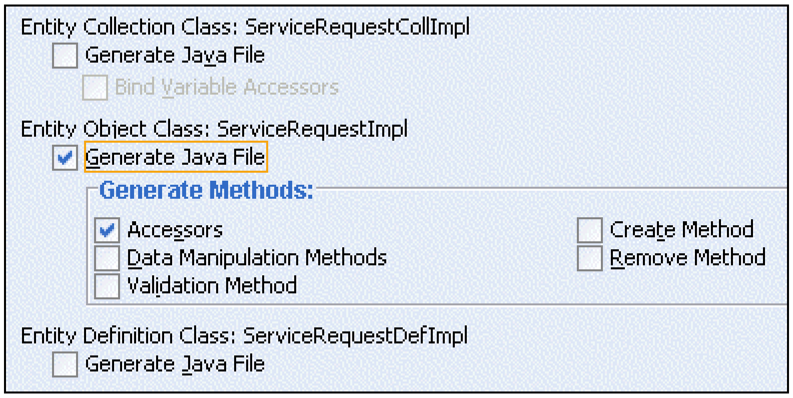
You can customize the data model to perform more complex operations. One thing you could do to make it easier for the end user, is to calculate and assign the next service history line number every time a new service history record is added. Another can be to populate the created date attribute with the current date.

To customize a class, you need to have access to the entity object implementation file. It is not necessary to create the object class file, unless you need to include customizations. In the next steps, generate the object class file for the ServiceHistory entity object and add custom code to increment the line number.

Customizing the Service Request Entity Object

Generate the class file for the ServiceRequest entity object. Customize the class to calculate the next line number.

1. In the Applications Navigator, expand the `DataModel > Application Sources > oracle.srtutorial.datamodel > entities` nodes.
2. Double-click the **ServiceRequest** entity object.
3. In the Entity Object Editor, select the **Java** node.
4. In the Entity Object Class: ServiceRequestImpl area, select the **Generate Java File** check box.



Entity Collection Class: ServiceRequestCollImpl
- [ ] Generate Java File
  - [ ] Bind Variable Accessors

Entity Object Class: ServiceRequestImpl
- [✓] Generate Java File

Generate Methods:
- [✓] Accessors
- [ ] Data Manipulation Methods
- [ ] Validation Method
- [ ] Create Method
- [ ] Remove Method

Entity Definition Class: ServiceRequestDefImpl
- [ ] Generate Java File

5. Click **OK**.
6. Select the **ServiceRequest** entity object and from the context menu, select **Go To Entity Object Class**. Notice that by default the file contains getter and setter methods for each of the attributes in the ServiceRequest entity object.
7. The `ServiceRequestImpl.java` file displays in the code editor. Scroll down to the end of the file and add the following method to the file.

```
public Number getMaxHistoryLineNumber() {
  RowSet hist = (RowSet)getServiceHistory();
  Number maxLine = new Number(0);
  while (hist.hasNext()) {
    Number curLine = (Number) hist.next().getAttribute("LineNo");
```

```
          if (curLine.compareTo(maxLine) > 0) {
            maxLine = curLine;
          }
        }
        hist.closeRowSet();
        return maxLine;
      }
```

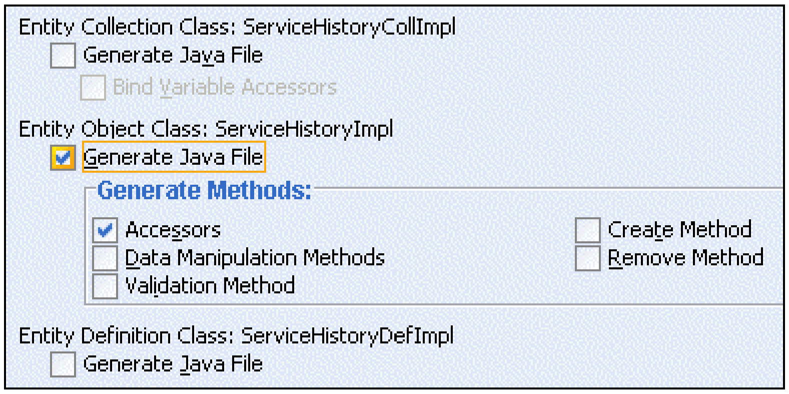
In this code, you access the row set of related ServiceHistory rows by using the `getServiceHistory()` getter method, and then iterate over the rows to determine the maximum line number currently in use by the related history rows.

8. When prompted, import **oracle.jbo.RowSet** using the **[Alt]** + **[Enter]** key combination.
9. **Save** your work.

Customize the Service Histories Entity Object

In the next steps, generate the java file for the ServiceHistory and customize it to set default values for all the attributes, except the Note. You add a method to create the note in the application module in a later step.

1. Double-click the **ServiceHistory** entity object.
2. In the Entity Object Editor, select the **Java** node.
3. In the Entity Object Class: ServiceHistoryImpl area, ensure that the **Generate Java File** check box is checked.



Entity Collection Class: ServiceHistoryCollImpl
☐ Generate Java File
☐ Bind Variable Accessors
Entity Object Class: ServiceHistoryImpl
☑ Generate Java File
Generate Methods:
☑ Accessors
☐ Data Manipulation Methods
☐ Validation Method
☐ Create Method
☐ Remove Method
Entity Definition Class: ServiceHistoryDefImpl
☐ Generate Java File

4. Click **OK**.
5. Select the **ServiceHistory** entity object and from the context menu, select **Go To Entity Object Class**.
6. The `ServiceHistoryImpl.java` file displays in the code editor. You need to override the default processing for this entity object when a new entity gets created. To override a method from a base class, click your cursor into the `ServiceHistoryImpl.java` class in the code editor and choose **Source > Override Methods...** from the main menu. Find the **`create(oracle.jbo.AttributeList)`** method in the list and check the box, then click **OK**.

7. Scroll down to the end of the ServiceHistoryImpl.java file and add the following code to the body of the overridden create() method, after the line super.create(…). This is custom code that will execute whenever an instance of the ServiceHistory entity gets created, after calling super.create() to perform the default functionality first.

```
protected void create(AttributeList nameValuePair) {
    super.create(nameValuePair);
    // Add these lines below
    setLineNo(getServiceRequest().getMaxHistoryLineNumber().add(1));
    setCreatedBy((Number)getDBTransaction().getSession().
          getUserData().get("CurrentUserId"));
    boolean isTech =
       getDBTransaction().getSession().isUserInRole("technician");
    setSvhType(isTech ? "Technician" : "Customer");
    }
```

You must add code to set the default value for the line number to increment by one. The code must also set the created by value, to default to the currently logged in user. This code programmatically defaults the **LineNo** attribute to the next available line number for the service request this line belongs to. This calculation is performed by calling getServiceRequest() to access the ServiceRequest entity object to which this new ServiceHistory entity instance belongs, calling getMaxHistoryLineNumber() on that service request, and adding one to the result. It also defaults the value of the **CreatedBy** attribute to the value of the CurrentUserId entry in the session user data hashtable. Finally, it uses the `isUserInRole()` method on the session object to determine whether the currently logged-in user is a technician or not, and then defaults the value of the **SvhType** attribute to either "Technician" or "Customer" as appropriate.

8. If prompted, import the following statement **`oracle.jbo.AttributeList`** using the [Alt] + [Enter] key combination.
9. Use **Save All** to save your work.

Creating the Data View Components

As with the SRList page, the SRMain page is based on view objects. The page has a master/detail presentation with the service request as the master, and the service histories as the detail. Even though you created a view object for the service request for the SRList page, you need a more complex one for the SRMain page.

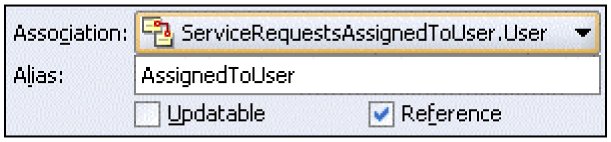
Creating the Service Request View

The SRMain page supports adding new service history records to a specific service request. Earlier, you created a ServiceRequest view objects for the SRList page. The ServiceRequest view object used on the SRList page contained a bind variable to satisfy the needs of that page to show the Open, Pending, Closed, or All status. The requirements for this new page are to edit any service request, regardless of its status, so that it is easier to create a new view object (based on the same underlying entity object) for this new page/purpose. It is good practice when your page needs a view object with a different data shape, or with different bind variable filtering, to create a new view object for that purpose. The requirements for the SRMain page are to view both service requests and their service histories. In this case create a new Service Request view object and link it to the ServiceHistories view object you created earlier.

1. In the Applications Navigator, right-click the **`oracle.srtutorial.datamodel > queries`** node, and select **New View Object** from the short cut menu.
2. Click **Next** on the Welcome page.
3. In the first step of the Create View Object Wizard, set the properties using the values from the table below. If the property is not listed below, then leave it at its default value.

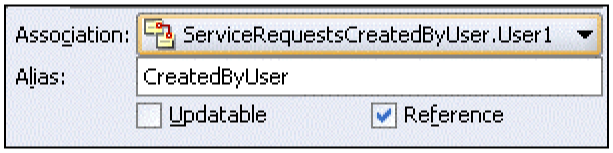
Property	Value
`Package`	`oracle.srtutorial.datamodel.queries`
`Name`	`ServiceRequestMain`

4. Click **Next**.
5. In Step 2: Entity Objects, expand the **`oracle.srtutorial.datamodel.entities`** node and shuttle the **ServiceRequest** entity to the Selected side.
6. Change the Alias property value to **`ServiceRequestMain`**.
7. Again under the **`oracle.srtutorial.datamodel.entities`** node, shuttle the **User** entity to the Selected side. It will appear second in the list of selected entities for this view object. This instance represents the user who is assigned to work on the service request.
8. Change the Alias property value to **`AssignedToUser`** and, in the Association property, ensure that the value is **`ServiceRequestsAssignedToUser.User`**.


Association: ServiceRequestsAssignedToUser.User
Alias: AssignedToUser
[ ] Updatable [✓ Reference]

Notice that by default, the second entity is marked as contributing **Reference** information to the view object's main ServiceRequest data. By default, the second and subsequent entity objects related to a view object get marked as reference information, which is appropriate for this tutorial, so you may leave that setting.

9. Shuttle a second instance of the User entity object to the Selected side. This instance, now the third entity in the selected list for this new view object being created, represents who created the service request.
10. Change the Alias property value to **`CreatedByUser`** and in the Association property set the value to the second association between the two entities, **`ServiceRequestsCreatedByUser.User1`**.

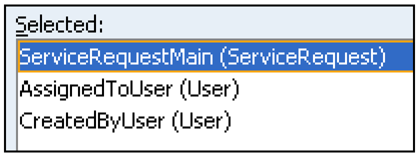


Association: ServiceRequestsCreatedByUser.User1

Alias: CreatedByUser

Updatable [✓ Reference]

11. The selected side should look like the image below. Select any of the existing instances to refresh the display.



Selected:

ServiceRequestMain (ServiceRequest)

AssignedToUser (User)

CreatedByUser (User)

12. Click **Next** to continue.

In the next steps, define the attributes required by this page which automatically determine the default SQL statement the view object will use.

13. In Step 3: Attributes, multi-select all attributes of the **ServiceRequestMain(ServiceRequest)** entity and shuttle them to the selected side.
14. From the **AssignedToUser** entity in the Available list, select the following attributes and shuttle them to the selected side.
    - `UserId`
    - `FirstName`
    - `LastName`
15. From the **CreatedByUser** entity, select the following attributes and shuttle them to the selected side.
    - `UserId`
    - `FirstName`
    - `LastName`
16. Click **Next** to continue.
17. In Step 4: Attribute Settings, change the Name property for the attribute names to the following values. Use the **Select Attribute** drop-down list at the top of the panel to switch between attributes as you modify their names.

Old Attribute Name	New Attribute Name
UserId	`AssignedToUserId`
FirstName	`AssignedToFirstName`
LastName	`AssignedToLastName`
UserId1	`CreatedByUserId`
FirstName1	`CreatedByFirstName`
LastName1	`CreatedByLastName`

18. Click **Next**. The default values for the SQL statement are fine and no Bind Variables are required in the view object.
19. Click **Finish** to complete the view object. It should now appear in the Applications Navigator.

Refining the ServiceRequestMain View

The service request view object makes the createdBy and assignedTo information available in a page. This page should display the service request record, even those when the service request record's assigned-to value is null.

2. In the Applications Navigator, expand the **`oracle.srtutorial.datamodel > queries`** node and double-click the **ServiceRequestMain** view object.
3. Expand the Attributes node, and select the **CreatedByUserId** attribute.
4. Deselect the **Key Attribute** check box
5. Deselect the **Key Attribute** check box for the **AssignedToUserId** attribute.
6. Select the **SQL Statement** node, and update the WHERE clause to reflect the outer-join for the ASSIGNED_TO field which can be null **(+)**. The WHERE clause should now look like:

```
(ServiceRequest.ASSIGNED_TO = AssignedToUser.USER_ID(+)) AND
(ServiceRequest.CREATED_BY = CreatedByUser.USER_ID)
```

7. Click **OK**.
8. Use **Save All** to save your work.

You now have the master component of the page created from the ServiceRequest entity object. Next create the link to connect the detail view object based on the Service History entity object.

Linking the Service History with Service Request

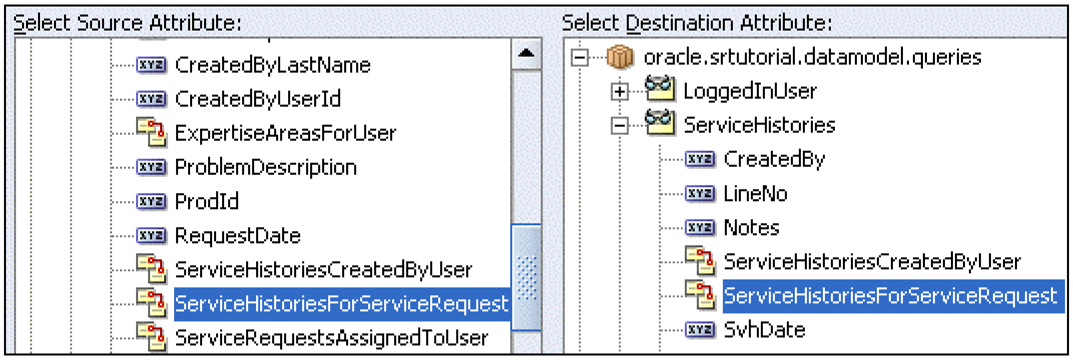
In the next few steps, create the view link to connect a service request to its service histories.

1. In the Applications Navigator, expand the **`oracle.srtutorial.datamodel > queries`** nodes.
2. Select the **`viewlinks`** node, and from the context menu select **New View Link**.
3. Click **Next** in the Welcome page, to continue.

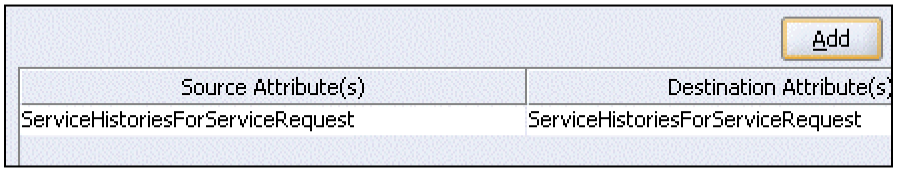
4. In the first step of the Create View Link Wizard, set the properties using the values from the table below. If the property is not listed below, then leave it at its default value.

Property	**Value**
`Package`	`oracle.srtutorial.datamodel.queries.viewlinks`
`Name`	`HistoryLinesForRequest`

5. Click **Next**.
6. For Step 2: View Objects, in the Source Attribute pane expand **`oracle.srtutorial.datamodel.queries`** and then the **`ServiceRequestMain`** query.
7. Scroll down and highlight the **`ServiceHistoriesForServiceRequest`** association.
8. In the Destination Attribute pane, expand **`oracle.srtutorial.datamodel.queries`** and then the **`ServiceHistories`** query.
9. Select the same **`ServiceHistoriesForServiceRequest`** association.


Select Source Attribute:
CreatedByLastName
CreatedByUserId
ExpertiseAreasForUser
ProblemDescription
ProdId
RequestDate
ServiceHistoriesCreatedByUser
ServiceHistoriesForServiceRequest
ServiceRequestsAssignedToUser
Select Destination Attribute:
oracle.srtutorial.datamodel.queries
LoggedInUser
ServiceHistories
CreatedBy
LineNo
Notes
ServiceHistoriesCreatedByUser
ServiceHistoriesForServiceRequest
SvhDate

10. With the same association selected on both sides, click the **Add** button. The link will appear in the lower pane of the step.



Source Attribute(s)	Destination Attribute(s)
ServiceHistoriesForServiceRequest	ServiceHistoriesForServiceRequest

11. Click **Next** then **Finish** to create the view link.

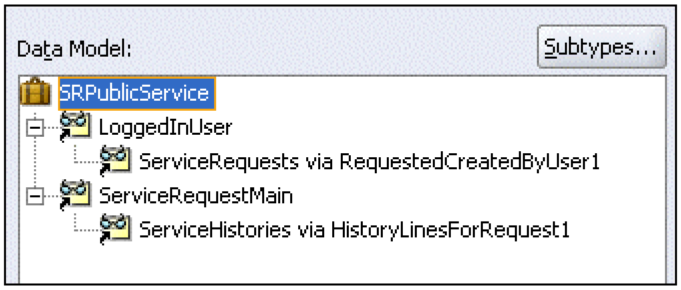
The new service request view object and the link to connect it to the service history view object are now complete. In the next step add the two view objects using the view link to the application module's data model.

Including the View in the Application Module

To make the view objects available to Faces pages, they need to be included in the SRPublicService application module. In the next steps, include the two view objects representing a service request and all its service history records into the SRPublicService application module.

1. In the Applications Navigator, open the `oracle.srtutorial.datamodel` node and double-click the `SRPublicService` application module
2. In the Application Module editor, select the **Data Mode**l node in the left pane.
3. In the Available View Objects pane, expand the `oracle.srtutorial.datamodel.queries` node.
4. Select the **ServiceRequestMain** node.
5. Change the name from **ServiceRequestMain1** to **ServiceRequestMain,** and shuttle it to the Data Model pane.
6. In the Data Model tree of view object instances on the right, select the **ServiceRequestMain**. Then in the Available pane, expand the **ServiceRequestMain** node and select the nested **ServiceHistories via HistoryLinesForRequest** node.
7. Change the name from **ServiceHistories1** to **ServiceHistories,** and shuttle it to the Data Model pane. The Data Model pane should look like the image below.

Notice that you can also change the name in the Instance Name field of the Data Model list after shuttling it.


Data Model:
Subtypes...
SRPublicService
LoggedInUser
ServiceRequests via RequestedCreatedByUser1
ServiceRequestMain
ServiceHistories via HistoryLinesForRequest1

8. Click **OK** to update the application module.
9. Use **Save All** to save your work.
10. In the Application navigator, right click `SRPublicService` and select **Test** from context menu.
11. In the Business Component Browser dialog box, ensure that the Connection Name is set to **srdemo**, and then click **Connect**.
12. In the Oracle Business Component Browser (Local) window, double-click the **HistoryLines For Request1** to make sure the master/detail is working.
13. Close the browser window.

Adding a Custom Method to the Application Module

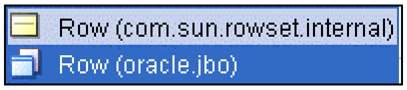
The final data model refinement you will complete is to add a method to create a service history note. In the SRMain page, users add a description of what they did to fix the problem or move it along. The rest of the service history values are then defaulted. In ADF applications using ADF Business Components, it is best practice to encapsulate any application logic inside a custom application module method when that logic is programmatically manipulating the view objects in the application module's data model. This makes the application easier to test and maintain.

In the next steps, modify the application module and add the method to create a note.

1. In the Application navigator, right-click the **SRPublicService** node.
2. From the context menu, select **Go To Application Module Class**. The **SRPublicServiceImpl.java** file will display in the editor.
3. Scroll down to the bottom of the file and add a new method to create a note. Copy and paste the method code shown below into the **SRPublicServiceImpl.java** file.

```
public void addNoteToServiceRequest(String noteText) {
      Row newHistory = getServiceHistories().createRow();
      getServiceHistories().last();
      newHistory.setAttribute("Notes",noteText);
      getDBTransaction().commit();
}
```

4. When prompted, use [Alt] + [Enter] to import the `Row(oracle.jbo)` class.



5. Use **Save All** to save your work.

Now that you have a custom method for adding a note, you must expose it as a data control. In the next steps, expose the new method to the application module so it can be used by clients through the data control.

6. Double-click the **SRPublicService** application module.
7. Select the **Client Interface** node, and shuttle the **addNoteToServiceRequest** to the Selected side.
8. Click **OK**.

All of the data model components for the SRMain page have now been created. The ServiceRequestMain and ServiceHistories view objects are defined and linked together using the HistoryLinesForRequest view link. Both view objects, the link and custom code to create a note have been included in the SRPublicService application module. You are ready to build the SRMain page.

Developing the Basic User Interface

In the next part of this chapter, you create the SRMain page and add the basic UI components. These are the layout components that you will use to hold the data-aware components that you add later.

Perform the following steps to create the SRMain page and copy the template you created earlier into this page:

1. If it is not open, right-click the **UserInterface** project and select **Open JSF Navigation** from the context menu to view the page-flow diagram.
2. Double-click the **/app/SRMain.jspx** page to invoke the JSF Page Wizard.

3. Ensure that the values for the first three steps of the wizard match those in the following tables:

Wizard Step 1: JSP File

Field	Value
File Name	`SRMain.jspx`
Directory Name	This is the location where the file is stored. Ensure that you create the page in the `\SRTutorialADFBC\UserInterface\public_html\app` folder.
Type	JSP Document (*.jspx)
Mobile	Clear the check box.

4. Click **Next** to continue.

Wizard Step 2: Component Binding

Field	Value
Do Not Automatically Expose UI Components in a New Managed Bean	Ensure that this option is selected.

5. Click **Next** to continue.

Wizard Step 3: Tag Libraries

Field	Value

Selected Libraries	`ADF Faces Components`
	`ADF Faces HTML`
	`JSF Core`
	`JSF HTML`

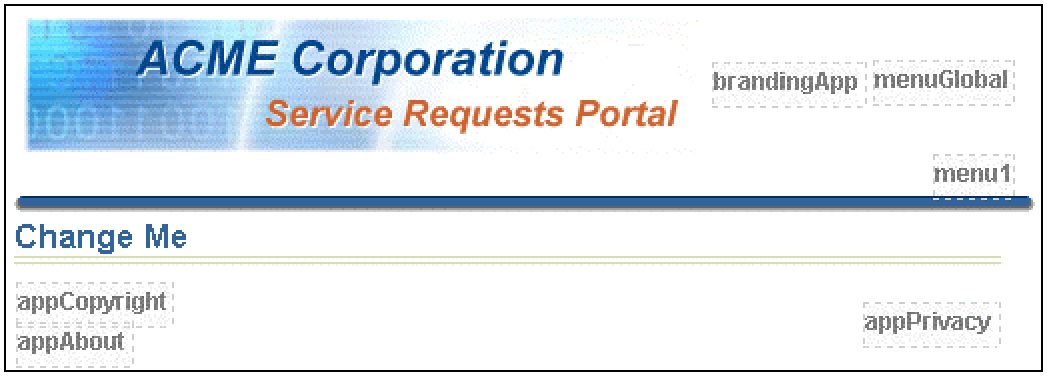
6. Click **Finish** to create the page details. The new SRMain page is displayed in the Visual Editor.

7. Open **SRTutorialTemplate.jspx** in the Visual Editor (if it is not already open). In the Structure window, collapse the **afh:html** node and select it. From the shortcut menu, choose **Copy**.

8. Click the tab to return to the **SRMain** page, and in the Structure window select **`f:view`** node.

9. Delete the **`html`** node. Then right-click **`f:view`** and choose **Paste** from the shortcut menu. The look and feel that you created earlier is now applied to the new page.

Your page should now look like the following:

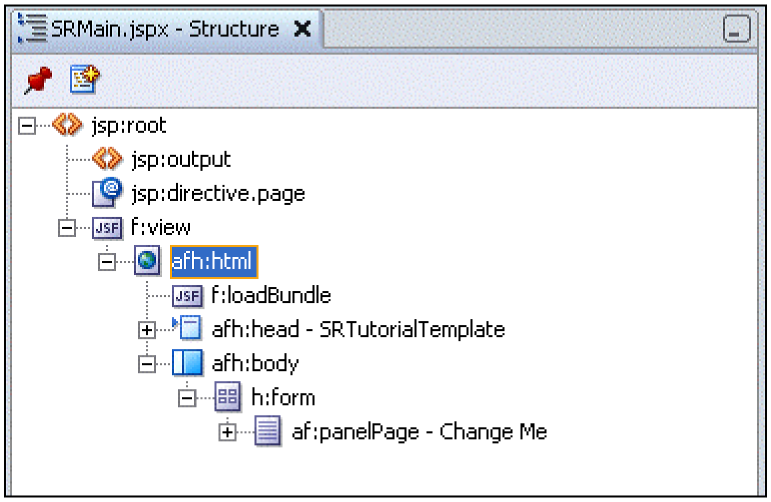


Creating Data Components

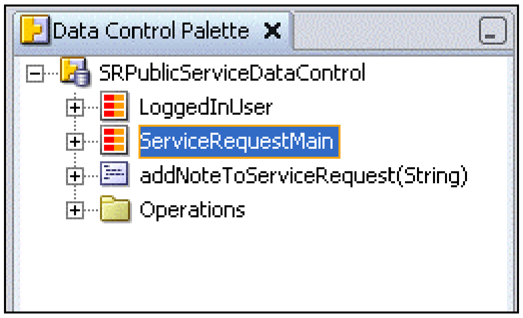
Now that you have created the basic page, you can begin to add data-aware components that display the service request data.

Adding Service Request Data

1. In the Structure pane, expand the **f:view > afh:html > afh:body > h:form** nodes.

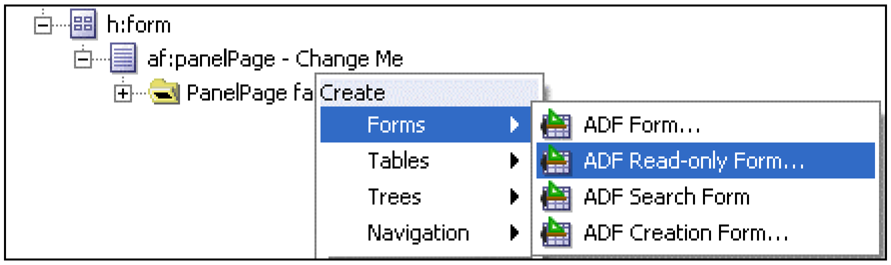


2. In the Data Control palette, expand the **SRPublicServiceDataControl** node and select the **ServiceRequestMain** node.



3. Drag the **ServiceRequestMain** view object and drop it on the **af:panelPage** node in the Structure pane.

4. In the context menu, select **Forms > ADF Read-only Form**.

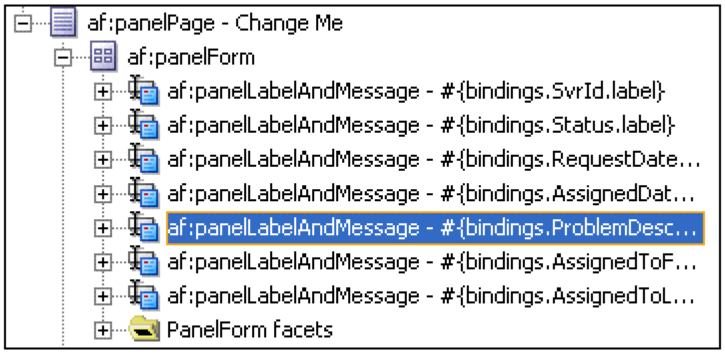


5. In the Edit Form Fields pane, keep the following attributes and delete all the rest. Resequence the attributes using the (Up) and (Down) buttons into the following order: **SvrId**, **Status**, **RequestDate**, **AssignedDate**, **ProblemDescription**, **AssignedToFirstName** and **AssignedToLastName.** Multiple select will work to delete more than one field.

6. Click **OK**.

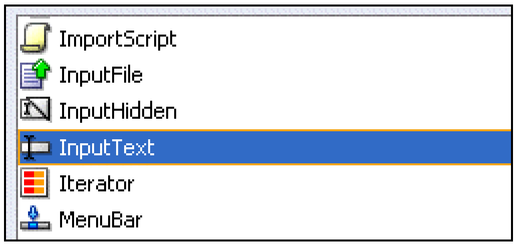
Another way to modify the component type of a field is to convert it. The next steps shows you how to convert a field once its been added to a page. If you already modified the component type for the ProblemDescription, you do not need to complete these steps. However, you can use this process anytime during the development of a page.

7. In the Structure pane, expand the new **af:panelForm** and select the **af:panelLabelAndMessage** for the **ProblemDescription** attribute.


af:panelPage - Change Me
af:panelForm
af:panelLabelAndMessage - #{bindings.SvrId.label}
af:panelLabelAndMessage - #{bindings.Status.label}
af:panelLabelAndMessage - #{bindings.RequestDate...
af:panelLabelAndMessage - #{bindings.AssignedDat...
af:panelLabelAndMessage - #{bindings.ProblemDesc...
af:panelLabelAndMessage - #{bindings.AssignedToF...
af:panelLabelAndMessage - #{bindings.AssignedToL...
PanelForm facets

8. Right-click the **af:panelLabelAndMessage:ProblemDescription** and select **Convert**, from the context menu.

9. Scroll down and select the **Input Text** item.



10. Click **OK** to convert the Problem Description to an Input Text item.

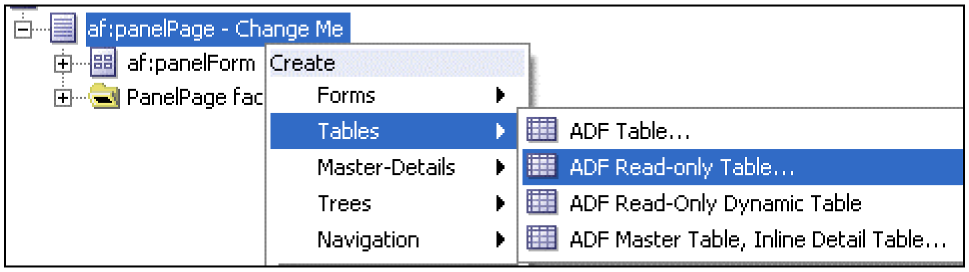
11. Select the ProblemDescription field, and in the Property Inspector, set the **Rows** property to **4**.

12. **Save** your work.

Adding the ServiceHistories Data

In the next steps, you create the detail data display for the Service History records. Typically, detail components in a master/detail page are displayed as a table. This can be accomplished easily with an ADF Table.

1. In the Data Control palette, expand the **ServiceRequestMain** node.
2. Select the nested **ServiceHistories** node and drag it to the Structure pane.
3. Drop it on top of the **af:panelPage**.
4. In the context menu, select **Tables > ADF Read-only Table**.



5. In the Edit Table Columns dialog, keep only the following attributes in the following order: **SvhType**, **Notes** and **SvhDate**.
6. Click **OK**.

In the next steps, create an action binding to set the current row in the ServiceRequestMain view object to the one you want to view based on a parameter set by a previous pages. First you set up the action binding, then you return to the previous page to set the parameter that the action binding is expecting to receive.

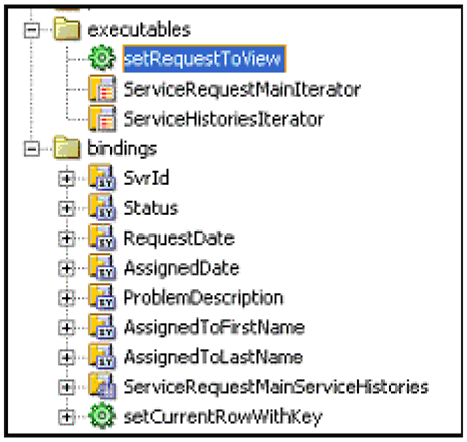
7. Right-click anywhere on the page and in the context menu, select **Go To Page Definition**.
8. In the Structure pane, right-click the **bindings** node, and select **Insert inside bindings > action**.
9. An Action Bindings Editor is invoked. Expand the **SRPublicServiceDataControl** and select the **ServiceRequestMain** node.
10. In the Select an Action drop-down list, scroll down and select **setCurrentRowWithKey**.
11. In the Parameters area, set the **Value** property to #**{requestScope.requestRowKey}**. This is the EL expression to reference an attribute named **requestRowKey** on the request scope that we will set in the calling page before navigating to the SRMain page.
12. Click **OK**.

Now that you have a binding for the current row, add an executable to invoke the action binding you just created. This executable will set the current row key automatically as part of the page rendering.

13. In the Structure pane, expand the **executables** node, then right-click the **ServiceRequestMainIterator** node, and select **Insert before ServiceRequestMainIterator > invokeAction**.
14. In the Insert invokeAction dialog, set the ID to `setRequestToView` and Binds to the **setCurrentRowWithKey** action binding you just created.
15. In the Advanced Properties tab, set the Refresh property to **prepareModel** and the RefreshCondition to `#{adfFacesContext.postback == false}`. The Refresh property controls when the invokeAction executes during the processing of the page. The **prepareModel** setting indicates that you want it to be invoked during the "prepare model" phase when ADF is preparing the data to display. The RefreshCondition is a Boolean-valued EL expression that further refines when the invokeAction will execute. The adfFacesContext.postback flag will be True when the user has interacted with some UI controls on the current page. It will be False when the page is rendering for the first time when navigating from another page. By setting the RefreshCondition to the above expression, you ensure that it only fires when the user navigates to SRMain from another page.

    NOTE: Sometimes the Insert InvokeAction dialog box "complains" about a missing ID field, when you clearly have already entered the ID value. As a workaround add a space to the end of the ID field, and then delete that space again to "dirty" the field value.
16. Click **OK**.
17. **Save** your work.

You now have a binding to the Service Request current record and an executable to receive the current row from the SRList page. The Structure pane should look like the following image.

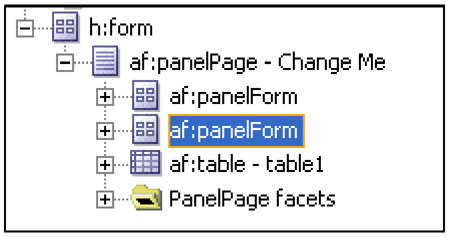


Adding an Add Note Component

Go back to the SRMain page and create a data component to add a note to the service request. In a previous section, you created the addNoteToServiceRequest method in the SRPublicService application module. In the next steps, use the method and create the note item on the page.

1. Use the tab to bring the **SRMain.jspx** to the forefront in the editor.

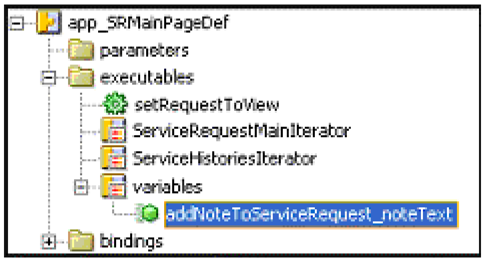
2. In the Data Control palette, drag the **addNoteToServiceRequest(String)** method to the Structure pane.
3. Drop it on the **af:panelPage** node.
4. In the context menu, select **Parameters > ADF Parameter Form**.
5. Accept all the default values in the Edit Form Field dialog, and click **OK**.
6. Sequence the new **af:panelForm** before the **af:table**. It should look like the image below.



7. Expand the new **af:panelForm** and select the **af:inputText** item for the note.
8. Set the Rows property to **4**.

When you dropped the ADF Parameter Form, JDeveloper created a page definition variable to temporarily hold the value the user will enter for the noteText parameter of the **addNoteToServiceRequest ()** method. In the next steps, set some display properties on that page definition variable to ensure a meaningful default label displays for the note item.

9. Bring the **app_SRMainPadeDef.xml** page definition file to the forefront of the editor.
10. Expand the **executables > variable nodes**.
11. Double-click the **addNoteToServiceRequest_noteText** and set the following Control Hint properties.



Property	**Value**
Label Text	`SR Notes`
Display Width	`35`

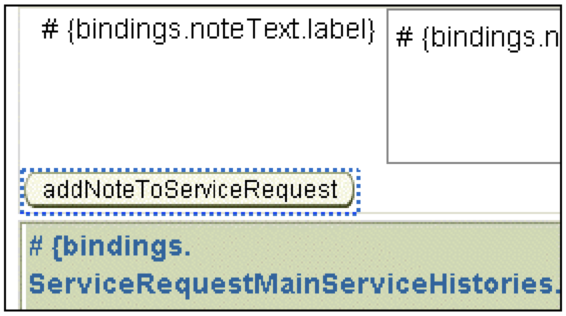
12. Click **OK**.

When you added the custom method to create a note to the page, a button was included on the page to control adding a note to the service history. So far the button does not perform any action. In the next steps refine what happens when the button is clicked.

Customizing Button Action

A new button is on the page, and when clicked the button populates the addNoteToServiceRequest method parameter with the value entered in the note item. It must also clear out the note item and be ready to accept another service history note. All of these behaviors are controlled using a managed bean. In the next steps, generate the binding code for the item and customize it to clear out the item value.

1. Open the **SRMain** page in the editor.
2. Double-click the **addNoteToServiceRequest** button.



# {bindings.noteText.label}
# {bindings.n
addNoteToServiceRequest
# {bindings.
ServiceRequestMainServiceHistories.

3. The Bind Action Property dialog opens. Click **New** next to the Managed Bean field.
4. In the Create Managed Bean dialog enter the following values, and then click **OK**.

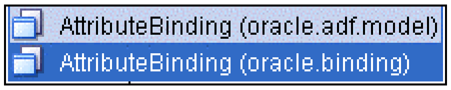
Field	Value
Name	`backing_app_SRMain`
Class	`SRMain`
Scope	`request`
Generate Class If It Does Not Exist P	`<check box checked>`

5. The Bind Action Property dialog shows the name of the managed bean class containing the method. The method name is derived from the name of the button with '_action' appended to the end. The default value should be commandButton_action and is fine.
6. Click **OK**. The SRMain.java backing bean class is now opened in the editor. The cursor is placed in the new method commandButton_action. Notice the binding container code and the get operation for the addNoteToServiceRequest.
7. Open up a few lines after the first **return null ; }** and add the code below to clear out the value in the note item. The code below is for the entire method with the code you add in bold.

```
public String commandButton_action() {

   BindingContainer bindings = getBindings();

  OperationBinding operationBinding =
bindings.getOperationBinding("addNoteToServiceRequest");

  Object result = operationBinding.execute();
```

```
          if (!operationBinding.getErrors().isEmpty()) {
            return null;
          }
          AttributeBinding noteText =
        (AttributeBinding)getBindings().getControlBinding("noteText");
          noteText.setInputValue(null);
          return null;
        }
```

8. When prompted, import the **oracle.binding.AttributeBinding** class.


AttributeBinding (oracle.adf.model)
AttributeBinding (oracle.binding)

9. **Save** your work.

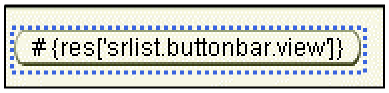
When the button is now clicked the value in the note item will be bound to the method's parameter and created with a service history record.

Linking the Page to the Application

The page will add service history records to the current service request being viewed. However there is no facility yet to determine what service request should be displayed in the SRMain page. The **`#{requestScope.requestRowKey}`** expression referenced by the setCurrentRowWithKey action binding you created above will evaluate to *null* unless you set its value to the key of the desired row to view on the calling page. In the next section modify the SRList page to have the (View) button set the value of **`requestScope.requestRowKey`** to the key of the selected record before navigating to the SRMain page. Then on the SRMain page, add a button to navigate back to the home page, SRList.

1. In the Applications Navigator, double-click the **SRList** page to view it in the visual editor.

2. Right-click the view button, and from the context menu select **Insert inside CommandButton > ADF Faces Core > SetActionListener**.



3. Set the **From** property to **`#{row.rowKeyStr}`** and the **To** property to **`#{requestScope.requestRowKey}`**
   The #**{row.rowKeyStr}** expression represents the key of the current row in the table. Note that this key is not literally the value of the primary key like "101," but instead is a particular encoded-string format for the key that supports situations where the primary key comprises multiple attributes.

5. Click **OK**.

In the SRMain page, add a new button to return to the SRList page.

6. If not already available, open the **SRMain** page.

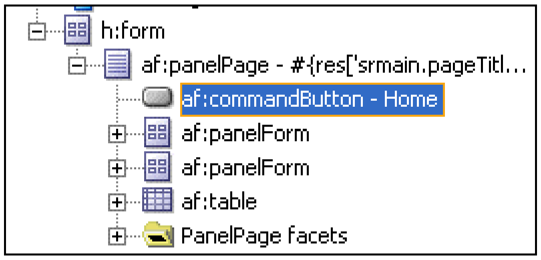
7. In the Structure pane, select the af:panelPage, and set the **Title** property to **`#{res['srmain.pageTitle']}`**.

8. Right-click the af:panelPage, and from the context menu select **Insert inside af:panelPage > ADF Faces Core**.

9. From the pop up pane, select **CommandButton** and click **OK**.

10. Set the Text property to **Home**, the Action property to **Home,** which is the global navigation rule, and the Immediate property to **true**.

11. In the Structure pane, move the new **af:commandButton** before the first af:panelForm immediately below the af:panelPage.



12. Use **Save All** to save your work.

Run and Test the Pages

Now that you have two pages, run them and test the functionality.

1. In the faces-config file, right-click the **SRList** page and select **Run**.

2. Select a record and click the **View** button. The SRMain page should be invoked, and the service request ID passed into the SRMain page.

3. Enter a value for the Note item, and click the **addNoteToServiceRequest** button. Notice a new service history record is added to the table.

4. Click the **Home** button to return to the SRList page.

Summary

In this chapter, you created a master-detail page using ADF Faces components. Those components display data that is coordinated between two panels on the page.

Here are the key tasks that you performed in this chapter:

- Refined the data model's basic user interface
- Added service request components
- Added the Service Histories panel
- Added the Notes panel
- Linked the page to the application

7 Implementing Transactional Capabilities

This chapter describes how to build the pages to create a service request. The service request process involves three main pages: one to specify the product and problem, one to confirm the values, and one to commit and display the service request ID.

The chapter contains the following sections:

Introduction

Transactional operations for creating and deleting a record are very similar in process. When creating a record, users enter information, click a create button, and then receive a confirmation of the creation. When deleting a record, users select the record, click a delete button, and then receive a confirmation of the deletion. In both cases, an additional step can be included to ask users if they are sure of their action.

In this chapter, you build four screens:

- A page to create or delete a service request record (SRCreate)
- A page to confirm the action, including the newly created service request values (SRCreateConfirm)
- A page to acknowledge the successful creation (SRCreateDone)

You perform the following key tasks in this chapter:

- Build the Create, Confirm, and Done pages
- Refine the prompts to get their values from UIResources.properties
- Add components for the data-bound objects
- Add command buttons to control transactions
- Manage transactions values
- Pass parameters and navigate between pages
- Define and activate a process train to show the progress of the create process

Note: If you did not successfully complete Chapter 6, you can use the end-of-chapter application that is part of the tutorial setup.

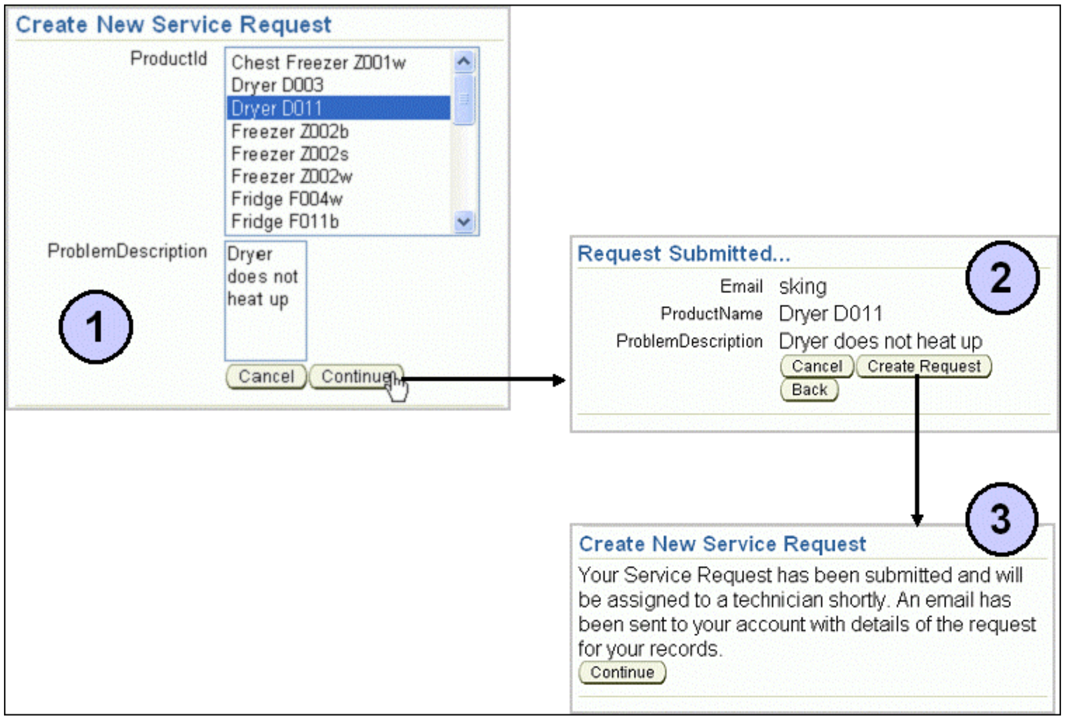
1. Create a subdirectory named `Chapter7` to hold the starter application. If you used the default settings, it should be in `<jdev_install>\jdev\mywork\Chapter7`.
2. Unzip `<tutorial_setup>\starterApplications\SRTutorialADFBC-EndOfChapter6.zip` into this new directory. Using a new, separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRTutorial application workspace.
4. Select **File > Open**, and then select `<jdev_install>\jdev\mywork\Chapter7\SRDemo\SRTutorialADFBC.jws`. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 6.

Right-click the **UserInterface** node in the Navigator and select the **Open JSF Navigation** option from the context menu (alternatively, you can double-click the **`faces-config.xml`** file in the Navigator) to revisit the page-flow diagram that you created and see how the Create pages relates to the other pages.

Transactional operations for creating and deleting a record are very similar in process. When creating a record, users enter information, click a create button, and then receive a confirmation of the creation. When deleting a record, users select the record, click a delete button, and then receive a confirmation of the deletion. In both cases, an additional step can be included to ask users if they are sure of their action.

The diagram below highlights how the pages work together:



The Service Request application contains three pages that implement creating a record:

- A page to create or delete a service request record (SRCreate)
- A page to confirm the action, including the newly created service request values (SRCreateConfirm)
- A page to acknowledge the successful creation (SRCreateDone)

Refining the Data Model

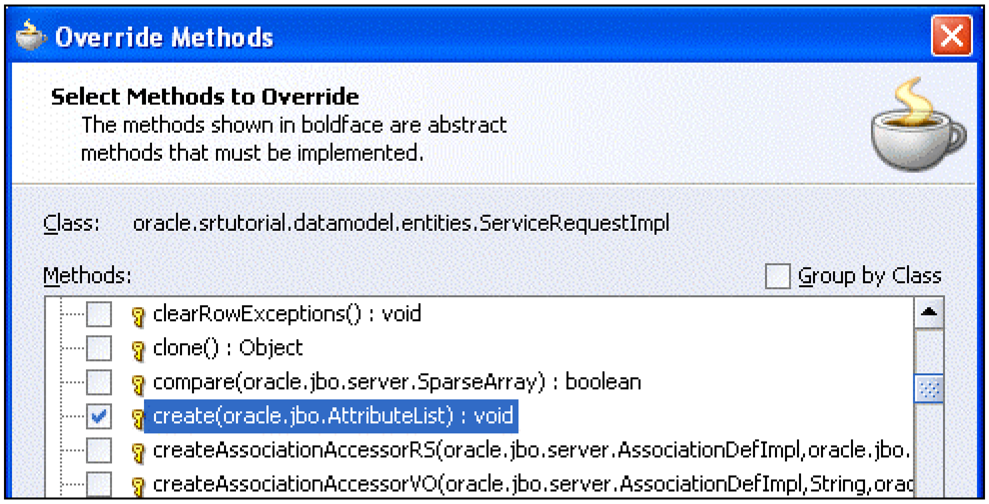
You can customize the data model to perform more complex operations. In Chapter 2: Developing Data Model Objects you made it easier for the end user by setting the created date attribute with the current date value. Another requirement implemented in Chapter 2 was to set the default value for the Status attribute to Open. In many cases the requirements can be implemented, declaratively.

Implementing other requirements may require adding a bit of custom code to complement the default declarative functionality. In the SRMain chapter, you exposed the ServiceRequestImpl.java file to make some code modifications. In this section you override the create() method to default the CreatedBy attribute of a service request to the userid of the currently logged on user:

Customize the Service Request Entity Object

In the next steps, generate the java file for the ServiceRequest entity object and customize it to set the default value of CreatedBy to the logged on user.

1. In the Applications Navigator, expand the following nodes **`DataModel > Application Sources > oracle.srtutorial.datamodel.entities`**, select the **`ServiceRequest`** entity object, and from the context menu, select **Go To Entity Object Class**.
2. With the ServiceRequestImpl.java file displaying in the code editor, select **Source > Override Methods** from the menu bar.
3. Scroll down the Override Methods list and select the **`create(oracle.jboAttributeList):void`** check box.


Override Methods
Select Methods to Override
The methods shown in boldface are abstract methods that must be implemented.

Class: oracle.srtutorial.datamodel.entities.ServiceRequestImpl

Methods: ☐ Group by Class

☐ clearRowExceptions() : void
☐ clone() : Object
☐ compare(oracle.jbo.server.SparseArray) : boolean
☑ create(oracle.jbo.AttributeList) : void
☐ createAssociationAccessorRS(oracle.jbo.server.AssociationDefImpl,oracle.jbo.
☐ createAssociationAccessorVO(oracle.jbo.server.AssociationDefImpl,String,orac

4. Click **OK** to be taken to that section of code.
5. Add the following code to the already existing method, after calling the super.create() method, which causes the entity object to perform its default processing for the create() method before our custom code executes.

```
protected void create(AttributeList AttributeList) {
 super.create(AttributeList);
setCreatedBy((Number)getDBTransaction().getSession().getUserData()
.get("CurrentUserId"));
}
```

6. **Save** your work.

Creating Data View Components

As with the previous pages, the SRCreate pages are based on view objects. A view object retains its values even across pages. In this section, create a view object to temporarily hold the details for the new service request while the user moves back and forth between the create page and the confirm page. Later, you will write a method to use this temporary information to create a new service request. .

Creating the Globals View

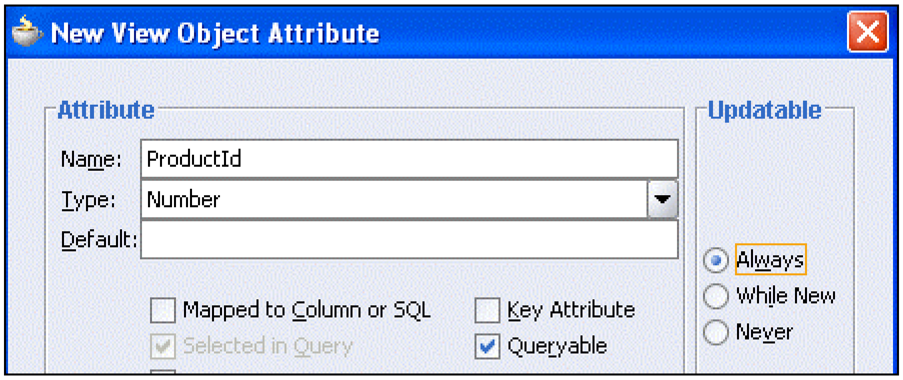
The SRCreate page allows users to select a product for the service request and the SRCreateConfirm page displays the product name. Then these pages will be used to create a new ServiceRequest entity object that will save the details to the SERVICE_REQUESTS table when the transaction is committed. In the next steps, create a view object with transient attributes to hold the current value of the product ID, name, and problem description.

1. In the Applications Navigator, right-click the **`oracle.srtutorial.datamodel.queries`** node, and select **New View Object** from the shortcut menu.
2. Click **Next** on the Welcome page.
3. In the first step of the Create View Object Wizard, set the properties using the values from the table below. If the property is not listed below, then leave it at its default value.

Property	Value
Package	`oracle.srtutorial.datamodel.queries`
Name	`Globals`
Extends	`<null>`
Rows Populated Programmatically, not Based on a Query	`Radio button` <Selected>

4. Click **Next**
5. In Step2: Attributes, click the **New** button.

6. Set the Name property to **ProductId**, set the type to **Number** and set the Updatable property to **Always**. The rest of the default values do not need to be changed.



Name: ProductId
Type: Number
Default:
Updatable: [✓ Always]
Queryable [✓]

7. Press **OK** to create the attribute.
8. Create two more attributes, **ProductName** and **ProblemDescription,** using the tables below.

Attribute Name	Type	Updatable
`ProductName`	`String`	`Always`
`ProblemDescription`	`String`	`Always`

9. Click **Finish** to create the **Globals** view object. The view does not need to interact with the database, so no SQL statement is needed. At run time, it supports holding programmatically populated rows containing the three attributes you defined. In this application, you will use just a single row in the Globals view to temporarily hold the ProductId, ProductName, and ProblemDescription values as we pass from one page to another.
10. When prompted, click **OK** to acknowledge that the view object is now in Expert mode.
11. In the Applications Navigator, right-click **Globals** view object and select the **Edit Globals** option from the context menu.
12. In the View Object Editor, expand the **Attributes** node and the **ProblemDescription** attribute, using the Control Hints tab to set the display width to **35**.
13. In the left side, click the **Tuning** node and, in the Retrieve from the database section, select the **No Rows** option. The view object receives its values from the SRCreate page, not the database. All the view object needs to do is hold the values from the SRCreate page, making them available in the SRConfirm page.
14. Click **OK** to exit the dialog.
15. Save your work.

Creating a Product List View

In the SRCreate page, users select a product to associate with the service request. In the next steps, create a read-only view object to retrieve the list of valid product IDs and product names. The view will then be used in displaying all the products on the SRCreate page.

1. In the Applications Navigator, right-click the **`oracle.srtutorial.datamodel > queries`** node, and select **New View Object** from the shortcut menu.
2. Click **Next** on the Welcome page.
3. In the first step of the Create View Object Wizard, set the properties using the values from the table below. If the property is not listed below, then leave it at its default value.

Property	Value
`Package`	`oracle.srtutorial.datamodel.queries`
`Name`	`ProductsList`
`Extends`	`<null>`
`Rows Populate by a SQL Query with:`	`Read-only Access` <Radio button selected>

   In this type of view object, you write your own SQL query rather than defining it declaratively.

4. Click **Next**.
5. In Step 2: SQL Statement, copy the code below and paste it into the Query Statement area.

```
SELECT PROD_ID, NAME
FROM PRODUCTS
ORDER BY NAME
```

6. In Step 3: Bind Variables, click **Next**.
7. In Step 4: Attribute Mapping, click **Finish**.
8. **Save** your work.

Including the Views into the Application Module

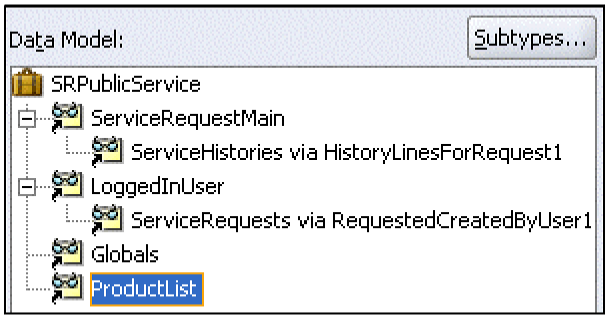
To make the view objects available to ADF Faces pages, they need to be included in the SRPublicService application module.

In the next steps, add the Globals and ProductList view objects to the SRPublicService application module.

1. In the Applications Navigator, open the **`oracle.srtutorial.datamodel`** node and double-click the **`SRPublicService`** application module
2. In the Application Module editor, select the **Data Model** node.
3. In the Available pane, expand the **`oracle.srtutorial.datamodel.queries`** node.

4. Select the **Globals** node, change the Name from **Globals1** to **Globals** and shuttle it to the Data Model pane.
5. In the `oracle.srtutorial.datamodel.queries` select the **ProductList** node, change the name from **ProductList1** to **ProductList**, and then shuttle the **ProductList** view object to the Selected side.

   The Data Model pane should look like the following image.



```
SRPublicService
  ServiceRequestMain
    ServiceHistories via HistoryLinesForRequest1
  LoggedInUser
    ServiceRequests via RequestedCreatedByUser1
  Globals
  ProductList
```

6. Click **OK.**

Adding Custom Methods to the Application Module

The final data model refinement you need is to add methods to ensure that there is a single blank row in the Globals view object. You will now do this.

1. In the Application navigator, in the `oracle.srtutorial.datamodel` node, right-click `SRPublicService` application module name.
2. From the context menu, select **Go To Application Module Class**. The **SRPublicServiceImpl.java** file will display in the editor.
3. Scroll down to the bottom of the file and add a new method to create a note. Copy and paste the following method code into the **SRPublicServiceImpl.java** file.

```
public void insureOneBlankRowInGlobals() {
 getGlobals().clearCache();
 getGlobals().insertRow(getGlobals().createRow());
}
```

4. Create another method to create a new service request based on the values in the transient attributes from the Globals row. Scroll down to the bottom of the **SRPublicServiceImpl.java** file and copy and paste the following method code.

```
  public void createNewServiceRequestFromGlobals() {
    ServiceRequestImpl sr =
(ServiceRequestImpl)getDBTransaction().createEntityInstance(ServiceReques
tImpl.getDefinitionObject(),null);
    Row globalsRow = getGlobals().first();
sr.setProblemDescription((String)globalsRow.getAttribute("ProblemDescript
ion"));
    sr.setProdId((Number)globalsRow.getAttribute("ProductId"));
    getDBTransaction().commit();
 }
```

Notice that the code inside the application module here can easily access the Globals view object, its first (and only) row, and the values in the attributes of that row without requiring the client to pass the data as parameters to the **`createNewServiceRequestFromGlobals()`** method.

5. When prompted, import the **`oracle.srtutorial.datamodel.entities.ServiceRequestImpl`** class.
6. Add one more import statement, at the end of the import list, to support the setProdId method:

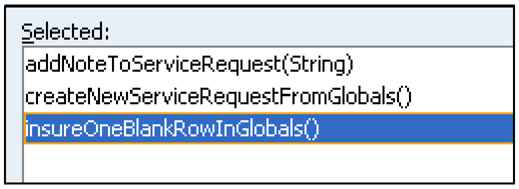
   ```
   import oracle.jbo.domain.Number;
   ```
7. Use **Save All** to save your work.

Including Custom Methods into the Application Module

To make the new methods available to ADF Faces pages, they need to be included in the SRPublicService application module's client interface.

In the next few steps, add the custom code from the `insureOneBlankRowInGlobals` and `createNewServiceRequestFromGlobals` methods, which then become available in the application module.

1. In the Applications Navigator, open the **`oracle.srtutorial.datamodel`** node and double-click the **`SRPublicService`** application module.
2. Select the **Client Interface** node, and shuttle the two new methods to the Selected side. The Selected side should look like the following image.



Selected:
addNoteToServiceRequest(String)
createNewServiceRequestFromGlobals()
insureOneBlankRowInGlobals()

3. Click **OK** to update the application module.
4. Use **Save All** to save your work.

All of the data model components for the SRCreate pages have now been created. The Globals and ProductList view objects are defined. Both view objects and some custom code have been included in the SRPublicService application module, and you have included those methods in the client interface for the application module. You are ready to build the SRCreate pages.

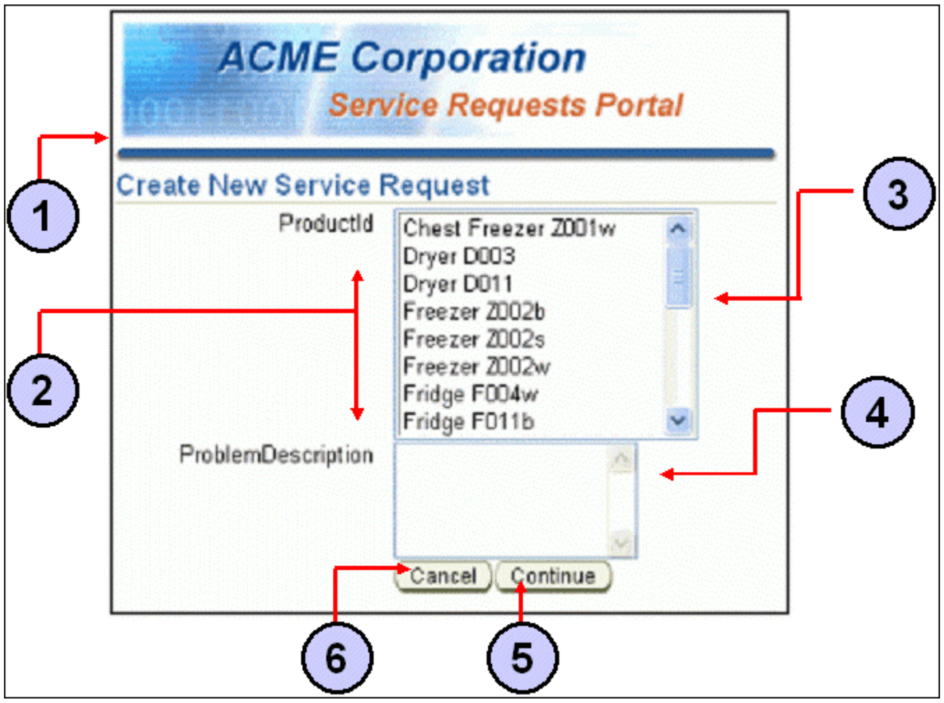
Developing the Create Page

The SRCreate page enables users to create a new service request. The main menu on the SRList page can call this page. It is also possible to call Create New Service Request on the global menu, which is available on all pages.

SRCreate enables users to select from a list of all appliance products and then enter a description. After entering the description, users can click the Continue button to access a confirmation page (see details on SRCreateConfirm).

Another important aspect of the SRCreate page is that users have the option of canceling out of creating a new service request by clicking the Cancel button. Clicking Cancel bypasses all form validation and returns to the SRList page.

The screen shot shows the completed SRCreate page.



SRCreate enables users to select from a list of all appliance products and then enter a problem description. They can then click the Continue button to access a confirmation page, or cancel and navigate to the main page.

To support this functionality and the appearance of the page, you define:

1. A template layout (created in a previous lesson) that you apply to this and all other pages
2. A view object with transient attributes that you use to store global variables that are used as the data source for the input items (the confirmation page calls a method that creates a service request record from these global variables)
3. A view object to serve as the data source for a select list component
4. A method to clear global variables, which is executed automatically when the page is first invoked
5. A command button that navigates to the confirmation page
6. A command button that navigates back to the home page (which you previously defined to be the list page in a global navigation rule) and also calls the method that clears the global variables

Developing the Basic User Interface

In the next part of this chapter, you create the SRCreate page and add the basic UI components. These are the layout components that you will use to hold the data-aware components that you add later.

Perform the following steps to create the SRCreate page and copy the template you created earlier into this page:

1. Open the page-flow diagram.
2. Double-click the **/app/SRCreate.jspx** page to invoke the JSF Page Wizard, and skip the Welcome page.
3. Ensure that the values for the first three steps of the wizard match those in the following tables:

Wizard Step 1: JSP File

Field	Value
File Name	`SRCreate.jspx`
Directory Name	This is the location where the file is stored. Ensure that you create the page in the `\SRTutorialADFBC\UserInterface\public_html\app` folder.
Type	JSP Document
Mobile	Unchecked.

5. Click **Next** to continue.

Wizard Step 2: Component Binding

Field	Value
Do Not Automatically Expose UI Components in a New Managed Bean	Ensure that this option is selected.

6. Click **Next** to continue.

Wizard Step 3: Tag Libraries

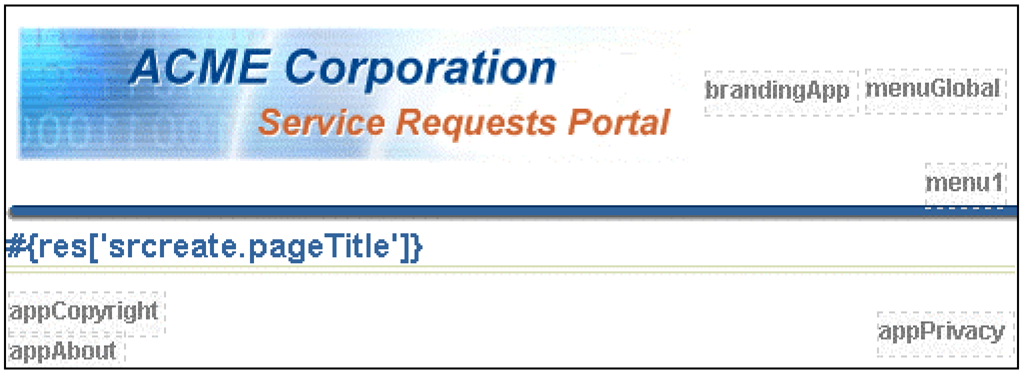
Field	Value

Selected Libraries	`ADF Faces Components`
	`ADF Faces HTML`
	`JSF Core`
	`JSF HTML`

7. Click **OK** to create the page details. The new SRCreate page is displayed in the Visual Editor.
8. Open **SRDemoTemplate.jspx** in the Visual Editor (if it is not already open). In the Structure window, right-click the **`afh:html`** node. From the shortcut menu, select **Copy**.
9. Click the tab to return to the SRCreate page, and in the Structure window select **`f:view`** node.
10. Delete the **`html`** node. Then right-click **`f:view`** and select **Paste** from the shortcut menu. The look and feel that you created earlier is now applied to the new page.
11. Add a title to your page as follows: Click the page in the Visual Editor to select it. (Alternatively, you can select **`af:panelPage`** in the Structure window.) In the Structure window, select **Properties** from the context menu.
12. In the PanelPage Properties dialog box, click **Bind** in the Title property.
13. In the Bind to Data dialog box, expand the **JSP Objects** node and then the **`res`** node in the Variables tree.
14. Scroll through to locate **srcreate.pageTitle**. Select it and shuttle it into the Expression pane. Click **OK**, and then click **OK** again.

This is the name of the page title resource as defined in the `UIResources.properties` file that you created earlier. The value displayed at run time is the actual title of the page.

Your page should now look like the following:

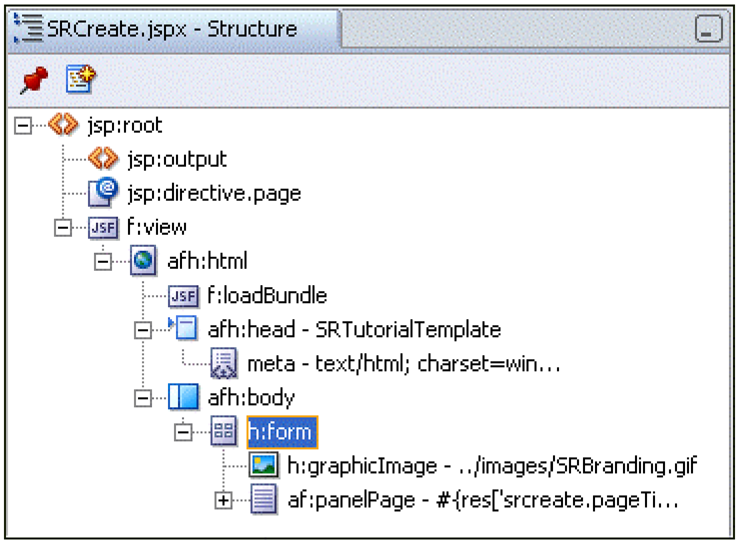


Creating Data Components

Now that you have created the basic page, you can begin to add data-aware components that display the service request data.

Adding Product Name and Problem Description Data

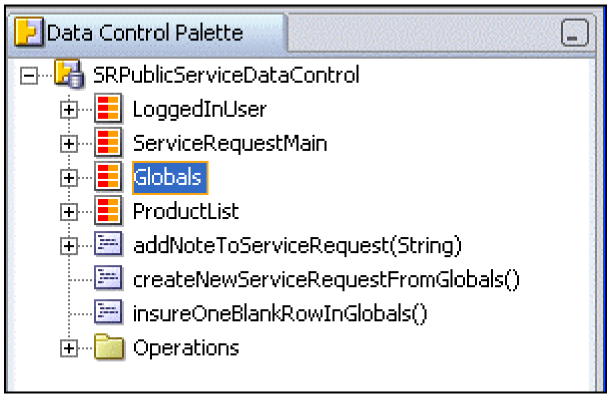
1. In the Structure pane, expand the **`f:view > afh:htmp > afh:body > h:form`** nodes.



SRCreate.jspx - Structure

```
jsp:root
  jsp:output
  jsp:directive.page
  f:view
    afh:html
      f:loadBundle
      afh:head - SRTutorialTemplate
        meta - text/html; charset=win...
      afh:body
        h:form
          h:graphicImage - ../images/SRBranding.gif
          af:panelPage - #{res['srcreate.pageTi...
```

2. In the Data Control palette, expand the **SRPublicServiceDataControl** node and select the **Globals** node. Notice that the two custom methods createNewServiceRequestFromGlobals and insureOneBlankRowInGlobals have been added as available methods for binding.



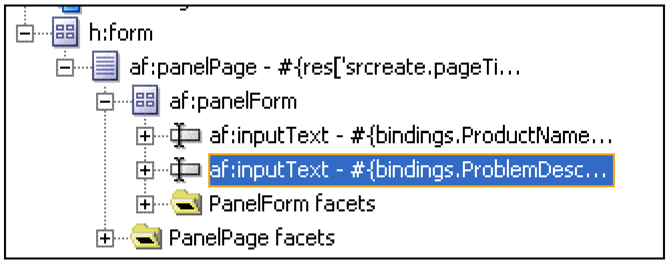
Data Control Palette

```
SRPublicServiceDataControl
  LoggedInUser
  ServiceRequestMain
  Globals
  ProductList
  addNoteToServiceRequest(String)
  createNewServiceRequestFromGlobals()
  insureOneBlankRowInGlobals()
  Operations
```

3. Drag the **Globals** view object and drop it on the **`af:panelPage`** node in the Structure pane.
4. In the context menu, select **`Forms > ADF Form`**.
5. In the Edit Form Fields pane, keep **ProductName** and **ProblemDescription**. Delete **ProductId**.
6. Click **OK**.

In the next steps, refine the display properties of the attribute on the page.

7. In the Structure pane, expand the new **af:panelForm**, and select the **af:inputText** node for the **ProblemDescription** attribute.


h:form
af:panelPage - #{res['srcreate.pageTi...
af:panelForm
af:inputText - #{bindings.ProductName...
af:inputText - #{bindings.ProblemDesc...
PanelForm facets
PanelPage facets

8. In the Property Inspector, set the Rows property to **4**.

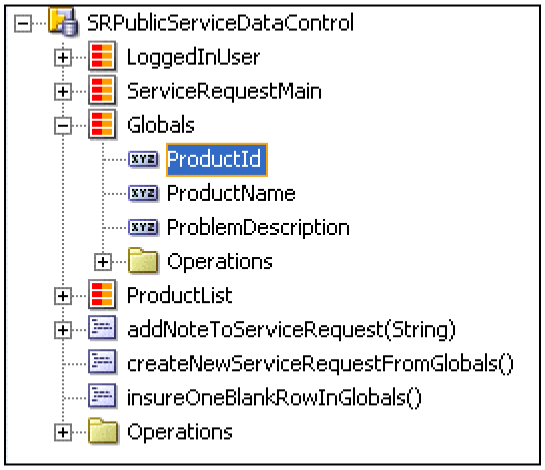
9. **Save** your work.

Adding Product ID Data

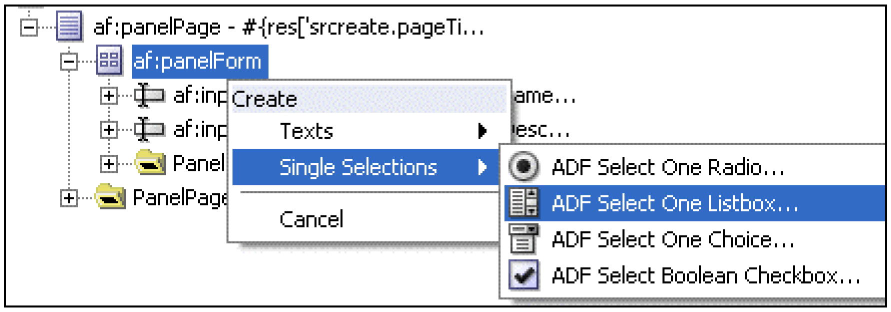
Users want to select the product for the service request from a list of product names. Add the product name as a list item, allowing users to only select one product name.

1. Back in the Data Control palette, expand the **Globals** node, exposing the attributes.

2. Select **ProductId** and drop it on the **af:panelForm**.

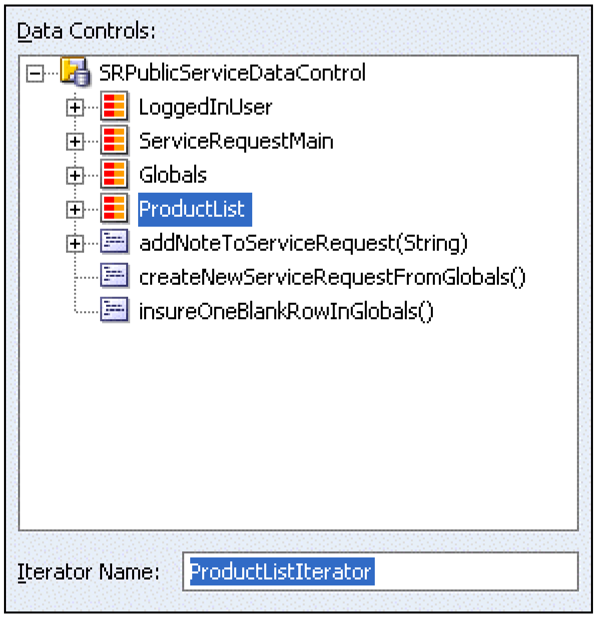

SRPublicServiceDataControl
LoggedInUser
ServiceRequestMain
Globals
ProductId
ProductName
ProblemDescription
Operations
ProductList
addNoteToServiceRequest(String)
createNewServiceRequestFromGlobals()
insureOneBlankRowInGlobals()
Operations

3. In the context menu, select **Single Selections > ADF Select One Listbox**.


af:panelPage - #{res['srcreate.pageTi...
af:panelForm
Create
Texts
Single Selections
Cancel
ADF Select One Radio...
ADF Select One Listbox...
ADF Select One Choice...
ADF Select Boolean Checkbox...

In the next steps you refine the data source for the ProductId as it needs to display all the products currently in the PRODUCTS table.

4. In the List Binding Editor, click the **Add** button, next to the List Data Source item. This action creates a new Iterator.
5. Select the **ProductList** view object and keep the default Iterator Name of `ProductListIterator`.

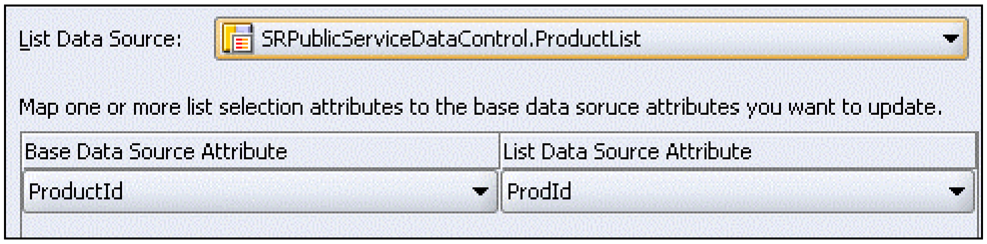


Data Controls:

- SRPublicServiceDataControl
  - LoggedInUser
  - ServiceRequestMain
  - Globals
  - ProductList
  - addNoteToServiceRequest(String)
  - createNewServiceRequestFromGlobals()
  - insureOneBlankRowInGlobals()

Iterator Name: ProductListIterator

6. Click **OK**.
7. Make sure that the Base Data Source Attribute is **ProductId** and the List Data Source Attribute is **ProdId** (that should be the default).



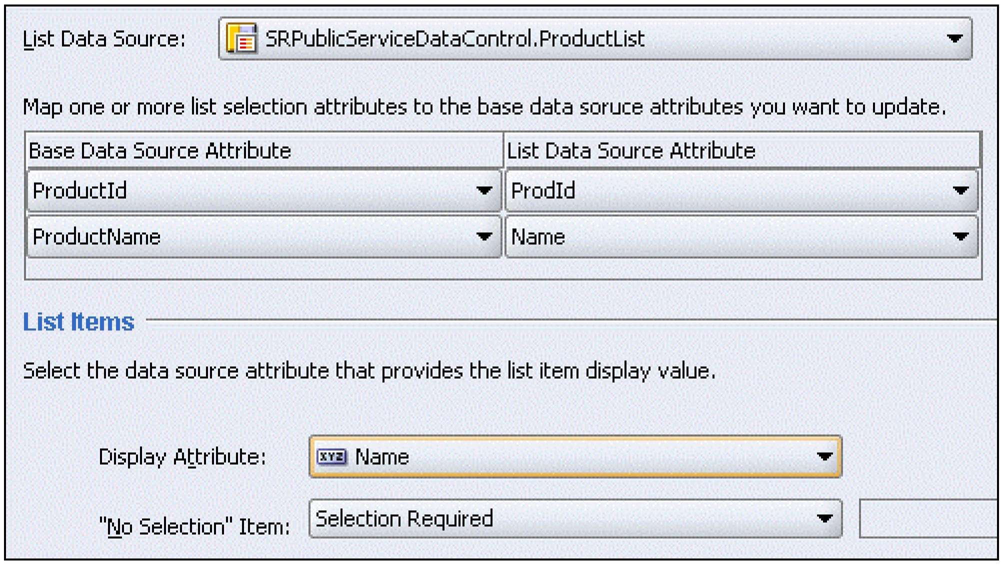
List Data Source: SRPublicServiceDataControl.ProductList

Map one or more list selection attributes to the base data soruce attributes you want to update.

Base Data Source Attribute	List Data Source Attribute
ProductId	ProdId

8. Click the **Add** button next to the table to add a second pair of attribute; then, using the drop-down list, set the Base Data Source Attribute to **ProductName** and the List Data Source Attribute to **Name**.

9. In the List Items area, set the Display Attribute to **Name**.



List Data Source: SRPublicServiceDataControl.ProductList

Map one or more list selection attributes to the base data soruce attributes you want to update.

Base Data Source Attribute	List Data Source Attribute
ProductId	ProdId
ProductName	Name

Select the data source attribute that provides the list item display value.

Display Attribute: Name

"No Selection" Item: Selection Required

10. Click **OK**.
11. In the Structure pane move up **`af:selectOneListBox`** to beneath **`af:panelForm`**.
12. **Save** your work.

This process created an iterator for all the products and displays the product name on the page. All the data components are now included on the page. The application still needs to allow user to cancel or continue with the step in the process.

Adding Button Actions

When a user enters the page, they may fill out the two items or not. If they complete the items, they may want to continue and create a new service request record. They could also decide to cancel the process and return back to the SRList page.

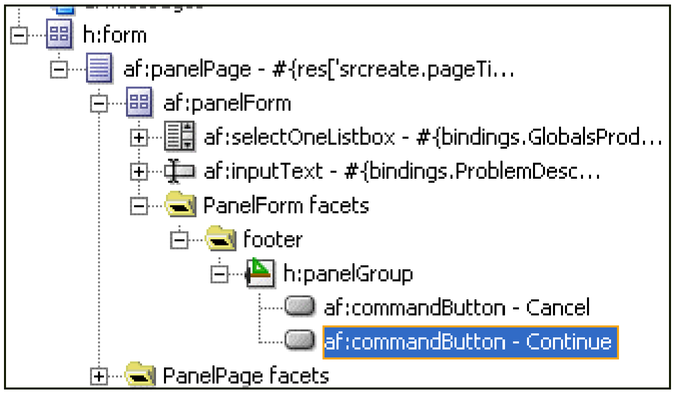
In the next steps, add Cancel and Continue buttons to the page.

1. In the Structure Pane, expand the **`af:panelForm > PanelForm facets`** node, exposing the **`footer`** facet.
2. Open the Data Control palette and select the **`insureOneBlankRowInGlobals()`** node.
3. Drag the **`insureOneBlankRowInGlobals()`** node and drop it onto the footer facet.
4. In the context menu, select **ADF Command Button**.
5. Change the Text property for the new button to **Cancel**.
6. Set the Action property to **`Home`**. And the Immediate property to **true**.
7. **Save** your work.

Pressing this button fires the code to insure the Globals view object is cleaned out in preparation for the next service request creation. It also navigates you to the SRList page. In the next steps, create the continue button.

8. In the Component Palette, select the Command Button item in the ADF Faces Core.

9. Drag and drop it onto the visual editor next to the Cancel button you just created.
10. Set the Text property to **Continue** and the Action property to **confirm**.
11. When complete, the Structure pane should look like the following image.



```
h:form
  af:panelPage - #{res['srcreate.pageTi...
    af:panelForm
      af:selectOneListbox - #{bindings.GlobalsProd...
      af:inputText - #{bindings.ProblemDesc...
      PanelForm facets
        footer
          h:panelGroup
            af:commandButton - Cancel
            af:commandButton - Continue
    PanelPage facets
```

12. **Save** your work.

Refine the Page Components

There are a few more modifications you need to do to complete the page. Complete the next steps to finish the page.

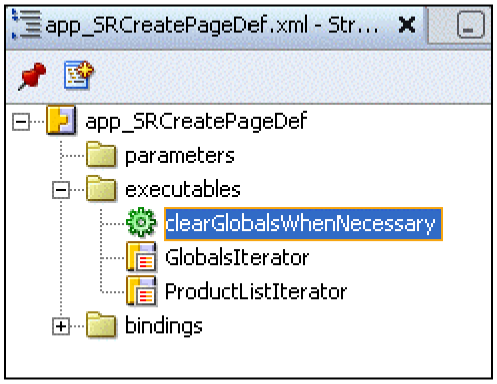
1. In the Structure pane, select the **`af:selectOneListbox`** item, and, using the Property pane, set the Required property to **`true`**.
2. Select the **`af:inputText`** item for the **Problem Description**, and also set the Required property to **`true`**.
3. Select the **`af:inputText`** item for the **Product Name**, and delete it. Keep the **`af:selectOneListbox`**, as it will serve as the mechanism for users to use to select the product for the service request.

In the next steps, modify the page definition file and bind the insureOneBlankRowInGlobals to a refresh condition.

4. Right-click the **`af:panelForm`** and select **Go to Page Definition**.
5. Right-click the **Executables** node, and select **`Insert inside Executables > invokeAction`**.
6. Set the ID property to **clearGlobalsWhenNecessary,** and then, using the drop-down list, set the Binds property to **insureOneBlankRowInGlobals**.
7. Click the **Advanced Properties** tab.
8. Set the Refresh property **prepareModel** to and the RefreshCondition property to **`#{adfFacesContext.postback == false and empty requestScope.performingBack}`** this Boolean expression insures that the clearGlobalsWhenNecessary invokeAction will only fire when we are navigating into the page for the first time, and not when we perform a back navigation from the SRConfirm page. Later you will have the SRConfirm page set the requestScope.performingBack attribute as a signal to the SRCreate page that it should avoid clearing out the Globals

view object, because the users perform the back operation to see and modify their service request details when they noticed something was incorrect on the SRConfirm page.

9. Click **OK** to create the invokeAction binding.
10. Move the `clearGlobalsWhenNecessary` action above the two iterators. The Structure pane should look like the following image.



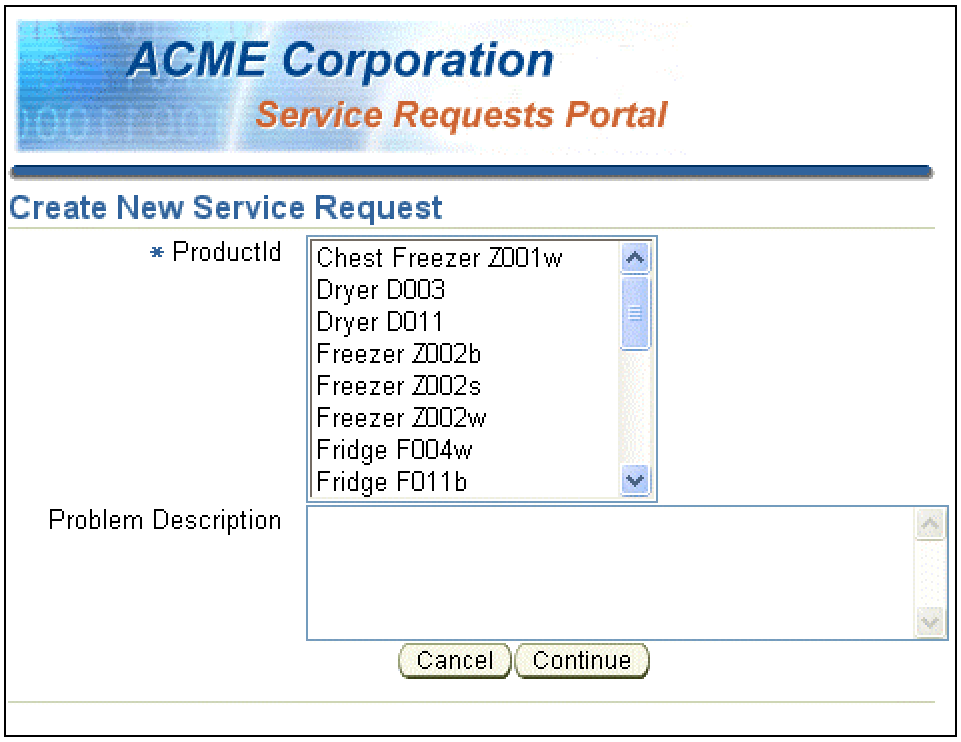
11. **Save** your work.

Running the Page

In the SRList page you created a menu item to invoke the SRCreate page. Run the SRList page and test the link. At this point, the SRCreate page supports selecting a product name and entering a problem description—nothing more. Test the page and functionality.

1. Open the faces-config.xml file, right-click the **SRList** page, and select **Run**.
2. When the page is invoked, select the last option in the menu, **Create New Service Request**.

3. When invoked, the SRCreate page should look like the following image.



Create New Service Request

* ProductId	Chest Freezer Z001w Dryer D003 Dryer D011 Freezer Z002b Freezer Z002s Freezer Z002w Fridge F004w Fridge F011b
Problem Description	

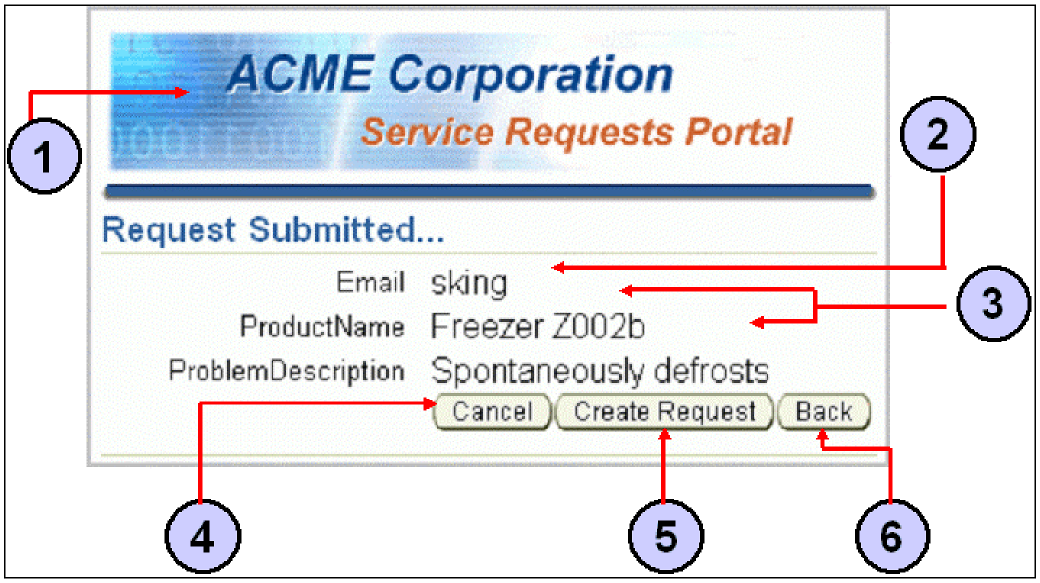
4. Select a product and enter a problem description.
5. Click **Cancel** to be taken back to the **SRList** page.

In the next section, create the SRCreateConfirm page.

Creating the Confirmation Page

The SRCreateConfirm page enables a user to confirm a newly created service request and commit it to the database. It is called when the user clicks the Continue button on the SRCreate page. It displays the new service request information and has three buttons: Cancel, Go Back, and Submit Request. Clicking Cancel cancels the entire new service request entry and navigates to the SRList page. Clicking Back returns the user to the SRCreate page but preserves the new tentative service request entry.

Note: In some cases, labels of fields will be pulled from the general resource bundle rather than being inherited from the bindings for that field. All of the labels which contain "#{res[...]}" reference the resource bundle.



Request Submitted...

Email sking
ProductName Freezer Z002b
ProblemDescription Spontaneously defrosts

The SRCreateConfirm page displays the values that the user entered on the SRCreate page by referring to the same Globals view object where the values are stored. If the values are correct, the user can click a button to create the service request. If the user needs to change the product name or problem description, then the Back button on the page returns to the create page for further editing.

To support this functionality and the appearance of the page, you define:

1. A template layout that you apply to this and all other pages
2. A view object (created in a previous lesson) that contains current user information
3. A view object with transient attributes that stores the global variables that are used as the data source for the displayed values
4. A command button that navigates to the home page without creating the service request
5. A command button that calls a method to create the service request from the values in the global variables and then navigates to the home page
6. A command button that navigates back to the create page for editing

Developing the Basic UI

Here, you continue to develop pages to support the creation process. The second page in the process is the SRCreateConfirm page, which also uses the template you created earlier.

1. With the **`faces-config.xml`** file open, double-click the **SRCreateConfirm** page to invoke the JSF Page Wizard.

Wizard Step 2: Component Binding

Field	Value
Do Not Automatically Expose UI Components in a New Managed Bean	Ensure that this option is selected.

2. The default values for the wizard should be correct. Ensure that the values for the second and third steps of the wizard match those in the following tables
3. Click **Next** to continue.

Wizard Step 3: Tag Libraries	Value

Selected Libraries	`ADF Faces Components`
	`ADF Faces HTML`
	`JSF Core`
	`JSF HTML`

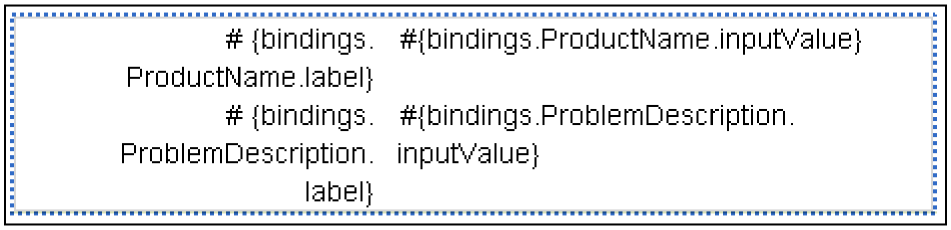
4. Click **Finish** to create the page details. The new **SRCreateConfirm** page is displayed in the Visual Editor.
5. Open the **SRTutorialTemplate** file if it is not already open. In the Structure window, right-click the **`afh:html`** node, and, from the shortcut menu, select **Copy**
6. Click the tab to return to the **SRCreateConfirm** page, and, in the Structure window, expand the **`f:view`** node.
7. Delete the **`html`** node. Then right-click **`f:view`** and select **Paste** from the shortcut menu.
8. In the Structure window, double-click the **`afh:head`** node and change the **Title** property to **SRTutorial Confirm**. Click **OK**.
9. In the Structure window, double-click the **`af:panelPage`** and change the Title property **`#{res['srcreate.confirmPanel.title']}`**. Click **OK.**
10. The Visual Editor now displays the **SRCreateConfirm** page with the look and feel of the other pages.

Adding Data Components

The SRCreateConfirm page displays the product name and problem description values that the user entered on the SRCreate page by referring to the same Globals view object where the SRCreate page saved that information. The values are displayed for users to review. If the users are happy with the values, then they can click a button and the service request is created. If the users needs to change the product name or problem description, then they can click a Back button on the page to go back to the Create page and make their changes.

In the next section add the data components for the product name, problem description and the user who is creating the user request.

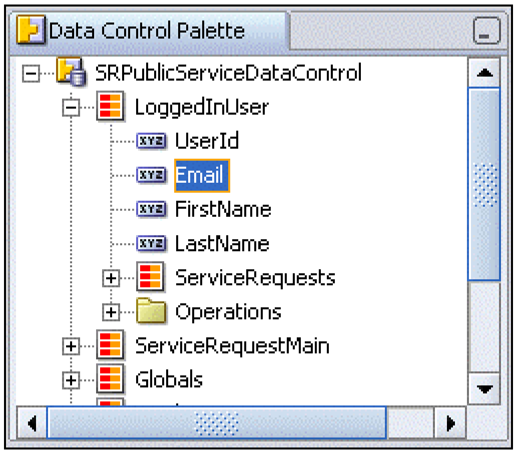
1. In the Data Control palette, expand the **SRPublicServiceDataControl** node and select the **Globals** view object.
2. Drag it into the Structure pane, and drop it on `af:panelPage`.
3. In the context menu, select `Forms > ADF Read-only Form`.
4. In the Edit Form Fields dialog, delete the **ProductId**.
5. Click **OK** and the new form should look like the image below.



```
        # {bindings.   #{bindings.ProductName.inputValue}
    ProductName.label}
        # {bindings.   #{bindings.ProblemDescription.
ProblemDescription.    inputValue}
             label}
```

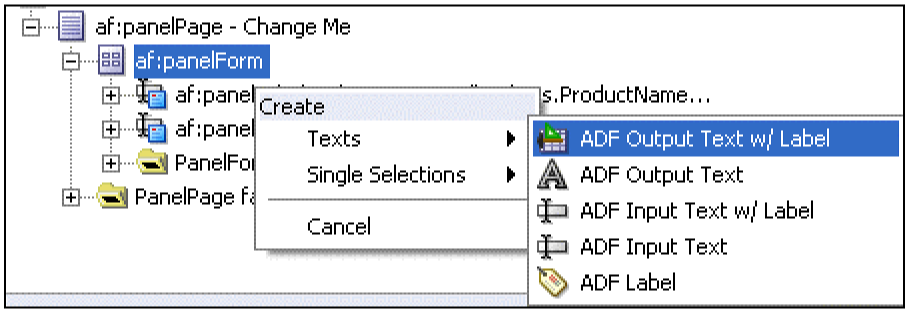
Add the e-mail of the logged-in user to the form.

6. In the Structure pane, expand the `af:panelForm` node
7. In the Data Control palette, expand the `SRPublicServiceDataControl > LoggedInUser` nodes and select the **Email** attribute.



8. Drag the **Email** attribute into the Structure pane, and drop it between the `af:panelForm` and the first `af:panelLabelAndMessage`.

9. In the context menu select **Texts > ADF Output Text w/Label**.



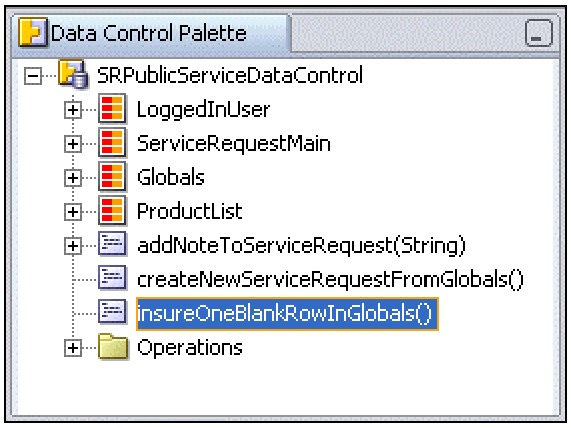
Controlling the Page Action

In the next steps create three buttons to control the page actions.

- A Cancel button to take the user back to the SRList page
- A Create Request button to submit the product name and problem description as a new service request
- A Back button to make changes to the product name or problem description.

First, add the Cancel button

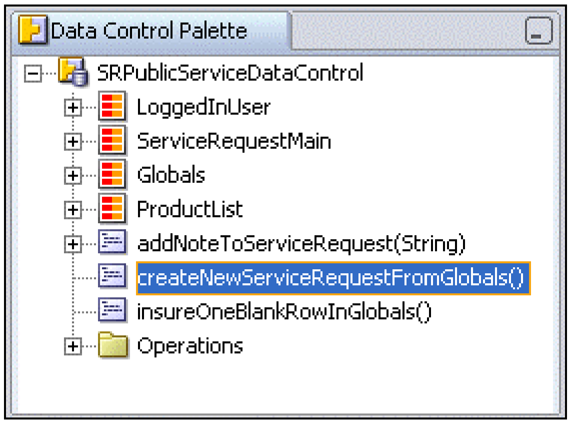
1. In the Structure pane, expand the `af:panelForm > PanelForm facets` nodes, exposing the **footer** node.
2. In the Data Control Palette, select the **insureOneBlackRowInGlobals()** method.


Data Control Palette
SRPublicServiceDataControl
LoggedInUser
ServiceRequestMain
Globals
ProductList
addNoteToServiceRequest(String)
createNewServiceRequestFromGlobals()
insureOneBlankRowInGlobals()
Operations

3. Drag the method into the Structure pane, and drop it on the **footer** node.
4. In the context menu, select **ADF CommandButton**.
5. Set the Text property to **Cancel**, the Action property to **Home**, and the Immediate property to **true**.

Now, add the Create Request button.

6. In the Data Control Palette, select the `createNewServiceRequestFromGlobals()` method.


Data Control Palette
SRPublicServiceDataControl
LoggedInUser
ServiceRequestMain
Globals
ProductList
addNoteToServiceRequest(String)
createNewServiceRequestFromGlobals()
insureOneBlankRowInGlobals()
Operations

7. Drag the method into the Structure pane, and drop it on the **footer** node.

8. In the context menu, select **ADF CommandButton**.

9. Set the Text property to **Create Request** and the Action property to **complete**.

In the final steps, create the Back button.

10. In the Component Palette, select **ADF Faces Core** from the drop-down list.

11. Scroll down and find the **CommandButton** component.

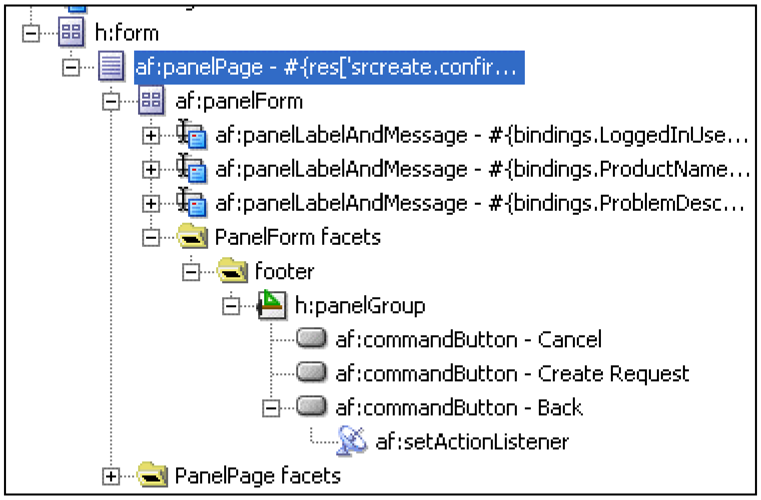
12. Drag the **CommandButton** into the Structure pane, and drop it on the **footer** node.

13. Set the Text property to **Back** and the Action property to **back**.

14. Right-click the **Back** button, and, from the context menu, select **Insert inside af:commandButton > ADF Faces Core > SetActionListener**.

15. Set the From property to **`#{true}`** and the To property to **`#{requestScope.performingBack}.`**

The Structure pane with all the items and buttons should look like the following image:


h:form
af:panelPage - #{res['srcreate.confir...
af:panelForm
af:panelLabelAndMessage - #{bindings.LoggedInUse...
af:panelLabelAndMessage - #{bindings.ProductName...
af:panelLabelAndMessage - #{bindings.ProblemDesc...
PanelForm facets
footer
h:panelGroup
af:commandButton - Cancel
af:commandButton - Create Request
af:commandButton - Back
af:setActionListener
PanelPage facets

16. Use **Save All** to save your work.

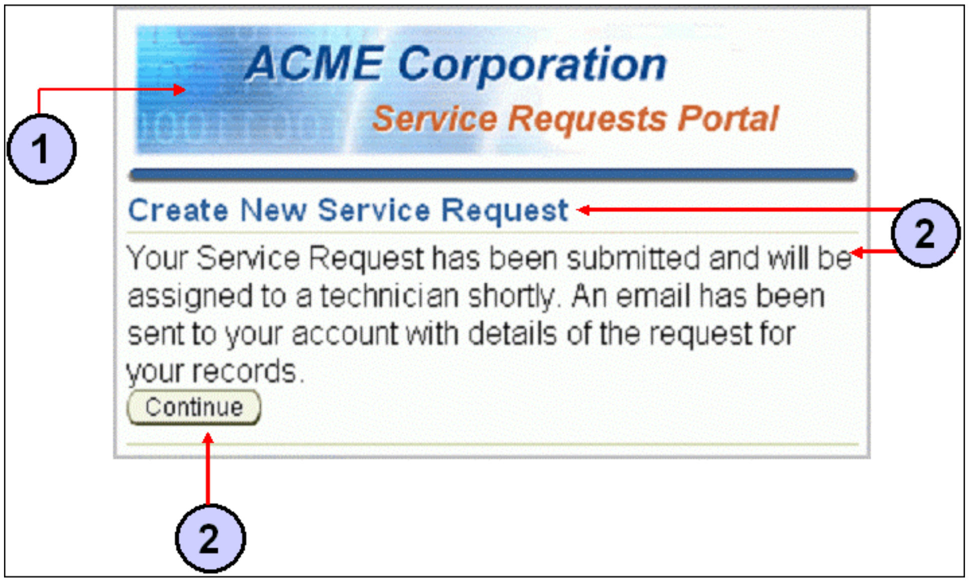
Run and test the pages.

17. Run the SRCreate page

18. Select a product name and enter a problem description, and then click the **Continue** button.
19. In the SRCreateConfirm page, confirm that the values entered on the create page are passed into the confirm page.
20. Test that the **Back** button goes to the SRCreate page and the **Cancel** button goes to the SRList page. Not until the SRDone page is finished will you be able to test the Create Request button. In the next section create the SRDone page.

Creating the Done Page

The SRDone page is the last page in the create process. It informs the user that a service request will be assigned and processed by a technician. This page does not display any data, and is used to complete the process and provide feedback to the user.



Your Service Request has been submitted and will be assigned to a technician shortly. An email has been sent to your account with details of the request for your records.

The SRDone page is the last page in the create process. It informs the user that a service request will be assigned and processed by a technician. This page does not display any data, and is used to complete the process and provide feedback to the user.

To support this functionality and the appearance of the page, you define:

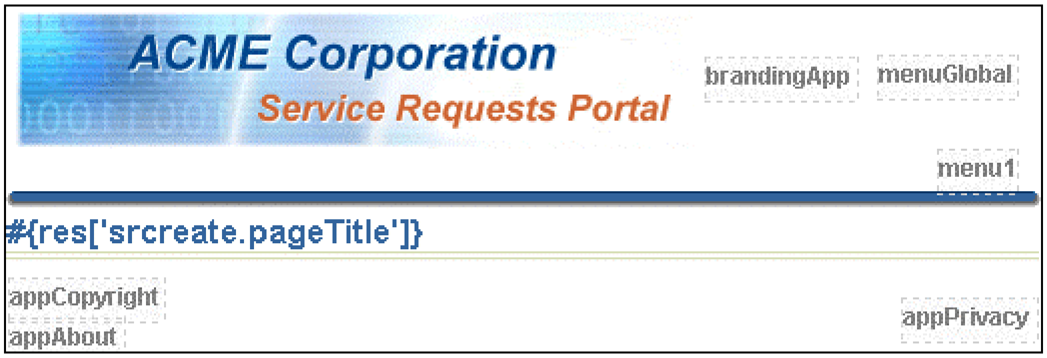
1. A template layout that you apply to this and all other pages
2. A resource file (created in a previous lesson) that is the source of the title and message displayed on the page
3. A command button that navigates to the home page

Developing the Basic UI

The SRCreateDone page also uses the template that you created earlier in the tutorial.

In the faces-config.xml file, double-click the **SRCreateDone** page to invoke the JSF Page Wizard.

1. In the **faces-config.xml** file, double-click the **SRCreateDone** page to invoke the JSF Page Wizard
2. The default properties are acceptable. Click **Finish** to create the page details. The new **SRCreateDone** page is displayed in the Visual Editor.
3. Open the **SRTutorialTemplate** file if it is not already open. In the Structure window, right-click the `afh:html` node, and, from the shortcut menu, select **Copy**
4. Click the tab to return to the SRCreateDone page, and in the Structure window expand the `f:view` node.
5. Delete the `html` node. Then right-click **f:view** and select **Paste** from the shortcut menu.
6. In the Structure window, double-click the `afh:head` node and change the Title property to **SRTutorial Done**.
7. Click **Finish**. The Visual Editor now displays the SRCreateDone page with the look and feel of the other pages.
8. In the Structure window, expand the `f:view> afh.html> afh.body> h:form` nodes to expose the `af:panelPage`.
9. Double-click the `af:panelPage` and change the Text property to `#{res['srcreate.pageTitle']}`. Click **OK**. The Visual Editor displays the new panel title.

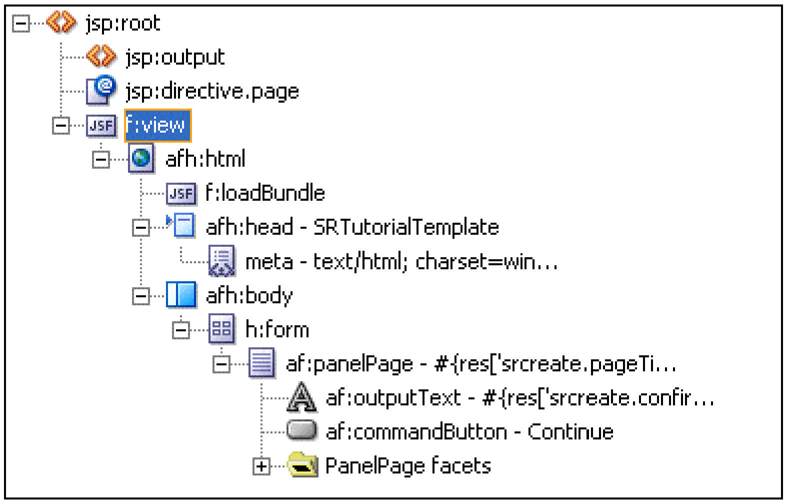


Refining the Page

In the page, add some text informing the users that their service requests have been created and will be assigned to a technician. Also, add a button to navigate back to the SRList page.

1. In the Structure pane, select the `af:panelPage` node.
2. In the Component Palette, select the **ADF Faces Core** group from the drop-down list.
3. Scroll down and click the **Output Text** component. An `af:outputText` component will be nested under the af:panelPage in the Structure pane.

4. Select it, then set the Value property to **`#{res['srcreate.confirmPanel.message']}`** and the Escape property to **`false`**.
   At run time this will return the following: "Your Service Request has been submitted and will be assigned to a technician shortly. An email has been sent to your account with details of the request for your records."
5. Again, in the Structure pane, select the **`af:panelPage`** node.
6. In the Component Palette, scroll and click the **CommandButton** component.
7. A new **`af:commandButton`** is added below the **`af:outputText`** item you just created.
8. In the Property pane, set the Text to **Continue** and the Action to **Home**. The Structure pane should look like the following image.



```
jsp:root
  jsp:output
  jsp:directive.page
  f:view
    afh:html
      f:loadBundle
      afh:head - SRTutorialTemplate
        meta - text/html; charset=win...
      afh:body
        h:form
          af:panelPage - #{res['srcreate.pageTi...
            af:outputText - #{res['srcreate.confir...
            af:commandButton - Continue
            PanelPage facets
```

9. **Save** your work.

Now that the last page of the create process is complete, run and test all three pages. Run the SRCreate page and enter data to create a service request. Continue to the confirmation page, and finally to the done page. Test that the Continue button navigates you to the SRList page.

Summary

In this chapter, you created a master-detail page using ADF Faces components. Those components display data that is coordinated between two panels on the page.

Here are the key tasks that you performed in this chapter:

- Refined the data model
- Created the data view components
- Developed the create page
- Developed the confirmation page
- Developed the done page

8

Developing a Search Page

This chapter describes how to build the Search page using JavaServer Faces and ADF components. The page contains two sections: one to specify the query criteria and the other to display the results. You create buttons enabling the user to select a record and to view or edit the record.

The chapter contains the following sections:

- Introduction
- Creating Data View Components
- Creating the Search Page
- Creating the Query Form
- Creating the Return Table
- Including Navigation in the Page
- Summary

Introduction

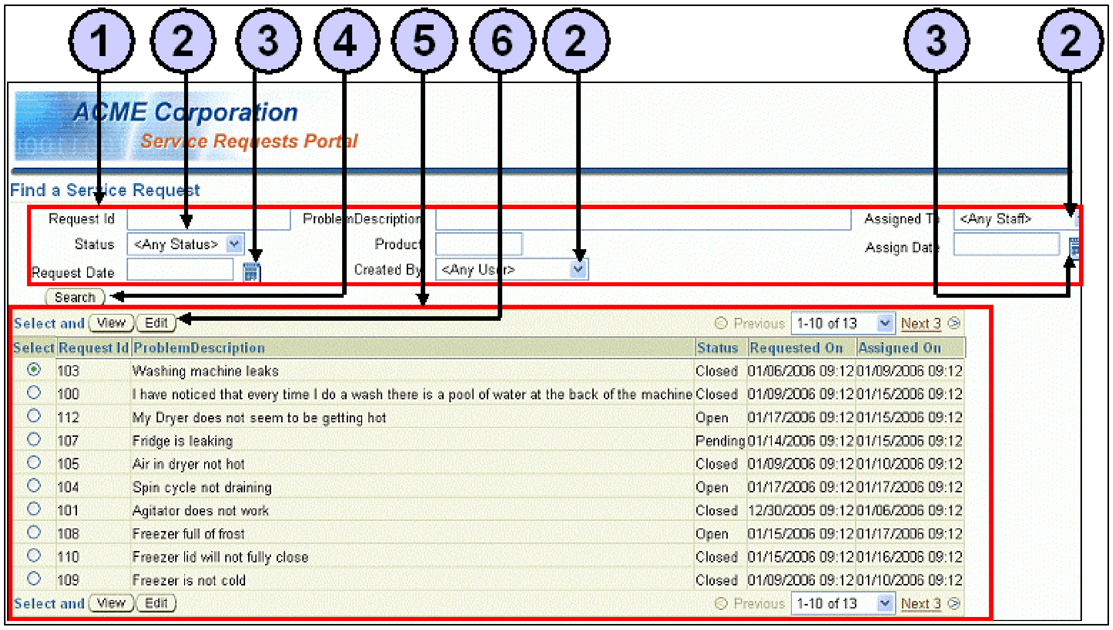
The SRSearch page provides a query-by-example screen for technicians and managers to search the entire list of service requests. The page is divided into two areas: a query area at the top (which is always in query mode) and a results area in the form of a table (which displays the results of the last search).

Note: If you did not successfully complete Chapter 7, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named `Chapter8` to hold the starter application. If you used the default settings, it should be in `<jdev_install>\jdev\mywork\Chapter8`.
2. Unzip `<tutorial_setup>\starterApplications\SRTutorial-EndOfChapter7.zip` into this new directory. Using a new, separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRTutorialADFBC application workspace.
4. Select **File > Open**, and then select `<jdev_install>\jdev\mywork\Chapter7\SRTutorial\SRTutorialADFBC.jws`. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 7.

Right-click the **UserInterface** node in the Navigator and select the **Open JSF Navigation** option from the context menu (alternatively, you can double-click the `faces-config.xml` file in the Navigator) to revisit the page-flow diagram that you created and see how the SRSearch page relates to the other pages. The screen shot shows the completed SRSearch page.



Select	Request Id	ProblemDescription	Status	Requested On	Assigned On
●	103	Washing machine leaks	Closed	01/06/2006 09:12	01/09/2006 09:12
○	100	I have noticed that every time I do a wash there is a pool of water at the back of the machine	Closed	01/09/2006 09:12	01/15/2006 09:12
○	112	My Dryer does not seem to be getting hot	Open	01/17/2006 09:12	01/15/2006 09:12
○	107	Fridge is leaking	Pending	01/14/2006 09:12	01/15/2006 09:12
○	105	Air in dryer not hot	Closed	01/09/2006 09:12	01/10/2006 09:12
○	104	Spin cycle not draining	Open	01/17/2006 09:12	01/17/2006 09:12
○	101	Agitator does not work	Closed	12/30/2005 09:12	01/06/2006 09:12
○	108	Freezer full of frost	Open	01/15/2006 09:12	01/17/2006 09:12
○	110	Freezer lid will not fully close	Closed	01/15/2006 09:12	01/16/2006 09:12
○	109	Freezer is not cold	Closed	01/09/2006 09:12	01/10/2006 09:12

The SRSearch page provides a query-by-example screen for technicians and managers to search the entire list of service requests. The page is divided into two areas: a query area at the top, which is always in query mode, and a results area, which is a table that displays the results of the last search.

Some features of this page include:

1. An area on the page to enter query criteria; created by dragging the service request from the Data Control Palette and creating as an ADF Search Form
2. Components that display a dropdown list for selecting values to query on
3. Components that enable users to enter a date either directly or by selecting from the date picker; supported by new attributes added to the main so that users do not have to enter the time portion of the date
4. A command button to execute the search
5. A table to display the service requests that meet the query criteria; the ADF Read-only Table includes selection and sorting capability
6. Command buttons to view or edit the selected service request

Creating Data View Components

Like with the previous pages, the SRSearch page is based on view objects. Remember, a view object retains its rows even across pages. In this way a returned record from the search can be selected and passed through its key value to the SRMain or SREdit page.

In this section, create two view objects, one to query all users and a second to query only those technician and manager users. These view objects are used to populate the created by and assigned to items on the page.

Creating the User Queries

The SRSearch allows users to query service requests and filter the returning records. The created by and assigned to items support using a select one drop-down list to populate their values. In the next steps, create the two view objects to support populating the drop-down list items with database values.

Create the UserList view object, returning all user names.

1. In the Applications Navigator, right-click the **`oracle.srtutorial.datamodel.queries`** node, and select **New View Object** from the shortcut menu.
2. Click **Next** on the Welcome page.
3. In the first step of the Create View Object Wizard, set the properties using the values from the table below. If the property is not listed below, then leave it at its default value.

Property	**Value**
`Package`	`oracle.srtutorial.datamodel.queries`
`Name`	`UserList`
`Extends`	`<null>`
`Read-only Access`	`Radio button` <Selected>

4. Click **Next**.
5. In Step 2: SQL Statement. Add the following SQL statement to the Query Statement item.

```
SELECT user_id, first_name||' '||last_name as name
FROM users
ORDER BY last_name, first_name
```

6. Click **Next**.
7. In Step 3: Bind Variables, click the **Next** button. There are no variables required for this view object.
8. In Step 4: Attribute mappings, click **Finish**. The view object queries user ID and name data only. The rest of the default values do not need to be changed.

Create the StaffList view object to query only those users who are technicians or managers.

9. In the Applications Navigator, right-click the `oracle.srtutorial.datamodel.queries` node, and select **New View Object** from the shortcut menu.
10. Click **Next** on the Welcome page.
11. In the first step of the Create View Object Wizard, set the properties using the values from the table below. If the property is not listed below, then leave it at its default value.

Property	**Value**
`Package`	`oracle.srtutorial.datamodel.queries`
`Name`	`StaffList`
`Extends`	`<null>`
`Read-only Access`	`Radio button` <Selected>

12. Click **Next**.
13. In Step 2: SQL Statement. Add the following SQL statement to the Query Statement item.

```
SELECT user_id, first_name||' '||last_name as name
FROM users
WHERE user_role IN ('technician','manager')
ORDER BY  last_name, first_name
```

14. Click **Next**.
15. In Step 3: Bind Variables, click the **Next** button. There are no variables required for this view object.
16. In Step 4: Attribute mappings, click **Finish**. The view object queries user ID and name data only. The rest of the default values do not need to be changed.

Refining the Service Request Query

In the SRSearch page you want users to be able to query service requests by assigned and requested date. The date attribute stores time as well as the date. Create two new truncated attributes that do not require specifying a time.

1. In the Applications Navigator, expand the `oracle.srtutorial.datamodel.queries` node, and double-click the **ServiceRequestMain** view object to invoke the property dialog.
2. Select the **Attributes** node.
3. Click the **New** button below the Selected pane.
4. Create a new attribute for the truncated request date. Use the following table to complete the attribute properties.

Property	**Value**
`Name`	`TruncRequestDate`
`Type`	`Date`
`Mapped to Column or SQL`	Checkbox selected
`Expression`	`TRUNC(REQUEST_DATE)`

5. Click **OK** to finish the attribute.
6. Create another attribute for the truncated assigned date, using the values from the following table.

Property	**Value**
`Name`	`TruncAssignedDate`
`Type`	`Date`
`Mapped to Column or SQL`	Checkbox selected
`Expression`	`TRUNC(ASSIGNED_DATE)`

7. Click **OK** to create the attribute. Do not close the ServiceRequestMain view object.

In the next steps, refine the format of the two new attributes.

8. In the ServiceRequestMain view object editor, expand the **Attributes** node in the left pane, and select the **TruncRequestDate** attribute.
9. Click the **Control Hints** tab, and change the following properties for the **TruncRequestDate** attribute to the values in the following table.

Property	**Value**
`Label Text`	`Request Date`

Format Type	`Simple Date`
Format	`dd-MMM-yyyy`

10. In the **Attributes** node, select the **TruncAssignedDate** attribute.
11. Click the **Control Hints** tab, and change the properties for the **TruncAssignedDate** attribute to the values in the following table.

Property	Value
Label Text	`Assigned Date`
Format Type	`Simple Date`
Format	`dd-MMM-yyyy`

12. Click the **OK** button to finish the refinements, and dismiss the editor.
13. Use **Save All** to save your work.

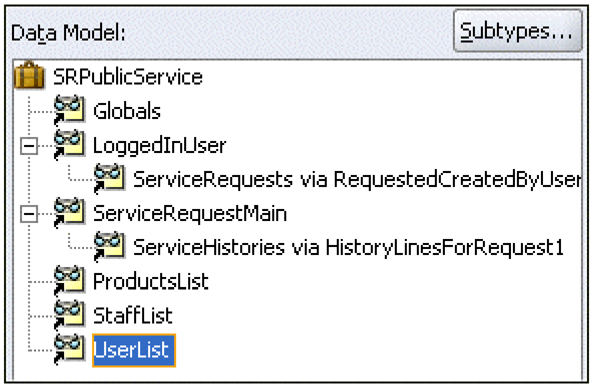
All the data model components have been created and modified to support the SRSearch page. In the next steps, include them in the SRPublicService application module.

Including the Views in the Application Module

To make the new view objects available to ADF Faces pages, they need to be included in the SRPublicService application module.

In the next steps, include the UserList and StaffList view objects into the SRPublicService application module.

1. In the Applications Navigator, open the **`oracle.srtutorial.datamodel`** node and double-click the **SRPublicService** application module.
2. In the Application Module editor, select the **Data Mode**l node.
3. In the Available pane, expand the **`oracle.srtutorial.datamodel.queries`** node.
4. Select the **StaffList** node, change the name from **StaffList 1** to **StaffList** and shuttle it to the Data Model pane.
5. Select the **UserList** node, change the instance name from **UserList1** to **UserList** and shuttle it to the Data Model pane. The Data Model pane should look like the following image.



SRPublicService
- Globals
- LoggedInUser
  - ServiceRequests via RequestedCreatedByUser
- ServiceRequestMain
  - ServiceHistories via HistoryLinesForRequest1
- ProductsList
- StaffList
- UserList

6. Click **OK**.

Creating the Search Page

The first task in creating the Search page is to build its structure using ADF Components and then adding the data component from the data model.

Note: Earlier in the tutorial, you created the page outline in the page-flow diagram. Now you complete the page and apply the template that you created in Chapter 2. You could also create the page from the New Gallery by using the JSF JSP item.

Perform the following steps to create the SRSearch page and attach the template that you created earlier:

1. If it is not open, double-click the `faces-config.xml` file to view the page-flow diagram.
2. Double-click the **SRSearch** page to invoke the JSF Page Wizard.
3. Complete the wizard using the following values:

Wizard Step 1: JSP File

Field	Value
File Name	`SRSearch.jspx`
Directory Name	Make sure the path is set to: `<jdev_install>\jdev\mywork\SRDemo\UserInterface\public_html\app\staff`.
Type	`JSP Document`
Mobile	Clear the check box.

4. Click **Next** to continue.

Wizard Step 2: Component Binding

Field	Value
Do Not Automatically Expo UI Components in a New Managed Bean	Ensure that this option is selected.

5. Click **Next** to continue.

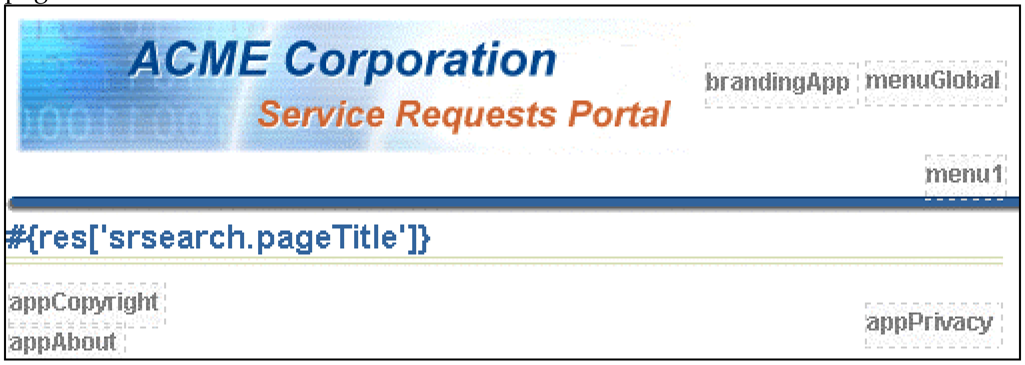
Wizard Step 3: Tag Libraries

Field	Value

Selected Libraries	`ADF Faces Components`
	`ADF Faces HTML`
	`JSF Core`
	`JSF HTML`

6. Click **Finish** to create the page details. The new SRSearch page is displayed in the Visual Editor.
7. Open the **SRTutorialTemplate** file if it is not already open. In the Structure window, right-click the **`afh:html`** node and select **Copy** from the shortcut menu.
8. Click the tab to return to the SRSearch page. In the Structure window, expand the **f:view** node.
9. Delete the **html** node. Then right-click **`f:view`** and select **Paste** from the shortcut menu.
10. Double-click **`afh:head`** and set the Title property to **SRTutorial Search**. The value of this property becomes the browser title. Click **OK**.
11. In the Structure window, expand the **`f:view > afh.html > afh.body > h:form`** nodes to expose `af:panelPage`.
12. Double-click **`af:panelPage`** and, in the properties, set the Title field to **`#{res['srsearch.pageTitle']}`**. This value is set in the template, so you could refer to the res variable when the template is first used or on each page that uses the template. You could set this property to the literal text you want for the title. We are setting it to a value in the `UIResources.properties` file, which contains a list of paired properties and values. At run time, this file is read and the values are replaced on the page. Throughout the tutorial, you use this convention for button names, page titles, and field prompts. At run time, the value should be "Find a Service Request."
13. Click **OK**.

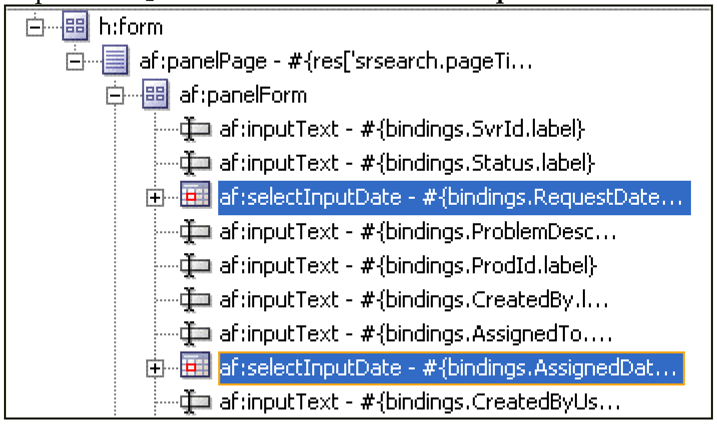
14. The Visual Editor now displays the SRSearch page with the look and feel of the other pages:



Creating the Query Form

In this section, create the main form to accept search criterion to filter returning service requests. Also, refine some of the Search Form items to display as select one drop-down list items.

1. In the Structure pane, expand the **`f:view > afh:html > afh:body > h:form`** nodes.
2. In the Data Control palette, expand the **SRPublicServiceDataControl** node, and select the **ServiceRequestMain** view object node.
3. Drag the ServiceRequestMain view object onto the Structure pane, and drop it on the **`af:panelPage`** node.
4. From the context menu, select **`Forms > ADF Search Form`**.
5. Expand **`af:panelForm`** and delete the **RequestDate** and **AssignedDate** label items.

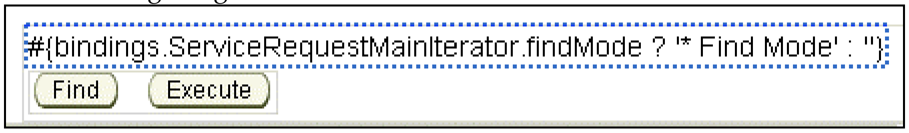


```
h:form
  af:panelPage - #{res['srsearch.pageTi...
    af:panelForm
      af:inputText - #{bindings.SvrId.label}
      af:inputText - #{bindings.Status.label}
      af:selectInputDate - #{bindings.RequestDate...
      af:inputText - #{bindings.ProblemDesc...
      af:inputText - #{bindings.ProdId.label}
      af:inputText - #{bindings.CreatedBy.l...
      af:inputText - #{bindings.AssignedTo....
      af:selectInputDate - #{bindings.AssignedDat...
      af:inputText - #{bindings.CreatedByUs...
```

The page will use the two truncated attributes instead, so the user can search for an assigned date without needing to remember the hours and minutes to find what they are looking for.

6. Select the **`af:PanelForm`** you just created and set the **Rows** property to `3` and the **FieldWidth** property to `100%` (type the `%` character).
7. Expand the **`af:panelForm > Panelform facets > footer > af:panelGroup`** nodes and select the first **`af:outputText`** item. When selected, the design page should look like

the following image.



```
#{bindings.ServiceRequestMainIterator.findMode ? '* Find Mode' : ''}
```

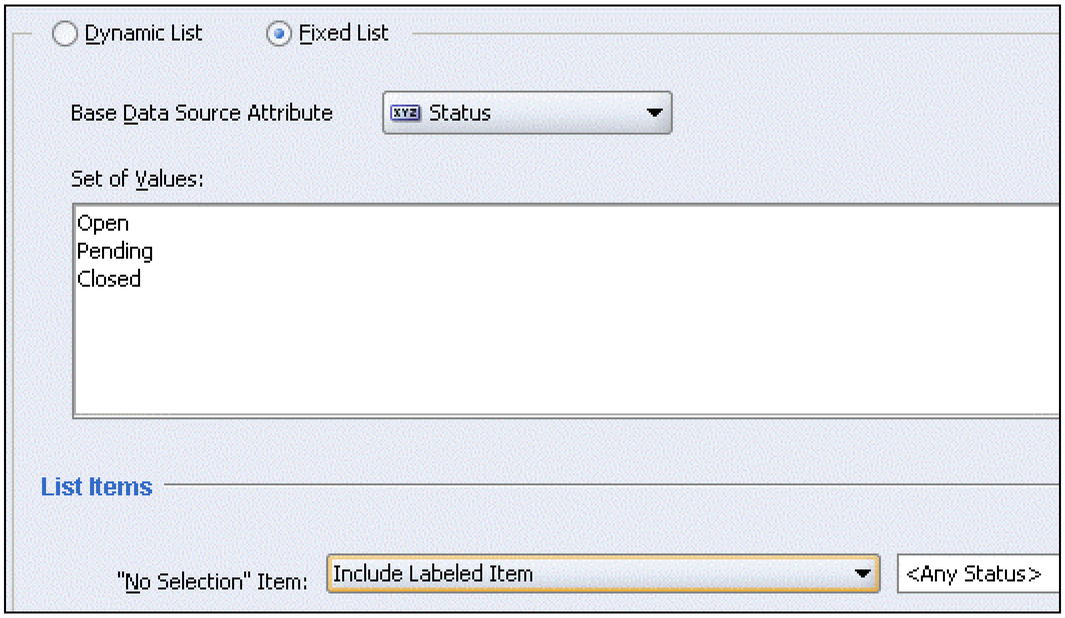
8. Delete this first **`af:outputText`** item, representing the Find Mode. This leaves two buttons: Find and Execute.
9. Underneath the **`af:panelForm`**, delete the following items and labels from the form.
   - `Status`
   - `CreatedBy`
   - `AssignedTo`
   - `CreatedByUserId`
   - `CreatedByFistName`
   - `CreatedByLastName`
   - `AssignedToUserId`
   - `AssignedToFirstName`
   - `AssignedToLastName`

Adding the Status Item

In the next few steps, add a Status item back as a select one choice item.

1. In the Data Component palette, expand the **ServiceRequestsMain** node, and select the **Status** attribute.
2. Drag the **Status** node and drop it on the **`af:panelForm`**.
3. In the context menu, select **`Single Selection > ADF Select One Choice`**.
4. In the List Binding Editor, select the **Fixed List** radio button.
5. In the Base Data Source Attribute, select **Status**.
6. In the Set of Values box, enter the values **`Open`**, **`Pending`**, and **`Closed`**. Make sure each value is on its own line.

7. Set the No Selection Item property to **Include Labeled Item**, and the property to **`<Any Status>`**. The editor should look like the following image.



Base Data Source Attribute: Status

Set of Values:

```
Open
Pending
Closed
```

"No Selection" Item: Include Labeled Item <Any Status>

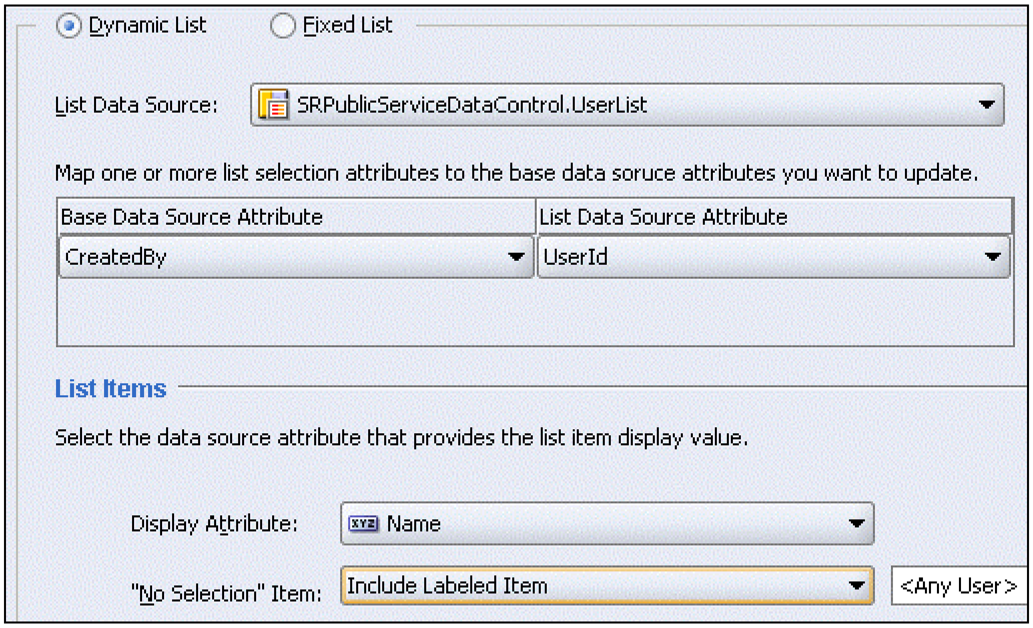
8. Click **OK** to continue.
9. Resequence the **Status** item to be second in the display, after the SvrId.

Adding the CreatedBy Item

In the next few steps, create the CreatedBy user and display it as a select one choice item. The list of users is derived from the UserList view object. In the next few steps add the CreatedBy field and derive its values from the UserList view object.

1. In the Data Component palette, expand the **ServiceRequestsMain** node, and select the **CreatedBy** attribute.
2. Drag the **CreatedBy** node and drop it between the two **`af:selectInputDate`** items.
3. In the context menu, select ***`Single Selection > ADF Select One Choice`***.
4. In the List Binding Editor, make sure that the Base Data Source is set to SRPublicServiceDataControl.ServiceRequestMain, and click the **Add** button next to the **List Data Source** item.
5. In the List Data Source dialog, select the **UserList** view object. The default iterator name is fine and should be **`UserListIterator`**.
6. Click **OK** to return to the List Binding Ecditor.
7. Make sure the Base Data Source Attribute is **CreatedBy** and the List Data Source Attribute is **UserId**.
8. Set the Display Attribute to **Name**.

9. Set the No Selection Item to **Include Labeled Item** and make the text of the no selection item be **`<Any User>`**. The editor should look like the following image



● Dynamic List ○ Fixed List

List Data Source: SRPublicServiceDataControl.UserList

Map one or more list selection attributes to the base data soruce attributes you want to update.

Base Data Source Attribute	List Data Source Attribute
CreatedBy	UserId

List Items

Select the data source attribute that provides the list item display value.

Display Attribute: Name

"No Selection" Item: Include Labeled Item <Any User>

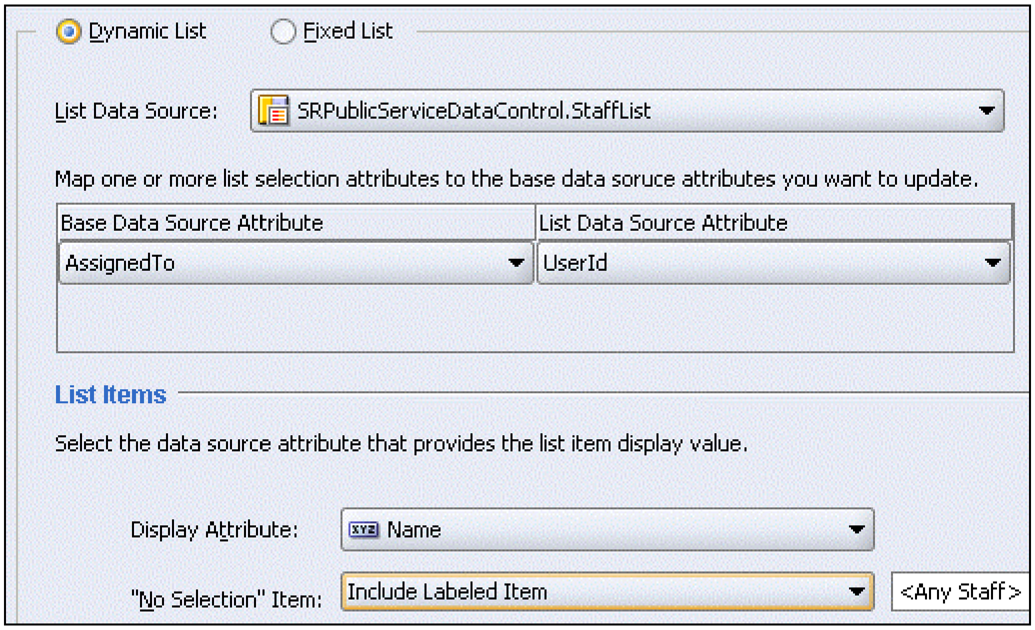
10. Click **OK** to continue.

Adding the AssignedTo Item

In the next few steps, create the AssignedTo user and display it as a select one choice item. The list of users is derived from the UserList view object. In the next few steps add the AssignedTo field and derive its values from the UserList view object.

1. In the Data Component palette, expand the **ServiceRequestsMain** node, and select the **AssignedTo** attribute.
2. Drag the **AssignedTo** node and drop it beneath the **`af:selectOneChoice`** item you just created.
3. In the context menu, select **`Single Selection > ADF Select One Choice`**.
4. In the List Binding Editor, click the **Add** button next to the **List Data Source** item.
5. In the List Data Source dialog, select the **StaffList** view object. The default iterator name is fine and should be **`StaffListIterator`**.
6. Click **OK** to return to the List Binding Editor.
7. Make sure that the Base Data Source Attribute is **AssignedTo** and the List Data Source Attribute is **UserId**.
8. Set the Display Attribute to **Name**.

9. Set the No Selection Item to **Include Labeled Item** and make the text of the no selection item be **`<Any Staff>`**. The editor should look like the following image.



(● Dynamic List) ○ Fixed List

List Data Source: SRPublicServiceDataControl.StaffList

Map one or more list selection attributes to the base data soruce attributes you want to update.

Base Data Source Attribute	List Data Source Attribute
AssignedTo	UserId

List Items

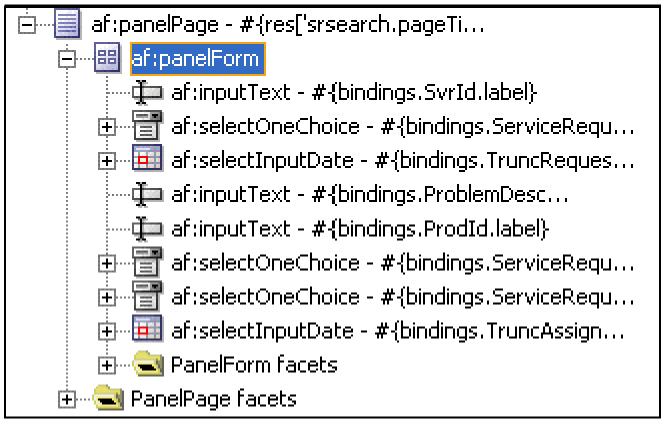
Select the data source attribute that provides the list item display value.

Display Attribute: Name

"No Selection" Item: Include Labeled Item <Any Staff>

10. Click **OK** to continue.

11. Rearrange the items to look like below. The Structure pane should look like this:



```
af:panelPage - #{res['srsearch.pageTi...
  af:panelForm
    af:inputText - #{bindings.SvrId.label}
    af:selectOneChoice - #{bindings.ServiceRequ...
    af:selectInputDate - #{bindings.TruncReques...
    af:inputText - #{bindings.ProblemDesc...
    af:inputText - #{bindings.ProdId.label}
    af:selectOneChoice - #{bindings.ServiceRequ...
    af:selectOneChoice - #{bindings.ServiceRequ...
    af:selectInputDate - #{bindings.TruncAssign...
    PanelForm facets
  PanelPage facets
```

Creating the Return Table

Once the query executes there needs to be a place where the returned rows are displayed. In the next section, add a table component to display the returned rows.

1. In the Structure pane, collapse the **af:panelPage**
2. Drop the **ServiceRequestMain** view object from the Data Control palette as **Table > ADF Read-only Table** inside the af:panelPage.
3. In the Edit Table Columns dialog, select the **Enable Selection** and **Enable Sorting** check boxes.
4. Include only columns for **SvrId**, **ProblemDescription**, **Status**, **RequestDate**, **AssignedDate** in that order.
5. Click **OK**.
6. Select the **Submit** button and change the Text property to **Edit**.
7. Set the Action property to **edit**. Clicking this button will navigate you to the SREdit page, which you will create in the next chapter.

Add a command button to navigate to the SRMain page.

8. In the Structure pane, expand the **`af:table > Table facets > selection`** nodes right click the **`af:tableSelectOne`** one and select **`Insert inside af:tableSelectOne > CommandButton`**.
9. Set the Text property to **View** and the Action property to **view**.
10. Use **Save All** to save your work.

In the search form, you will use two different iterator bindings, both for the ServiceRequestMain view object. You will declaratively force the iterator to which the search fields are bound to always stay in Find Mode, the query-by-example mode that ADF view objects support. In this way, any values entered in these fields are automatically applied as query-by-example criteria against their underlying view object attributes. A second iterator that we will create will be in regular data display mode so that any controls bound to it will display the search results from the ServiceRequestMain view object after having applied the above query-by-example criteria. First, you will force the existing iterator to always stay in Find Mode.

11. Right-click anywhere on the page and select **Go to the Page Definition.**
12. Right-click the **executables** node, and from the context menu, select **`Insert inside executables > invokeAction`**.
13. Set the ID property to **ForceFindModeAlways** and the Binds property to the **Find** action.
14. Click the Advanced tab, and set the RefreshCondition to the expression:

    ```
    #{bindings.ServiceRequestMainIterator.findMode == false}
    ```
15. Move the **ForceFindModeAlways** action to be first in the list of executables. This insures the page is invoked in 'Find' mode.

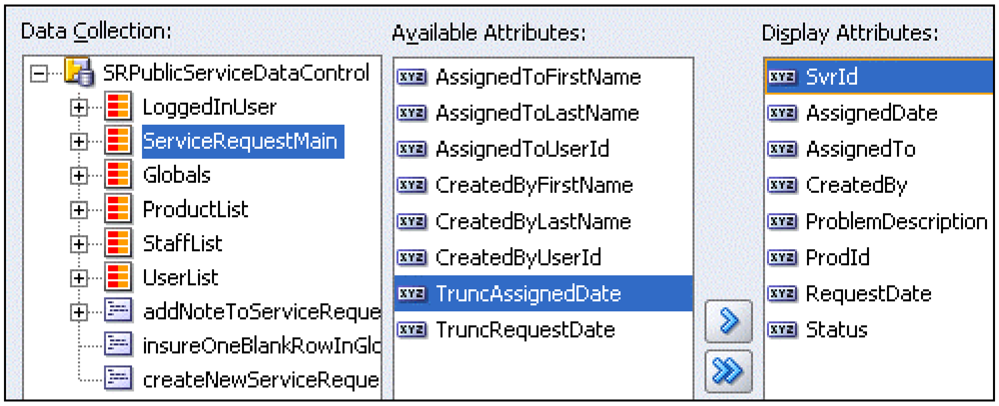
Next, you will create the second iterator binding for the ServiceRequestMain view object that will always be in data display mode (which is the default mode for an iterator).

16. Right-click the executables node, and from the context menu, select **`Insert inside executables > iterator`**.
17. Set the Iterator ID to **ServiceRequestMainResultsIterator**.
18. Expand the **SRPublicServiceDataControl** and select the **ServiceRequestMain** view object
19. Click **OK**.

Adjust the Results Table to Bind to the Results Iterator

When you initially dropped the read-only table for the ServiceRequestMain view object, it was automatically bound to the ServiceRequestMainIterator, the only one that was in the page definition at the time. This iterator is the one you've forced to stay in find mode, so we need to change the iterator to which the results table is bound to use the new ServiceRequestMainResultsIterator you just created above. This will ensure that the table will display the results of the Search Form.

1. Click the **SRSearch.jspx** tab to bring it to the forefront.
2. In the Structure pane, select the **`af:table`** and right-click **Edit Binding**.
3. In the Table Binding Editor, use the Select an Iterator drop-down list at the bottom of the pane to select **ServiceRequestMainResultsIterator**.
4. Shuttle all the attributes shown here, into the **Display Attributes** list.



5. Click **OK**.

Since you are using invokeAction above to force the ServiceRequestMainIterator to always be in find mode, the user will not need the Find button, which by default would toggle the form in and out of find mode.

6. In the Structure pane, expand the **`panelForm > PanelForm facets > footer > af:panelGroup > af:panelButtonBar`** nodes and delete the **Find** button from the `af:panelForm`.
7. Select the **Execute** button and set the Text property to **Search**.

Including Navigation in the Page

The SRSearch page is called from the SRList page. Once a set of records is returned from the search, the page needs to pass the selected service request onto the SRMain page and display the service histories. Pass the service request by adding an action listener to pass the current row.

Then to access the search page, you add a menu item on the SRList page to navigate to the SRSearch page.

In the next steps add an action listener to the View commandButton.

1. In the Structure pane for the SRList page, expand the **`panelPage > af:table > Table facets > selection > af:tableSelectOne`**, then right-click the **View** button and select **`Insert inside af:commandButton > ADF Faces Core > SetActionListener`**.
2. Set the From property to **`#{row.rowKeyStr}`** and the To property to **`#{requestScope.requestRowKey}`**.
3. Click **OK** to create the listener.

In the next steps, create a menu item to invoke the SRSreach page.

4. If not already available, open the **SRList** page in the editor.
5. Expand the **`h:form > af:panelPage > PanelPage facets > menu2`** nodes.
6. Right-click the **`af:menuBar`** and from the context menu, select **`Insert inside af:menuBar > ADF Faces Core`**.
7. In the Insert ADF Faces Core dialog, select **CommandLink**.
8. Click **OK**.
9. Set the Text property for the af:commandLink to **Search**, and the Action property to **search**.
10. Use **Save All** to save your work.

Running and Testing the Page

- Run the SRList page, and test the **Search** menu navigation item. (If you have problem running the Search page you might try to use Run > Clean UserInterface).
- Once in the SRSearch page, test each of the select one choice items, and search records.
- Finally, select a record and use the View button to navigate to the SRMain page to see the service history.

In the next chapter you create the SREdit page, and will come back to the search page to link it in.

Summary

In this chapter, you built the Search page using JavaServer Faces and ADF components. To accomplish this, you performed the following key tasks:

- Created the data view components
- Created the search page

- Created the query form
- Created the return table
- Included navigation in the page

9

Developing an Edit Page

This chapter describes how to create the SREdit page, the page in the SRDemo application that enables managers and technicians to edit service requests.

The chapter contains the following sections:

Introduction

The SREdit page is one of three screens intended to be used by the company's staff rather than by its customers. Managers and technicians use the page to update information on service requests.

Note: If you did not successfully complete Chapter 9, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named **`Chapter9`** to hold the starter application. If you used the default settings, it should be in `<jdev-install>\jdev\mywork\Chapter9`.
2. Unzip **`<tutorial-setup>\starterApplications\SRTutorialADFBC-EndOfChapter8.zip`** into this new directory. Using a new, separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRTutorialADFBC application workspace.
4. Select **File > Open**, and then select **`<jdev-install>\jdev\mywork\Chapter9\SRTutorialADFBC\SRTutorialADFBC.jws`**. This opens the starter application for this chapter.

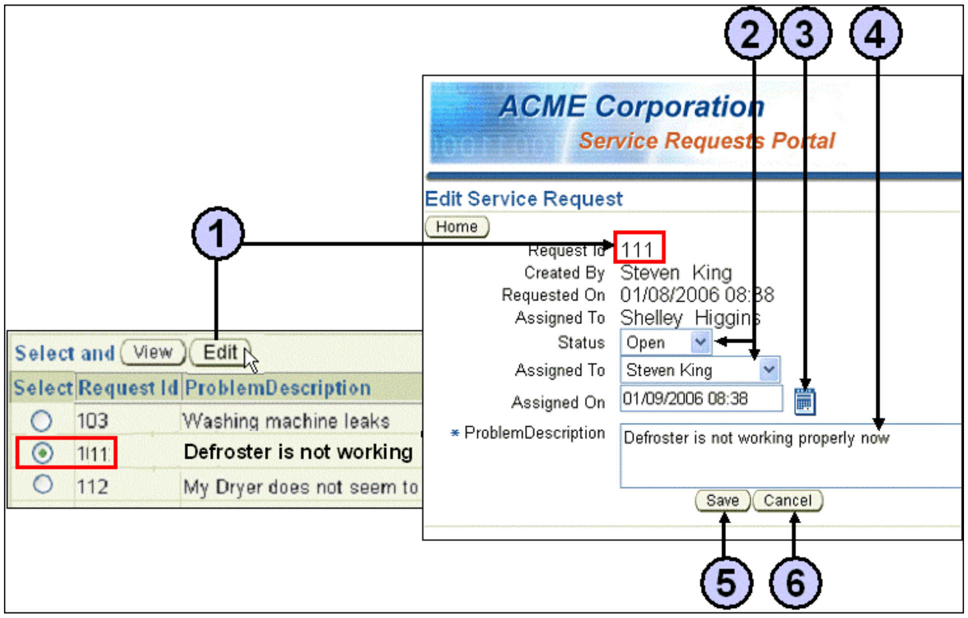
You can now continue with this tutorial using an application that implements all of the steps from Chapter 8.

Double-click the `faces-config.xml` file in the Applications Navigator to revisit the page-flow diagram and examine how the SREdit page relates to the other pages.

Here are some key points to note about the SREdit page:

- The page can be called from three different places within the application: the SRList page, the SRSearch page, and SRMain. In order to reuse the page from these different contexts, it is parameterized; that is, the calling page is expected to set parameters that inform the SREdit page about which service request to query and where to return after the edit is finished.
- The page enables a service request to be edited by a manager or a technician.
- It enables the user to modify the status of a request.
- It provides for the request to be assigned to another technician or manager.
- The Problem Description field and the Assigned Date field are disabled for requests with a status of Closed.

The following screenshot shows you how the finished SREdit page should look (from a manager login):



Select	Request Id	ProblemDescription
○	103	Washing machine leaks
●	111	Defroster is not working
○	112	My Dryer does not seem to

Edit Service Request

Request Id	111
Created By	Steven King
Requested On	01/08/2006 08:38
Assigned To	Shelley Higgins
Status	Open
Assigned To	Steven King
Assigned On	01/09/2006 08:38
* ProblemDescription	Defroster is not working properly now

The SREdit page enables a technician or manager to edit a service request. The page can be called from three different pages in the application. The calling page informs the SREdit page about which service request to query.

Some features of the edit page include:

1. Automatically displays the service request that is selected in the calling page
2. Dropdown lists for valid status and technicians, with the record displaying even if the service request is not assigned
3. A date picker for the assigned date that defaults to the current date; the date entered must be on or after the request date
4. A multi-line text area for the problem description
5. Disabled assigned date and problem description when the status is Closed
6. A Save button to commit the update
7. A Cancel button that calls a method to refresh the data from the database

Creating Data View Components

Like with the previous pages, the SREdit page is based on service request view objects.

Adding a Default Date Value

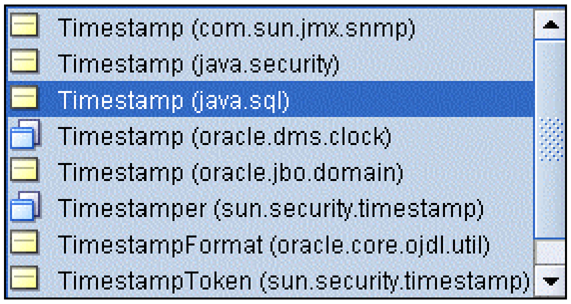
In the ServiceRequestImpl.java class for the ServiceRequest entity object, add one line of code into the setter method for the AssignedTo attribute that sets the AssignedDate to the current date whenever the AssignedTo attribute is set.

1. In the Applications Navigator, expand the `oracle.srtutorial.datamodel.entities` node and right-click the **ServiceRequest** entity object.

2. From the context menu, select **Go to Entity Object Class** item. The ServiceRequestImpl.java file should be displayed in the editor.
3. In the Structure pane, scroll down the list of getter and setter methods and double-click the **setAssignedTo(Number):void** node. The associated location in the code should be highlighted.
4. At the end of the method insert a new line of code to set the AssignedDate to the current date and time. Add the bolded line of code to the method to populate the date.

```
public void setAssignedTo(Number value) {
  setAttributeInternal(ASSIGNEDTO, value);
  setAssignedDate(new Date(new Timestamp(System.currentTimeMillis())));
}
```

5. When prompted, import the java.sql.Timestamp class



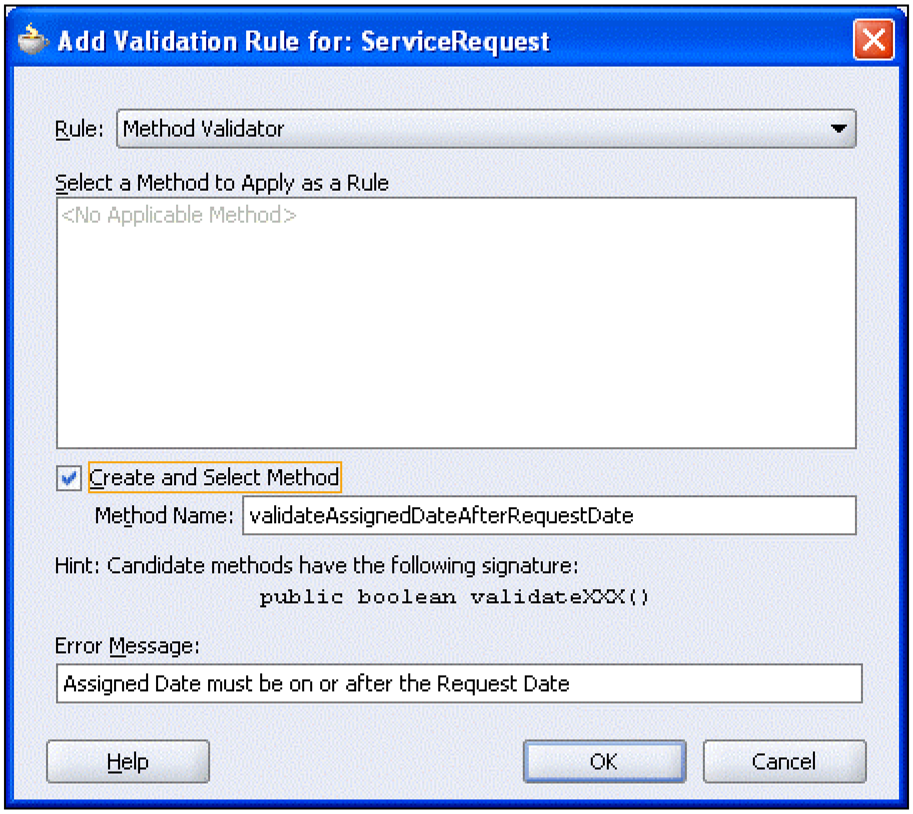
Timestamp (com.sun.jmx.snmp)
Timestamp (java.security)
Timestamp (java.sql)
Timestamp (oracle.dms.clock)
Timestamp (oracle.jbo.domain)
Timestamper (sun.security.timestamp)
TimestampFormat (oracle.core.ojdl.util)
TimestampToken (sun.security.timestamp)

6. Use **Save All** to save your work.

Adding Date Range Validation

Edit the ServiceRequest entity object and add validation to display an error if the AssignDate is earlier than the RequestDate.

1. In the Applications Navigator, expand the **`oracle.srtutorial.datamodel> entities`** node and double-click the **ServiceRequest** entity object.
2. Select the **Validation** node.
3. Define a validation rule at the entity level, by selecting the **ServiceRequest** node and clicking the **New** button.
4. In the Add Validation Rule for: ServiceRequest dialog box set the Rule property to **Method Validator**.
5. Set the Method Name property to **`validateAssignedDateAfterRequestDate`** and the Error Message property to **`Assigned Date must be on or after the Request Date`**.


Add Validation Rule for: ServiceRequest
Rule:
Select a Method to Apply as a Rule
Help
OK
Cancel

Rule: Method Validator

<No Applicable Method>

[✓] Create and Select Method

Method Name: validateAssignedDateAfterRequestDate

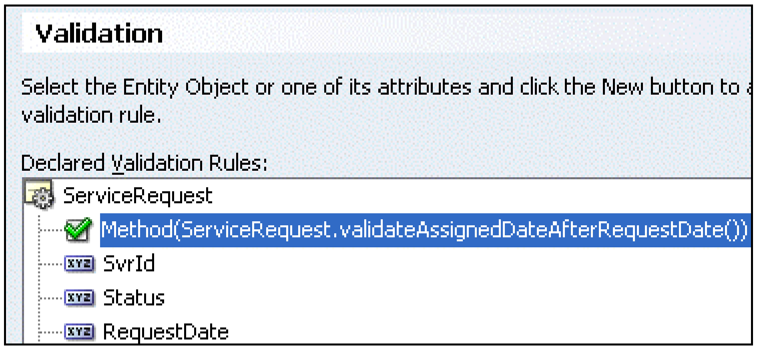
Hint: Candidate methods have the following signature:

```
public boolean validateXXX()
```

Error Message:

Assigned Date must be on or after the Request Date

6. Click **OK** to return to the Validation panel.



Select the Entity Object or one of its attributes and click the New button to a validation rule.

```
ServiceRequest
  Method(ServiceRequest.validateAssignedDateAfterRequestDate())
  SvrId
  Status
  RequestDate
```

7. Click **OK** to exit the editor.

In the next steps, modify the new method and add one line of code that evaluates when the validation will succeed. It will succeed either when the assigned date is null or when the assigned date is greater than or equal to the request date.

8. Open the **ServiceRequestImpl.java** file, if it is not already open.

9. In the Structure pane, scroll down and double-click the **validateAssignedDateAfterRequestDate(): boolean** node. The method is highlighted in the code editor.

10. Modify the return statement and add the lines of code to ensure that the assign date is not before the request date. The code to add follows, in bold.

```
public boolean validateAssignedDateAfterRequestDate() {
        return getAssignedDate() == null
                ||
       getAssignedDate().dateValue().compareTo(getRequestDate().dateValue())
      >= 0;
    }
```

11. Use **Save All** to save your work.

The AssignedDate will never be allowed to be later than the RequestDate, assuming someone is given access to the attribute in any pages.

Adding a Cancel Method

If the user decides to cancel the editing session, then discard the changes. Add the method to the SRPublicService application module.

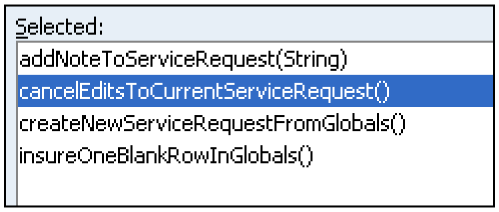
1. In the Applications Navigator, expand the `oracle.srtutorial.datamodel` node and right-click the **SRPublicService** application module.
2. On the context menu, select **Go to Application Module Class**. The SRPublicServiceImpl.java class is loaded into the editor.
3. Scroll down to the bottom of the file and add the new method. The code for the method is listed, following. .

```
public void cancelEditsToCurrentServiceRequest() {
getServiceRequestMain().getCurrentRow().refresh(Row.REFRESH_WITH_DB_FORGE
T_CHANGES);
}
```

4. Use **Save All** to save your work.

In the next steps, add the custom method to be part of the application module's data control.

5. Double-click the **SRPublicService** application module.
6. Select the **Client Interface** node.
7. Notice the **cancelEditsToCurrentServiceRequest** is in the Available list. Select it and move it to the Selected list.


Selected:
addNoteToServiceRequest(String)
cancelEditsToCurrentServiceRequest()
createNewServiceRequestFromGlobals()
insureOneBlankRowInGlobals()

8. Click **OK**.
9. Use **Save All** to save your work.

All of the data model modifications for the SREdit page are complete. In the next section develop the page.

Creating the Page Outline

Perform the following steps to create the SREdit page and add the template to apply the appropriate look and feel.

1. If it is not already open, double-click the **`faces-config.xml`** file to view the **Page Flow Diagram**.
2. Double-click the **SREdit** page to invoke the JSF Page Wizard.
3. Complete the first three steps of the wizard using the values in the following tables:

Wizard Step 1: JSP File

Field	Value
File Name	`SREdit.jspx`
Directory Name	This is the location where the file is stored. Ensure that you create the page in the `\SRDemo\UserInterface\public_html\app\staff` folder.
Type	`JSP Document`
Mobile	Clear the check box.

4. Click **Next**.

Wizard Step 2: Component Binding

Field	Value
Do Not Automatically Expose UI Components in a New Managed Bean	Ensure that this radio button is selected.

5. Click **Finish** to create the page details. The new SREdit page is displayed in the Visual Editor.
6. Open the **SRTutorialTemplate** file if it is not already open. In the Structure window, shrink the **afh:html** node and select it. On the context menu, select **Copy**.
7. Click the tab to return to the SREdit page. In the Structure window, expand the **`f:view`** node.
8. Delete the **`html`** node. Then right-click **`f:view`** and select **Paste** on the context menu.

   The look and feel that you created earlier is now applied to the new page.
9. Add a title to your page as follows: Click in the page in the Visual Editor to select the panelPage. (Alternatively, you can select **`af:panelPage`** in the Structure window.) In the Property Inspector, type **`#{res['sredit.pageTitle']}`** in the Title property.

   Alternatively, as you have seen several times before in earlier chapters of this tutorial, you can invoke the PanelPage Properties dialog box and click **Bind** in the Title property and select **sredit.pageTitle** from the res node, and shuttle it into the Expression pane.

The name of the page title comes from a resource as defined in the `UIResources.properties` file.

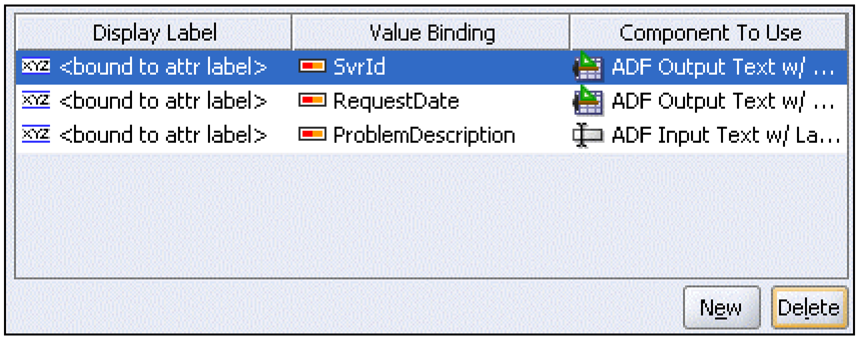
10. Add a header that will appear in the browser title when you run the page, as follows: In the Structure window, select **`afh:head`**. Set the Title property to **`#{res['sredit.pageTitle]}`**.

Adding Data Components to the Page

In the next section add the service request data components to the page. In some cases the default components will work fine, like the service request ID, request date, and problem description. Display other attributes, like created by, assigned to, and status, as a list of values.

In the next few steps, create the default data components for the service request ID, request date, and problem description.

1. In the Data Control palette, expand the SRPublicServiceDataControl and select the **ServiceRequestMain** view object.
2. Drag it into the Structure pane and drop it on the **`af:panelPage`** node.
3. On the context menu, select **`Forms > ADF Form`**.
4. In the Edit Form Fields dialog box, set the Component To Use property to **`ADF OutputText w/Label`** for the **SvrId** and **RequestDate** fields.
5. Leave the Component To Use as **ProblemDescription** as **`ADF InputText w/ Label`**.
6. Delete all of the other attributes in the list except for the three above. (We will be adding individual ones shortly to finish the job.)

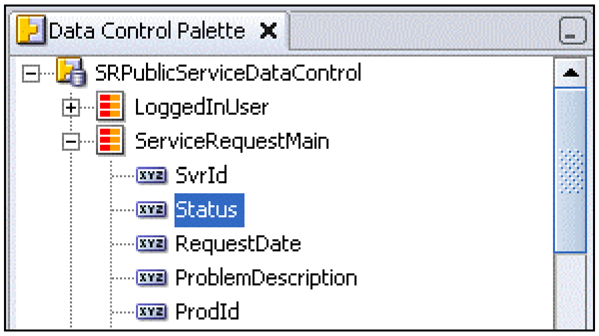


Display Label	Value Binding	Component To Use
<bound to attr label>	SvrId	ADF Output Text w/ ...
<bound to attr label>	RequestDate	ADF Output Text w/ ...
<bound to attr label>	ProblemDescription	ADF Input Text w/ La...

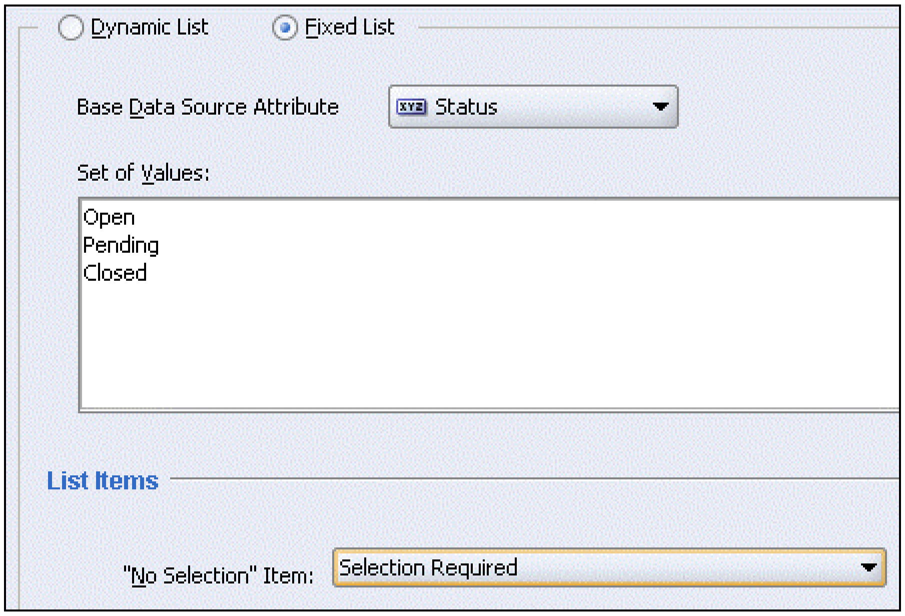
7. Click **OK**.
8. Select the **ProblemDescription** item, and set the Rows property to **4**.

In the next steps, add the Status item and restrict the available values.

9. In the Data Control palette, expand the **`SRPublicServiceDataControl > ServiceRequestMain`** nodes and select the **Status** attribute.



10. Drag the **Status** attribute into the Structure pane and drop it on the **`af:panelForm`**.
11. On the context menu, select **`Single Selections > ADF Select One Choice`**.
12. In the List Binding Editor, select the **Fixed List** radio button.
13. Set Base Data Source Attribute to **Status**.
14. In the Set of Values property, type in the following three values, one on each line:
    - **Open**
    - **Pending**
    - **Closed**
15. For the No Selection Item property, select **Selection Required**.


Dynamic List [✓ Fixed List]

Base Data Source Attribute: Status

Set of Values:

Open
Pending
Closed

List Items

"No Selection" Item: Selection Required

16. Click **OK** to create the Status item.

In the next steps, add the AssignedDate to the page.

17. Expand the **`SRPublicServiceDataControl > ServiceRequestMain`** nodes and select the **AssignedDate** attribute.
18. Drag the **AssignedDate** attribute into the Structure pane and drop it on the **af:panelForm**.

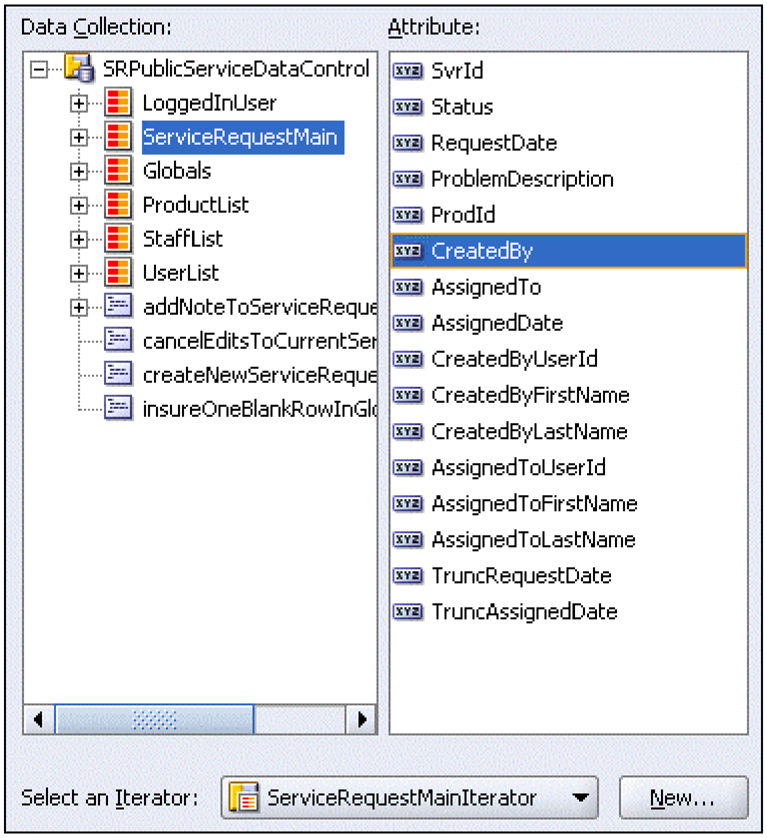
19. On the context menu, select `Dates > ADF Input Date w/ Label`.

Creating the CreatedBy Items

When the form was created, the CreatedBy items were included. Create their bindings and then add the items to the page.

In the next steps create the bindings.

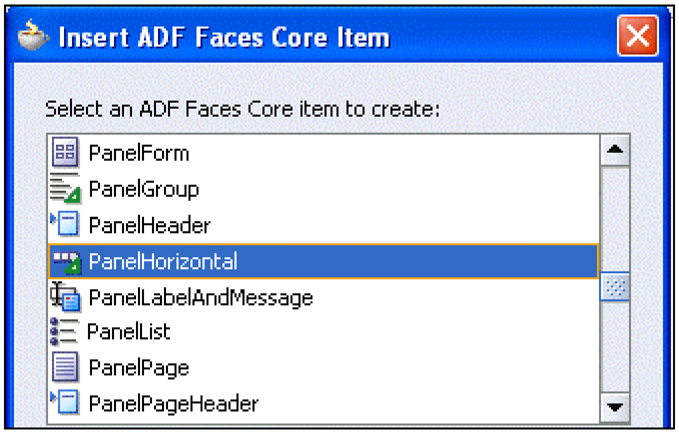
1. Right-click anywhere on the page, and, on the context menu, select **Go to Page Definition**.
2. In the Structure pane right-click the **bindings** node. On the context menu, select `Insert inside bindings > attributeValues`.
3. In the Attribute Binding Editor, expand the **SRPublicServiceDataControl** and select the **ServiceRequestMain** view object.
4. In the Attribute side, select **CreatedBy** and ensure that the iterator is set to **ServiceRequestMainIterator**.



5. Click **OK**.
6. Select the **bindings** node again, and insert another attributeValue. This time for the **AssignedTo** attribute of the **ServiceRequestMainIterator** iterator.
7. Use **Save All** to save your work.

In the next few steps, add createdBy data to the page.

8. Click the **SREdit.jspx** tab to bring the page to the forefront.
9. In the Component palette, select the **ADF Faces Core** category.
10. Scroll down and drag a **PanelLabelAndMessage** component onto the `af:panelForm`.
11. In the Property Inspector, set the label to `#{bindings.CreatedBy.label}`
12. In the Structure pane, right-click the new `af:panelLabelAndMessage` and, on the context menu, select `Insert inside af:panelLabelAndMessage > ADF Faces Core`.
13. In the dialog box, scroll down and select the **PanelHorizontal**.



14. Click **OK**.
15. Expand the `af:panelHorizontal > PanelHorizontal facets` nodes, and select the **separator** node.
16. Right-click the **separator** node and, on the context menu, select `Insert inside separator > ADF Faces Core`.
17. Scroll down the list, select **ObjectSpacer** and click **OK**.

Add the CreatedBy first and last names to the panel horizontal area:

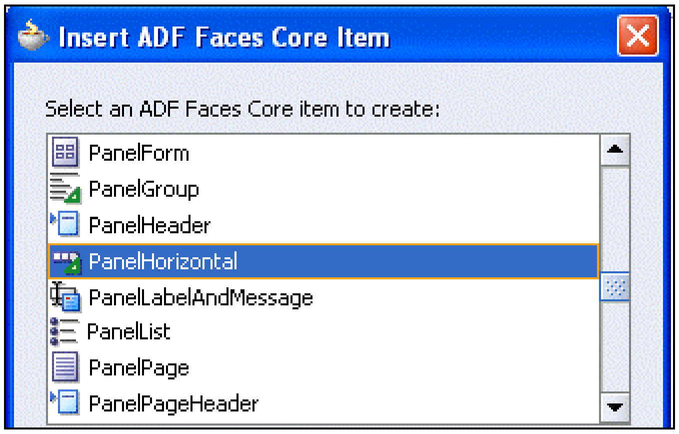
18. In the Data Control palette, expand the **ServiceRequestMain** view object, and select the **CreatedByFirstName** attribute.
19. Drag the **CreatedByFirstName** attribute into the Structure pane, and drop it on the new **af:panelHorizontal**.
20. On the context menu, select **Texts > ADF Output Text**.
21. Drop the **CreatedByLastName** attribute onto the `af:panelHorizontal` as an **ADF Output Text** (without a label). The CreatedBy component will display the user's first and last name on the page.

Creating the AssignedTo Items

Now that the created-by items are on the form, perform the same activities but only for the AssignedTo attributes. Now you will add a list to display technicians

1. In the Component palette, select the **ADF Faces Core** category.

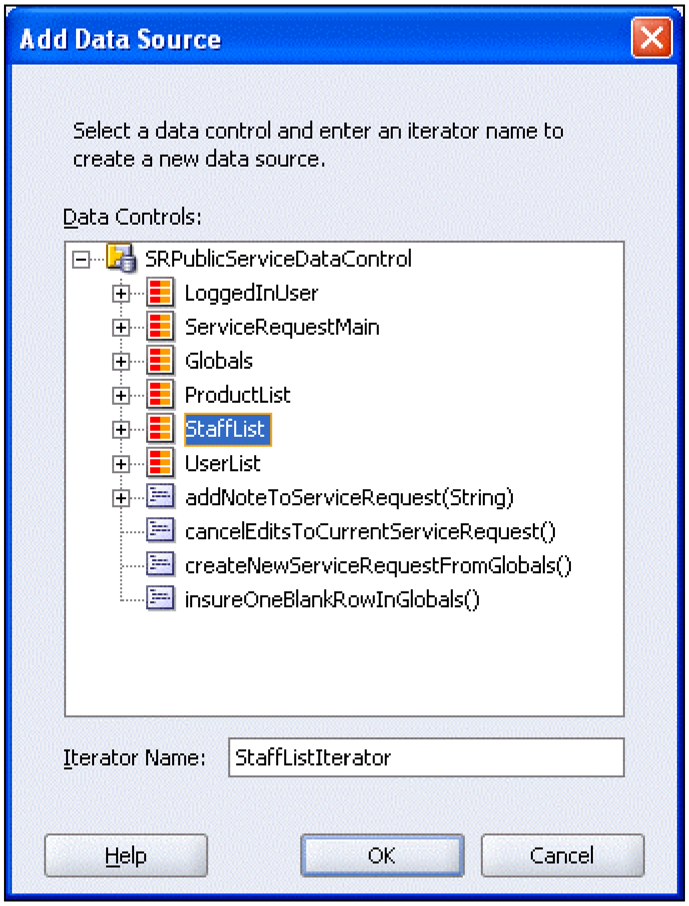
2. Drop another **PanelLabelAndMessage** component from the component palette and drop it on the `af:panelForm`
3. Set the Label property to **`#{bindings.AssignedTo.label}`**.
4. In the Structure pane, right-click the new **`af:panelLabelAndMessage`** and, on the context menu, select **Insert inside af:panelLabelAndMessage > ADF Faces Core**.
5. In the dialog box, scroll down and select the **PanelHorizontal**.



6. Click **OK**.
7. Expand the **af:panelHorizontal > PanelHorizontal facets** nodes, and select the **separator** node.
8. Right-click the **separator** node and, on the context menu, select **Insert inside separator > ADF Faces Core**.
9. Scroll down the list, select **ObjectSpacer** and click **OK**.

Add the assigned to first and last names to the panel horizontal area.

10. In the Data Control palette, expand the **ServiceRequestMain** view object, and select the **AssignedToFirstName** attribute.
11. Drag the **AssignedToFirstName** attribute into the Structure pane, and drop it on the new af:panelHorizontal.
12. On the context menu, select **Texts > ADF Output Text**.
13. Drop the **AssignedToLastName** attribute onto the af:panelHorizontal as **Texts** > **ADF Output Text**.
14. Drag the **AssignedTo** attribute and drop it on the af:panelForm as **Single Selection** > **ADF Select One Choice**.
15. In the List Binding Editor, click the **Add** button next to the List Data Source.
16. Create a new data source for the **StaffList** view object and name it for the **StaffListIterator.**



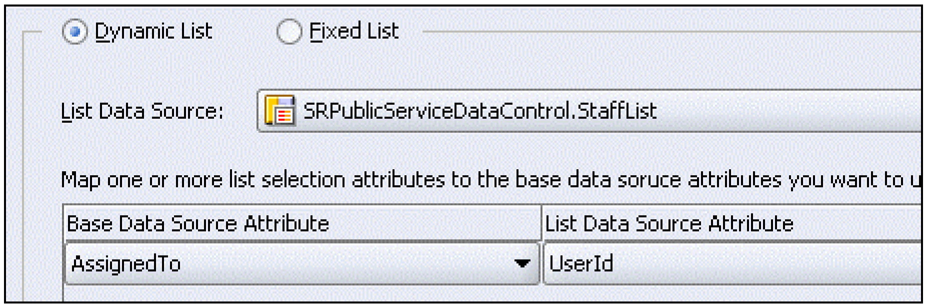
Select a data control and enter an iterator name to create a new data source.

Data Controls:

- SRPublicServiceDataControl
  - LoggedInUser
  - ServiceRequestMain
  - Globals
  - ProductList
  - StaffList
  - UserList
  - addNoteToServiceRequest(String)
  - cancelEditsToCurrentServiceRequest()
  - createNewServiceRequestFromGlobals()
  - insureOneBlankRowInGlobals()

Iterator Name: StaffListIterator

17. Click **OK**.
18. Make sure the Base Data Source Attribute is **AssignedTo** and the List Data Source Attribute is **UserId**.
19. Set the Display Attribute to **Name**.

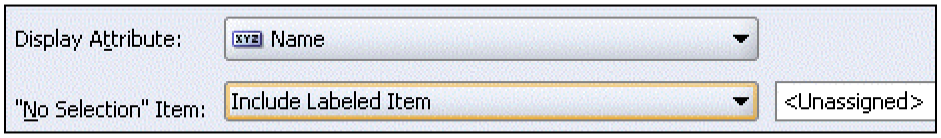

[✓ Dynamic List] Fixed List

List Data Source: SRPublicServiceDataControl.StaffList

Map one or more list selection attributes to the base data soruce attributes you want to u

Base Data Source Attribute	List Data Source Attribute
AssignedTo	UserId

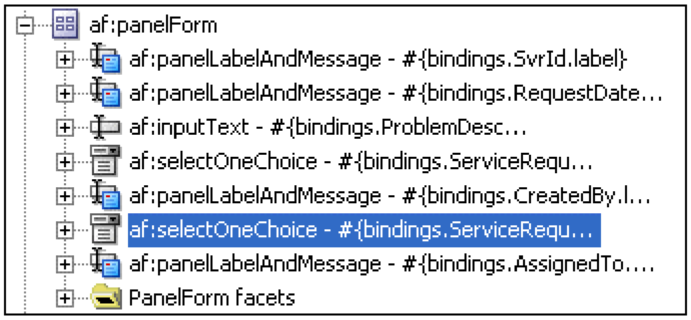
20. Set the No Selection Item property to **Include Labeled Item** and the text of the no selection item to **<Unassigned>**.


Display Attribute: Name

"No Selection" Item: Include Labeled Item <Unassigned>

21. Click **OK** to finish creating the assigned-to attributes.

22. In the Structure pane, sequence the `af:SelectOneChoice - Assigned to` item so it is between the **CreatedBy** and **AssignedTo** panelLabelAndMessage components.



```
af:panelForm
  af:panelLabelAndMessage - #{bindings.SvrId.label}
  af:panelLabelAndMessage - #{bindings.RequestDate...
  af:inputText - #{bindings.ProblemDesc...
  af:selectOneChoice - #{bindings.ServiceRequ...
  af:panelLabelAndMessage - #{bindings.CreatedBy.l...
  af:selectOneChoice - #{bindings.ServiceRequ...
  af:panelLabelAndMessage - #{bindings.AssignedTo....
  PanelForm facets
```

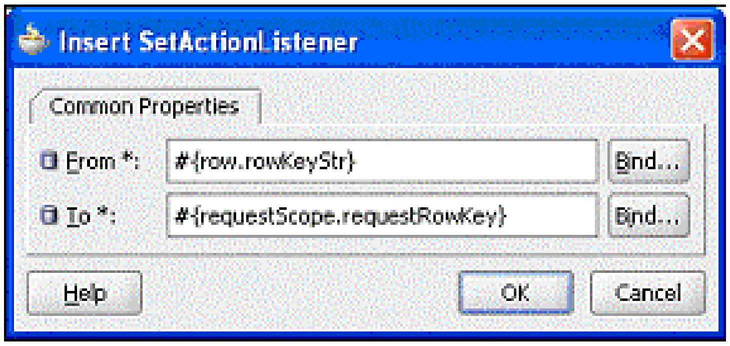
23. Use **Save All** to save your work.

Linking the Page into the Application

The page will add service history records to the current service request being viewed. However, there is no facility yet to determine what service request should be displayed in the SRMain page. The `#{requestScope.requestRowKey}` expression referenced by the setCurrentRowWithKey action binding you created ealier will evaluate to *null* unless you set its value to the key of the desired row to view on the calling page. In the next section modify the SRList page to have the (View) button set the value of `requestScope.requestRowKey` to the key of the selected record before navigating to the SRMain page. Then on the SRMain page, add a button to navigate back to the home page, SRList.

Linking the SRList page

1. Open the **SRList.jspx** page if it is not already open.
2. Right-click the **Edit** button and select **Insert inside CommandButton > ADF Faces Core > SetActionListener**.
3. Set the From property to `#{row.rowKeyStr}` and the To property to `#{requestScope.requestRowKey}`.



Insert SetActionListener

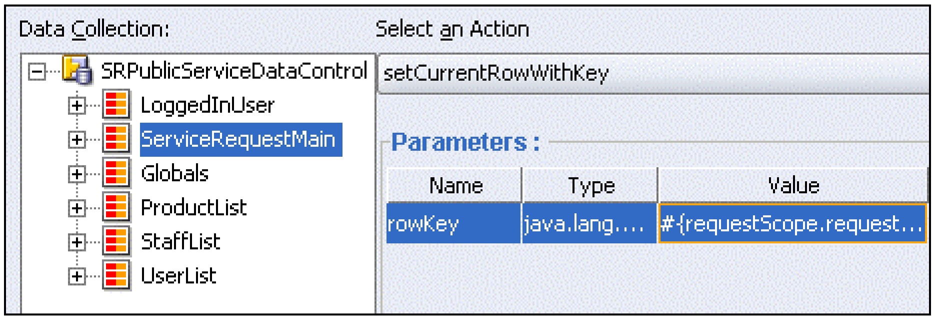
Common Properties

From *: #{row.rowKeyStr}

To *: #{requestScope.requestRowKey}

4. Open the **SREdit.jspx** page if it is not already open.
5. Right-click the page and select **Go to Page Definition**.
6. In the Structure pane, right-click the **bindings** node and select **Insert inside bindings > action**.

7. In the ActionBinding Editor, expand the **SRPublicServiceDataControl** and select the **ServiceRequestMain** node.
8. In the Select an Action property, select `setCurrentRowWithKey`.
9. In the Parameters area, set the Value property to `#{requestScope.requestRowKey}`.



Data Collection:

- SRPublicServiceDataControl
  - LoggedInUser
  - ServiceRequestMain
  - Globals
  - ProductList
  - StaffList
  - UserList

Select an Action: setCurrentRowWithKey

Parameters :

Name	Type	Value
rowKey	java.lang....	#{requestScope.request...

10. Ensure that the iterator is set to **ServiceRequestMainIterator**.
11. Click **OK**.
12. In the Structure pane, expand the **executables** node and right-click the **ServiceRequestsMainIterator** node.
13. On the context menu, select **InsertBefore > invokeAction**.
14. In the Insert invokeAction dialog box, set the ID to `setRequestToEdit` and the Binds property to `setCurrentRowWithKey`.
15. Click the Advanced Properties tab. Set the Refresh property to **prepareModel** and the RefreshCondition property to `#{adfFacesContext.postback == false}`.
16. Use **Save All** to save your work.

Linking the SRSearch page

In the next few steps, create an action listener on the search page for the Edit button.

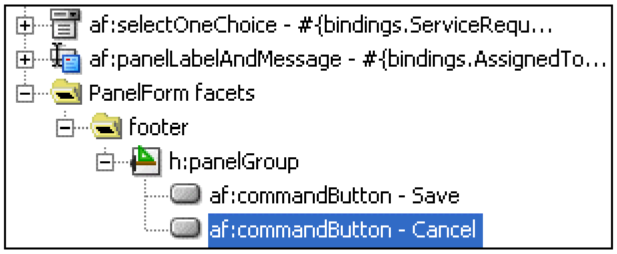
1. Open the **SRSearch.jspx** page and select the **Edit** button.
2. Right-click the Edit button and select Insert inside CommandButton > ADF Faces Core > SetActionListener.
3. Set the From property to `#{row.rowKeyStr}` and the To property to `#{requestScope.requestRowKey}`.
4. Use **Save All** to save your work.

Adding Buttons to the Edit Page

In the next few steps, add transaction control buttons on the edit page.

1. In the Structure pane, expand the `af:panelPage > af:panelForm > PanelForm facets` nodes, exposing the **footer**.
2. In the Data Control palette, expand the `SRPublicServiceDataControl > Operations` nodes.
3. Drag the **Commit** operation and drop it onto the **footer** facet.

4. On the context menu, select **ADF CommandButton**.
5. Change the Text property of the button to **Save**, and, in the Disabled property, click the **Reset to Default** icon to set the property to **false**. Also set **Immediate** to **false.**
6. From the Data Control Palette, drag the **`cancelEditsToCurrentServiceRequest`** method as an ADF Command Button to the **footer** item of the panel form. Notice that when you add the second button, a panelGroup is added to the footer facet, to better discriminate the two command buttons.
7. Set the Text property to **Cancel**, the Action property to **Home** and Immediate to **true.**



```
af:selectOneChoice - #{bindings.ServiceRequ...
af:panelLabelAndMessage - #{bindings.AssignedTo...
PanelForm facets
  footer
    h:panelGroup
      af:commandButton - Save
      af:commandButton - Cancel
```

In the next steps, add a button to navigate to the SRList page.

8. From the Component Palette, select the **ADF Faces Core** category.
9. Drag a **CommandButton** to the Structure pane and drop it on the **`af:panelPage`** node.
10. Sequence it before the **`af:panelForm`**.
11. Set the Text property to **Home**, the Action property to **Home**, and the Immediate property to **true**.
12. In the Structure pane, select the **`af:selectOneChoice - ServiceRequestMainAssignedTo`** item.
13. In the Property Inspector, set the Id property to **assignedToDropDown**.
14. Expand the `af:panelLabelAndMessage (AssignedTo)`, right-click the **`af:panelHorizontal`** containing the AssignedToFirstName and AssignedToLastName, and then select **Properties** on the context menu.
15. Select the **PartialTriggers** property and click the button in the value field.
16. In the Partial Triggers dialog box, click **New** and select the **assignedToDropDown** list item from the drop-down list.
17. Click **OK** twice to exit the properties window.
18. Sequence the items in the following order:
    - `SvrId`
    - `CreatedBy`
    - `RequestDate`
    - `AssignedTo`
    - `ServiceRequestMainStatus`
    - `ServiceRequestMainAssignedTo`
    - `ServiceRequestMainAssignedDate`

- `ProblemDescription`

19. Use **Save All** to save your work.

Changing the Application Look and Feel

You can change the look and feel of an application by changing its skin. A *skin* is a global style sheet that needs to be set in only one place for the entire application. The application developer does not need to add any code, and any changes to the skin are picked up at run time. Skins are based on the Cascading Style Sheets specification.

By default, ADF Faces applications use the Oracle skin, and so this is the look and feel that has been applied to the pages that you have created in the SRTutorialADFBC application. ADF Faces also provides two other skins, the Minimal skin (which provides some basic formatting) and the Simple skin (which provides almost no special formatting). Alternatively, you can create a custom skin specifically for your application; to find out how to do this, refer to the JDeveloper Help pages.

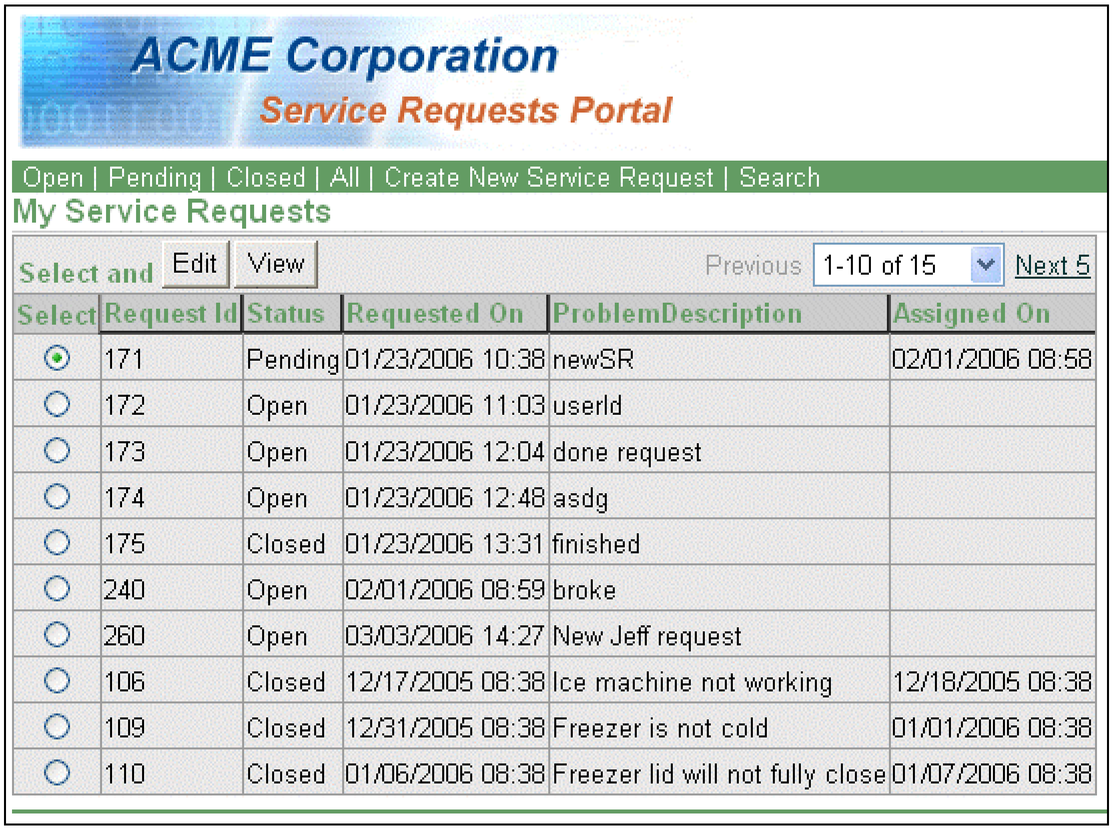
Perform the following steps to apply the Minimal skin to your SRTutorialADFBC application:

1. In the Navigator, locate the **`UserInterface > Web Content > WEB-INF > adf-faces-config.xml`** file and double-click it to open it.
2. Change the <skin-family> value from **`<skin-family>oracle</skin-family>`** to **`<skin-family>minimal</skin-family>`**.
3. Save the file.

To see the new look and feel, run the application.

4. Close the Visual Editor for the **SRList** page and re-open the page.

   Notice that the components in the Visual Editor also display the new style.
5. Run the **SRList** page. The page with the Minimal skin applied should look like the screenshot below. Notice that the buttons are squared off, the menu bar is simplified, and the labels have green backgrounds. If you see an error page (e.g., 500 Internal), try restarting OC4J then run it again.



Select	Request Id	Status	Requested On	ProblemDescription	Assigned On
●	171	Pending	01/23/2006 10:38	newSR	02/01/2006 08:58
○	172	Open	01/23/2006 11:03	userId	
○	173	Open	01/23/2006 12:04	done request	
○	174	Open	01/23/2006 12:48	asdg	
○	175	Closed	01/23/2006 13:31	finished	
○	240	Open	02/01/2006 08:59	broke	
○	260	Open	03/03/2006 14:27	New Jeff request	
○	106	Closed	12/17/2005 08:38	Ice machine not working	12/18/2005 08:38
○	109	Closed	12/31/2005 08:38	Freezer is not cold	01/01/2006 08:38
○	110	Closed	01/06/2006 08:38	Freezer lid will not fully close	01/07/2006 08:38

Summary

In this chapter, you created an Edit page that enables managers and technicians to modify service requests. To accomplish this, you performed the following key tasks:

- Created data view components
- Created the page outline
- Added data components to the page
- Linked the page into the application
- Changed the application look and feel

10

Enabling Security and Deploying the Application to Oracle Application Server 10*g*

In this chapter, you implement security for the application. You also use JDeveloper to create a deployable J2EE Web archive that contains your application and a few required deployment descriptors. You deploy the application to Oracle Application Server 10*g* using the JDeveloper deployment mechanism. You can then test the application and view its performance by using Oracle Enterprise Manager.

This chapter contains the following sections:

Introduction

You should implement both authentication and authorization in the environment where the SRTutorial application will run. Authentication determines which users get access to the application, and it is controlled by the roles assigned to each user. Authorization controls what type of behavior users are allowed to perform after they enter the application.

Security in SRTutorial is based on J2EE container security. The available roles for the application are (all lowercase): user, technician, and manager

Deploying a J2EE application is the last step in developing an application. After the application is completed and works as planned, the next step is to deploy it to a location where customers can use it.

JDeveloper has the built-in capability to deploy applications to a number of application servers. In this chapter, you deploy your application to Oracle Application Server 10*g*.

You perform the following key tasks in this chapter:

- **Implement security features:** Using user and role definitions, you use predefined security levels to access the Web Application
- **Create a connection to an existing Oracle Application Server 10*g*:** JDeveloper needs to establish a connection to the target application server so it can create the correct deployment profiles and push the completed files to the server.
- **Start Enterprise Manager (EM):** With EM, you can monitor or even redeploy an application.

  Note: If you do not have access to Oracle Application Server 10*g*, you will start an OC4J instance (this is an instance of OC4J that is not run from within JDeveloper).
- **Deploy the application:** After the profiles are created, you can deploy the application from within JDeveloper.
- **Test the deployment:** When the application is deployed, you can run it from a Web browser to verify that the application works as expected.
- **Use Enterprise Manager to explore the application server:** Enterprise Manager provides a detailed view of all the components of the application server. You can monitor and even change application parameters and fine-tune the performance of the application server.

Note: If you did not successfully complete Chapter 9, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named `Chapter10` to hold the starter application. If you used the default settings, it should be in `<jdev_install>\jdev\mywork\Chapter10`.
2. Unzip `<tutorial_setup>\starterApplications\SRTutorialADFBC-EndOfChapter9.zip` into this new directory. Using a new, separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRTutorial application workspace.
4. Select **File > Open**, and then select `<jdev_install>\jdev\mywork\Chapter10\SRTutorialADFBC\SRTutorialADFBC.jws`. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 9.

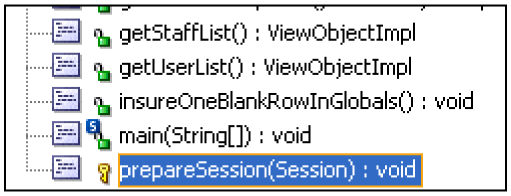
Implementing Application Security

Two areas of the application need to be modified to incorporate security into the application. The method of the application module currently contains a hard-coded reference to the user, "sking", which needs to be changed to accept the current logged-in user. In this section, also include an appropriate security file and link it into the application

Incorporating Security into the Data Model

In this section, you modify the application module to use the current logged-in user and set the configuration to use security.

1. In the Applications Navigator, expand the `oracle.srtutorial.datamodel` nodes and select the **SRPublicService** node.
2. From the structure window, double-click **SRPublicServiceImpl.java** class. The SRPublicServiceImpl.java class is displayed in the editor
3. In the Structure pane, scroll down and double-click the **prepareSession(Session): void** node.


getStaffList() : ViewObjectImpl
getUserList() : ViewObjectImpl
insureOneBlankRowInGlobals() : void
main(String[]) : void
prepareSession(Session) : void

4. In the editor, the method is highlighted. Replace the line of code where the "**sking**" user was hard coded with code to get the logged-in user.

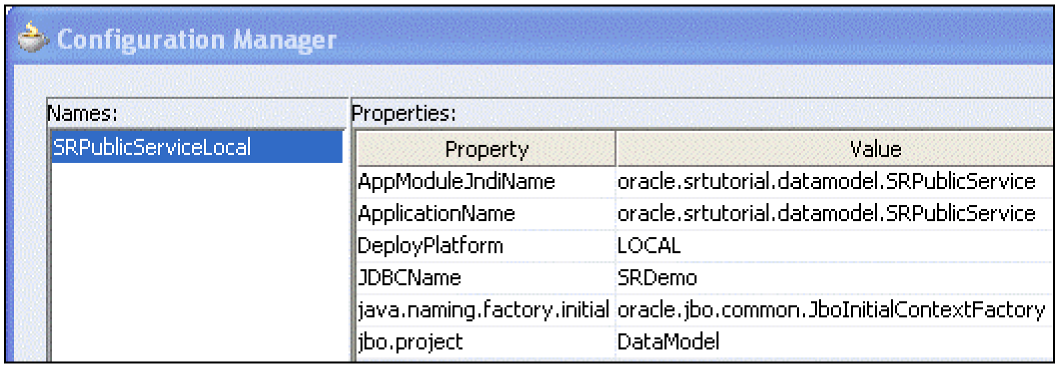
   Change - `String currentser = "sking";`

   To - `String currentUser = getUserPrincipalName();`

5. Save your work.

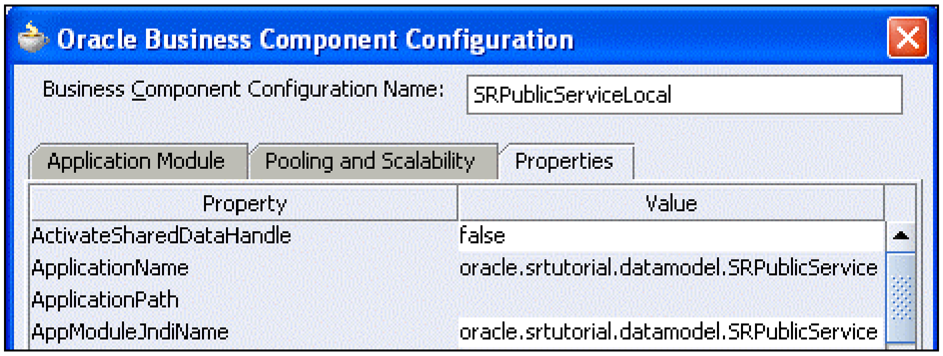
In the next steps, update the SRPublicServiceLocal configuration of the SRPublicService application module to enable security enforcement

6. In the Applications Navigator, expand the `oracle.srtutorial.datamodel` nodes and right-click the **SRPublicService**.
7. From the context menu, select **Configurations**.
8. In the Configuration Manager, ensure that the **SRPublicServiceLocal** name is selected, and click the **Edit** button.



Property	Value
AppModuleJndiName	oracle.srtutorial.datamodel.SRPublicService
ApplicationName	oracle.srtutorial.datamodel.SRPublicService
DeployPlatform	LOCAL
JDBCName	SRDemo
java.naming.factory.initial	oracle.jbo.common.JboInitialContextFactory
jbo.project	DataModel

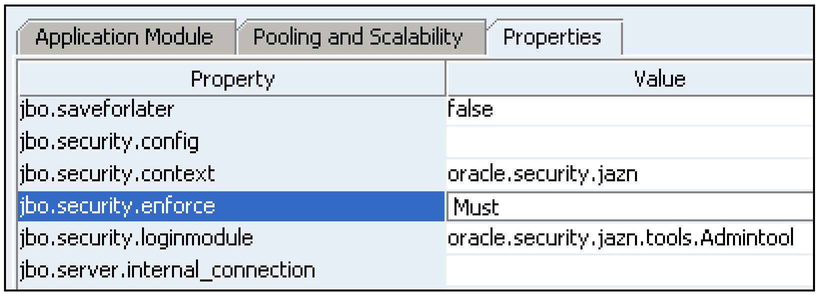
9. In the dialog box, click the **Properties** tab.



Business Component Configuration Name: SRPublicServiceLocal

Property	Value
ActivateSharedDataHandle	false
ApplicationName	oracle.srtutorial.datamodel.SRPublicService
ApplicationPath	
AppModuleJndiName	oracle.srtutorial.datamodel.SRPublicService

10. Scroll down, select the **jbo.security.enforce** property, and set it to `Must` to enforce Java Authentication and Authorization Service (JAAS).



Property	Value
jbo.saveforlater	false
jbo.security.config	
jbo.security.context	oracle.security.jazn
jbo.security.enforce	Must
jbo.security.loginmodule	oracle.security.jazn.tools.Admintool
jbo.server.internal_connection	

11. Click **OK** to exit the Configuration. Notice that the jbo.security.enforce property is added to the configuration.

12. Click **OK** again, to complete the configuration.

Adding Security to the Application

Security features implementation requires some additional files: the jazn-data.xml file, containing the users and roles definitions, and the orion-applicatio.xml file, pointing to resources definition name and location. The security files already exist in the setup, you just need to integrate them in your application. This section illustrates how to proceed with the security files.

1. In Windows Explorer (or the equivalent in your operating system), navigate to the directory where you unzipped the setup files.

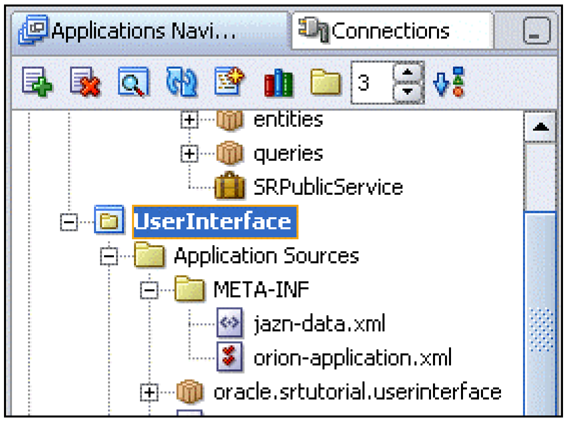
2. Expand the setup file directory. Right-click the **`META-INF`** folder and select the **Copy** option on the context menu.

3. In Windows Explorer, navigate to the directory where your application is stored: **`<jdevinstall>\jdev\mywork\SRTutorialADFBC\SRTutorialADFBC.jws`**. Expand the **`UserInterface`** node, right-click the **`src`** folder and select the **Paste** option from the context menu to add the META-INF folder to your project.

4. Close the Windows Explorer window.

5. Back in JDeveloper, select the **`UserInterface`** project and select **View | Refresh**. The META-INF folder should be loaded in the project.

6. Expand the **`META-INF`** folder. The Applications Navigator should now look like this:



7. Double-click the **`orion-application.xml`** file to open it in the editor.

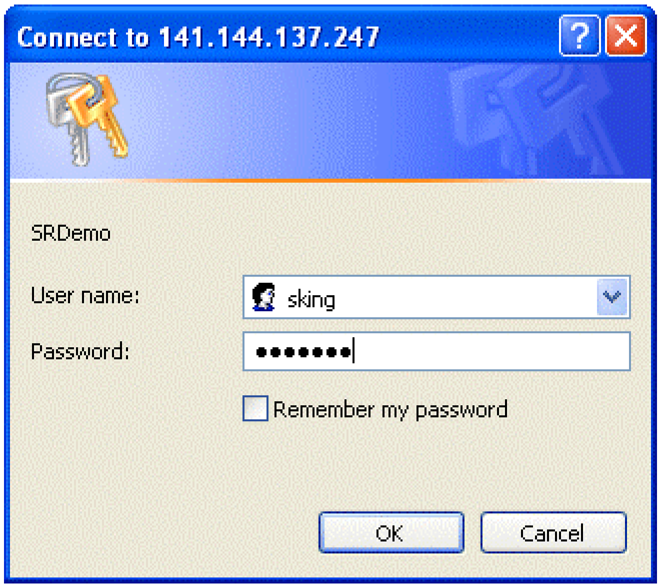
8. Review the content of the file. It specifies the path and filename for data sources, and Java Authentication and Authorization Service (JAAS)
9. Double-click the **`jazn-data.xml`** file to open it in the editor.

   Review the content of the file and notice the structure of the file content composed of user and role definitions.
10. In the Applications Navigator, expand the following nodes: **`UserInterface > Web Content > WEB-INF`**, and double-click the web.xml file to open it.
11. Scroll down to the bottom of the file and copy the following XML code, which defines the security constraints, before the last statement in the file. The code is also found in a file (securityConstraints.txt) within the **`files`** directory of the setup.

```
 <security-constraint>
    <web-resource-collection>
        <web-resource-name>AllManagers</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>AllUsers</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>BASIC</auth-method>
    <realm-name>SRDemo</realm-name>
</login-config>
<security-role>
    <role-name>user</role-name>
</security-role>
<security-role>
    <role-name>technician</role-name>
</security-role>
<security-role>
    <role-name>manager</role-name>
</security-role>
```

12. Locate the **SRList.jspx** page in the Applications Navigator and run it.
13. Now running the application requires providing an authorized user and password to

access the application. Enter **sking/welcome** on the security dialog. Click **OK**.



14. The List page should now display in your default browser.

15. Close the browser window running the application.

Creating a Connection to Oracle Application Server 10*g*

JDeveloper supports deploying your applications to a variety of production application servers, via application sever connections. The first step in using JDeveloper to deploy an application is to create a connection to the target application server. This section illustrates how to create a connection to an already existing and running application server.

1. Click the **Connections** tab in JDeveloper.

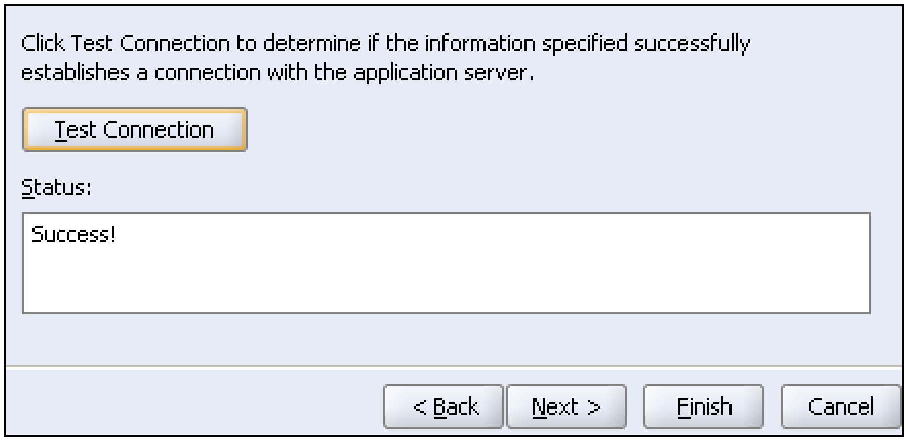
   If the Connections tab is not visible, select **View > Connections Manager** from the JDeveloper menu bar. (You can also press [Ctrl] + Shift + [O].)

2. Right-click **Application Server** in the Connections window and select **New Application Server Connection** on the context menu.

3. Enter the following values in the Create Application Server Connection Wizard:

Field	Value
Connection Name	`OracleAS10g`
Connection Type	`Oracle Application Server 10g 10.1.3`
Username	`oc4jadmin`
Password	`welcome` (or whatever the administrator password for your instance.)
Host Name	`Localhost`

4. On the last page of the wizard, click **Test Connection**. You should see a success message.


Click Test Connection to determine if the information specified successfully establishes a connection with the application server.

Status:

Success!

5. When the test is successful, click **Finish** to create the connection.

Note: If you do not have access to Oracle Application Server 10*g*, follow the next sections to create an instance of OC4J and a JDeveloper connection.

Starting an OC4J Instance

JDeveloper includes an installation of Oracle Application Server Containers for J2EE (OC4J) and a JDK. In this section, you start the stand-alone OC4J, you navigate to the directory where the `oc4j.jar` file is stored and start OC4J using the supplied version of the JDK.

1. Open a Windows Explorer window.
2. Open the **<jdev_home>\jdev\bin** directory.
3. Click the **start_oc4j .bat** file to start running it. (If start_oc4j.bat closes too quickly, append the DOS command **pause** to the end of the file; after the :end line label).
4. A command window opens. (If this is the first time you have run OC4J, it automatically installs the environment.)
5. When ready, it prompts you for a password for the administrator account. The administrator account is `oc4jadmin`. Enter the password, **`welcome`**.
6. The installation asks you to confirm the password by entering it twice.
7. When the install and startup are complete, you will see the following message:
   ```
   Oracle Containers for J2EE 10g <10.1.3.1.0> initialized
   ```
8. Once ready leave the command window open.

OC4J is now running and ready for use.

Creating a Connection to OC4J

The general steps to create a connection to any application server are basically the same. The differences come from the specific connection requirements of the server. Creating a connection to Oracle Application Server 10*g* is fundamentally the same as creating a connection to OC4J. There are only a few differences in the arguments you supply.

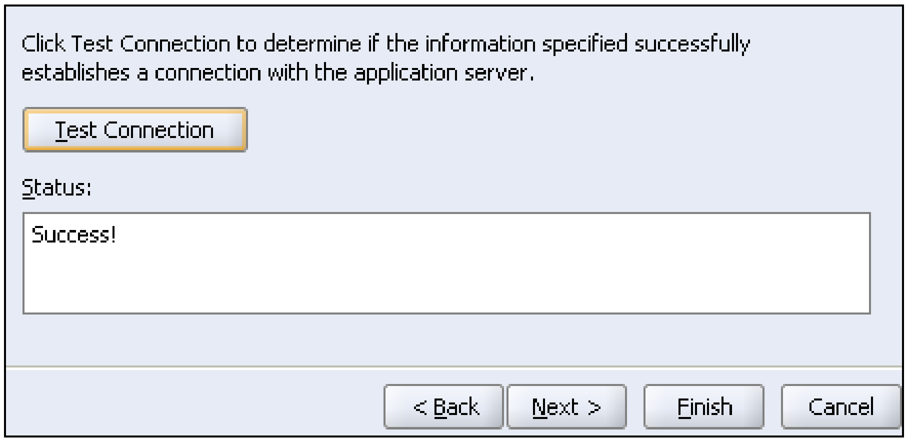
1. Click the **Connections** tab in JDeveloper.

   If the Connections tab is not visible, select **View > Connections Manager** from the JDeveloper menu bar. (You can also press **[Ctrl] + [Shift] + [O]**.)

2. Right-click **Application Server** in the Connections window and select **New Application Server Connection** on the context menu.
3. Enter the following values in the Create Application Server Connection Wizard:

Field	Value
Connection Name	`OC4J`
Connection Type	`Standalone OC4J 10g 10.1.3`
Username	`oc4jadmin`
Password	`welcome`
Host Name	`localhost`

4. On the last page of the wizard, click **Test Connection**. You should see a success message.



Click Test Connection to determine if the information specified successfully establishes a connection with the application server.

Success!

5. When the test is successful, click **Finish** to create the connection.

Starting Enterprise Manager

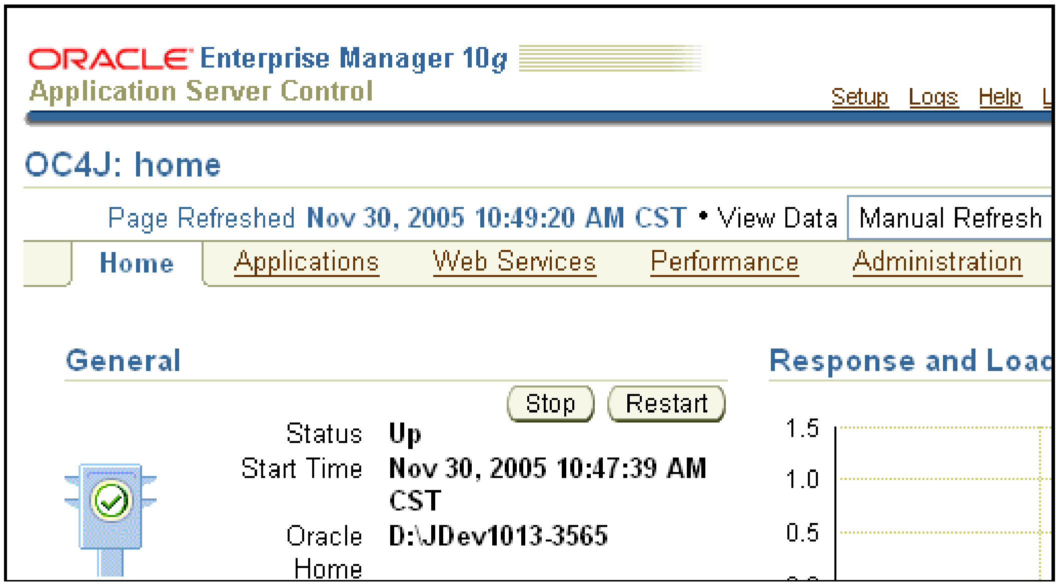
Oracle Application Server 10*g* comes with a browser-based Enterprise Manager (EM). Through this EM interface, you can monitor activities and applications deployed to the application server. After the application server is running, you can connect to EM using a browser. The next few steps open this interface and briefly explore the application server.

The stand-alone OC4J that comes with JDeveloper also includes Enterprise Manager.

1. Open a browser of your choice (Firefox, Internet Explorer, or another browser) and enter the following address: **http://<hostname>:7777/em**. If it does not work with port 7777 or 7780, to determine the port number to use you would need to open, on the application server, the **opmn.xml** file located in oracle_home\opmn\conf. Look for the "request" attribute of the "<opmn><notification-server><port> element to find the correct port to use. If you are using stand-alone OC4J, the address is **http://127.0.0.1:8888/em** or http://localhost:8888/em.

2. Enterprise Manager 10*g* prompts you for a username and password. The username is **`oc4jadmin`** with a password of **`welcome`** (or your administrator password). Click Login to enter EM.

3. After successful login, the browser looks like the following:



OC4J: home

Page Refreshed Nov 30, 2005 10:49:20 AM CST • View Data Manual Refresh

General

Status Up
Start Time Nov 30, 2005 10:47:39 AM CST
Oracle Home D:\JDev1013-3565

Response and Loac

You can now explore applications, Web services, and other components of Oracle Application Server 10*g*.

Creating a Deployment Profile

Deployment profiles are project components that manage the deployment of an application. A deployment profile lists the source files, deployment descriptors (as needed), and other auxiliary files that will be included in a deployment package.

There are three parts of the deployment package for the service request application: The Model project (`.ear`), the UserInterface project (`.war`), and the Deployment project (`.ear.`) files. When deploying an ADF Business Components application, you only need to create a deployment profile for the UserInterface project, and because of the dependencies that your UI project has with the DataModel one, the deployment process automatically integrates the Business Components elements.

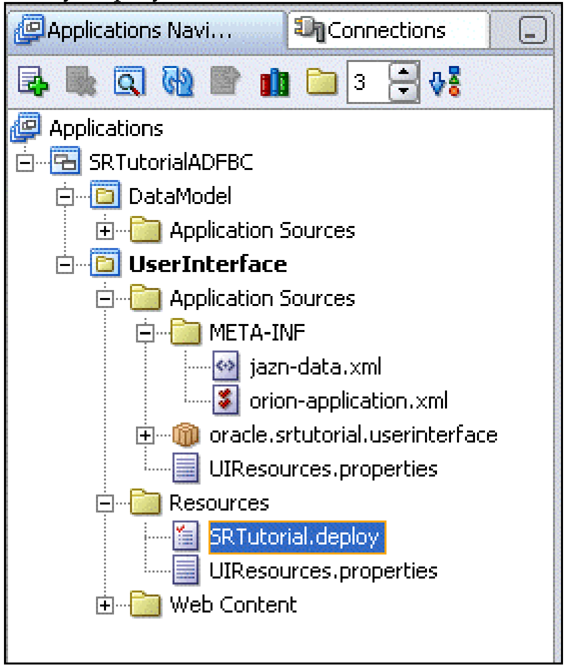
Creating the User Interface Deployment Profile

You now create a deployment profile for the UserInterface project. This project is where you created the user interface components of the application. The deployment file for this project is a `.war` file (Web Archive, for the Web components).

1. Right-click the **UserInterface** project in the Applications Navigator, and select **New** on the context menu.

2. In the New Gallery, under the General node, select **Deployment Profiles, WAR File** as the item, and click **OK**.

3. In the Create Deployment Profile, enter `SRTutorial` as the name. Click **OK**.
4. In the WAR Deployment Profile dialog, select the **Specify J2EE Web Context Root** and change the Web Application's Context Root to **SRTutorial**. This becomes part of the URL that customers use to access the application.
5. Click **OK** to accept the default values in the WAR Deployment Profile Properties dialog box.
6. Select the **UserInterface** project node and click the **Refresh** button.
7. Save your work.
8. The Applications Navigator should now look like the following screenshot showing a newly deployed file in the Resources folder:

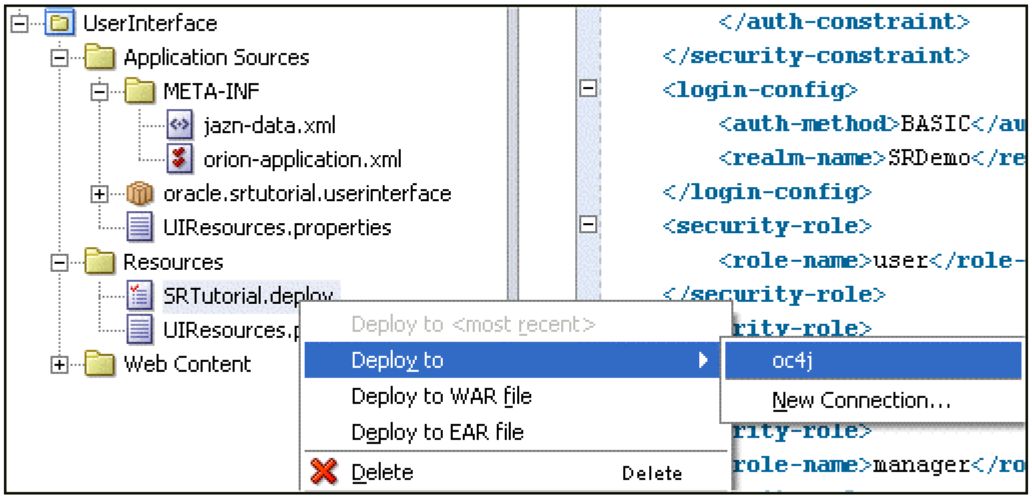


Deploying the Application

JDeveloper provides a one-click option to deploy an application to an application server. After you have assembled the application into a WAR file, you can right-click the deployment profile and select the target application server.

1. Right-click the `SRTutorial.deploy` deployment profile.
2. Select **Deploy to** > **OracleAS10g** or **OC4J** from the context menu, depending on what Application Server connection you defined. Remember that OracleAS10g is the name of the connection to Oracle Application Server 10*g* you created earlier in this tutorial.

Note: If you did not have access to Oracle Application Server 10*g* and used OC4J, use the name of the connection you created for OC4J, which should be **OC4J**.

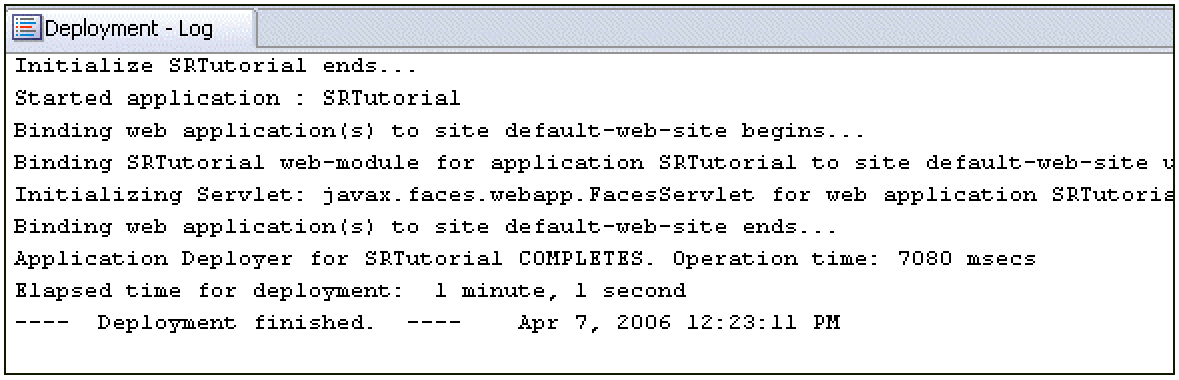


```
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</au
        <realm-name>SRDemo</re
    </login-config>
    <security-role>
        <role-name>user</role-
    </security-role>
```

During deployment, JDeveloper re-creates the `.war` file and then assembles an `.ear` file. After the file is assembled, JDeveloper deploys the file and unpacks it in a directory on the application server, depending on the target environment.

3. Click **OK** to accept the Application Configuration and begin the deployment.

4. When the deployment is complete, the following messages appear in the JDeveloper log window:



```
Initialize SRTutorial ends...
Started application : SRTutorial
Binding web application(s) to site default-web-site begins...
Binding SRTutorial web-module for application SRTutorial to site default-web-site u
Initializing Servlet: javax.faces.webapp.FacesServlet for web application SRTutoria
Binding web application(s) to site default-web-site ends...
Application Deployer for SRTutorial COMPLETES. Operation time: 7080 msecs
Elapsed time for deployment:  1 minute, 1 second
----  Deployment finished.  ----    Apr 7, 2006 12:23:11 PM
```

5. Opening the Command window shows the deployment execution in the stand-alone OC4J application server.

Testing the Application

You have now deployed the application to a stand-alone instance of OC4J. You can test the application with a Web browser using the context root that you specified for the application.

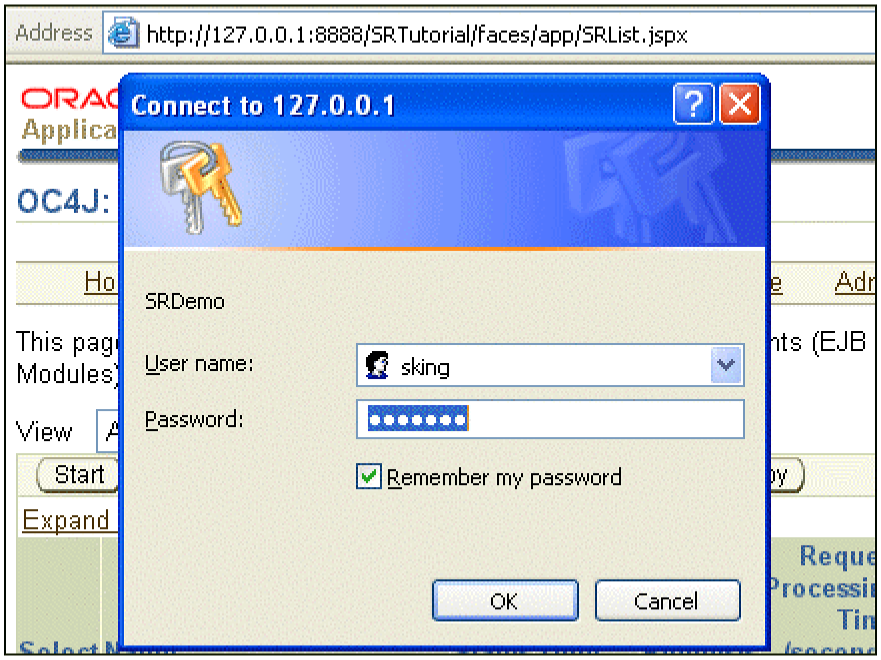
1. Open a Web browser.

2. Enter the URL, http://localhost:8888/SRTutorial/faces/app/SRList.jspx for the external OC4J or , http://localhost:7777/SRTutorial/faces/app/SRList.jspx for Application Server.

> **Note:** If you are using OC4J, the URL is the same except for the port number. Substitute **8888** for 7777.

3. This directs you to the login security screen of the application. Use the following as login credentials:

Field	Value
Username	`sking`
Password	`welcome`



4. When you have explored the application, leave the browser open.

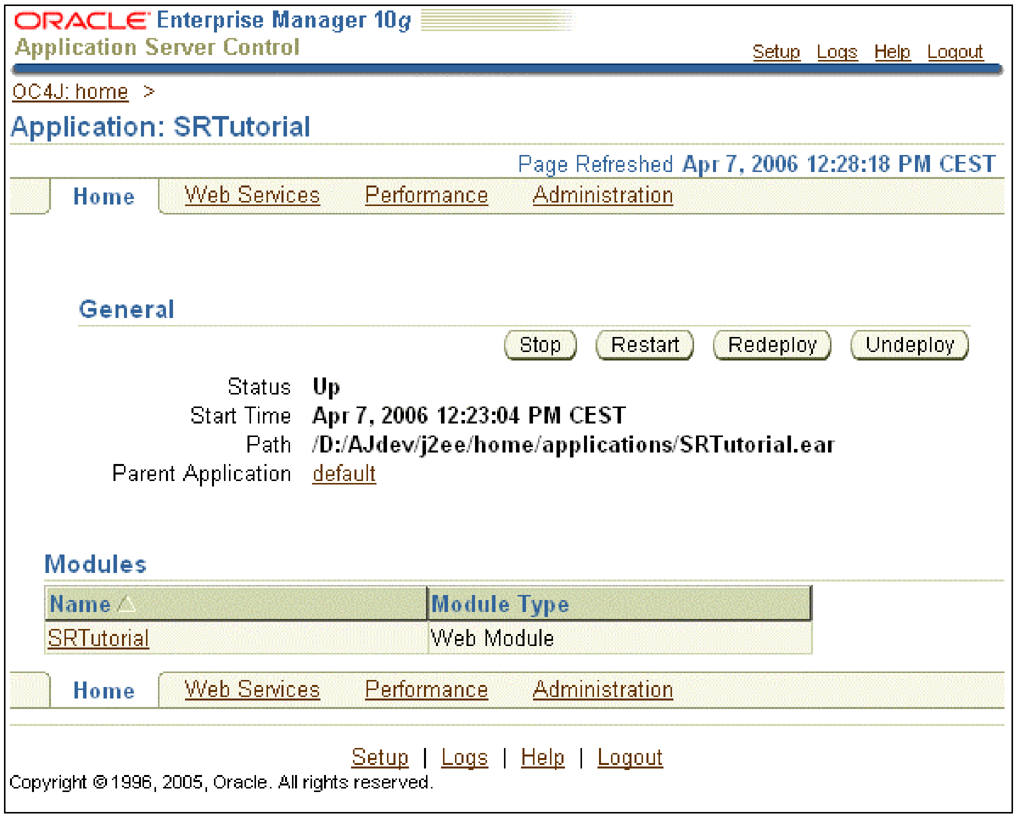
Using Enterprise Manager to Explore the Application Server

OC4J is delivered with a Web-based Enterprise Manager (EM) that you can use to monitor and manage deployed applications. In this final section, you use EM to explore your application and the application server.

1. Open a second Web browser.
2. Enter the URL, **http://localhost:7777/em** (for OC4J, use **8888**).
3. Use the following credentials to log in to EM:

Field	Value
Username	`oc4jadmin`
Password	`admin`

4. After you log in, you are directed to the EM home page, which should look like the following screenshot:



Application: SRTutorial

Page Refreshed Apr 7, 2006 12:28:18 PM CEST

General

Status	Up
Start Time	Apr 7, 2006 12:23:04 PM CEST
Path	/D:/AJdev/j2ee/home/applications/SRTutorial.ear
Parent Application	default

Modules

Name △	Module Type
SRTutorial	Web Module

Copyright © 1996, 2005, Oracle. All rights reserved.

You can explore a number of aspects of the application using EM, such as the following:

5. Click the **Applications** tab, and then click **SRTutorial** to view specifics about the application you just deployed.
6. Click the **Performance** link to see graphs of application performance.
7. Click the **Administration** link to see all of the deployment aspects of the application, including details of the JAZN security implementation.

Summary

In this chapter, you saw how to use JDeveloper to deploy an application to OC4J. JDeveloper enables you to easily deploy an application to a number of different application servers by using deployment profiles and application server connections.

Enterprise Manager is an easy-to-use and powerful interface for the management of Oracle Application Server and deployed applications.

Here are the key tasks that you performed in this chapter:

- Added resource files to implement security
- Created a connection to an existing Oracle Application Server 10*g*

- Created deployment profiles
- Started Enterprise Manager (EM)
- Deployed the application
- Tested the deployment
- Used Enterprise Manager to explore the application server