Oracle Application Server 10*g* Release 3: Tutorial for Java EE Developers (10.1.3.1.0)

September 2006

Oracle Application Server 10*g* Release 3: Tutorial for Java EE Developers (10.1.3.1.0)

Copyright © 2006, Oracle. All rights reserved.

Primary Authors: Pam Gamer, Dan Hynes, Raghu Kodali , Dana Singleterry

Contributors: Alfred Franci, Jonas Jacobi, Lynn Munsinger, Frank Nimphius, Chris Schalk

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

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

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

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

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

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

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

Preface

This preface outlines the contents and audience for the Oracle Application Server 10*g* R3: Tutorial for Java EE 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 JDeveloper and Enterprise JavaBeans (EJBs) 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 Persistence Entities"

This chapter describes how to build the persistence data model for your application by using EJB 3.0 Java Persistence API (JPA) entities.

Chapter 3: "Developing the Session Facade"

This chapter describes how to develop session facade beans and interfaces to present client objects with a unified interface to the underlying EJBs.

Chapter 4: "Deploying and Testing the Data Model"

This chapter describes how to start OC4J and deploy the session bean and how to test the data model by creating and running a sample client that inserts records into the database.

Chapter 5: "Planning the User Interface"

This chapter describes how to plan the user interface for the application, including defining the page flow and standardizing the application's look and feel.

Chapter 6: "Defining Login Logic"

This chapter describes how to build security for the SRDemo application.

Chapter 7: "Implementing Login, Display, and Navigation Logic"

In this chapter, you create pages that implement the login logic. You also create a page to conditionally display links enabling users to navigate to the pages that they are authorized to use.

Chapter 8: "Developing a Page to List Records"

In this chapter, you develop a page that lists a user's service requests.

Chapter 9: "Developing Pages to Edit Records"

This chapter describes how to build the pages to edit a service request. There is one page for customers and a different page for staff (technicians and managers.)

Chapter 10: "Creating the Triage Page"

This chapter describes how to create a page that enables managers to triage (assign) service requests to technicians.

Chapter 11: "Developing a Page to Insert Records"

This chapter describes how to create an insert page that enables users to insert new service request records into the database.

Chapter 12: "Installing Oracle Application Server 10*g*"

This chapter describes how to install two clustered instances of Oracle Application Server 10*g* for deployment of the SRDemo application.

Chapter 13: "Deploying the Application to Oracle Application Server 10*g*"

This chapter describes how to use JDeveloper to create a deployable package that contains your application and required deployment descriptors. It also describes using Application Server Control to deploy the package to the Oracle Application Server 10*g* cluster, and then performing a rolling upgrade of the application.

Related Documents

For more information about building applications EJB 3.0, see *EJB 3.0 Resources* at http://www.oracle.com/technology/tech/java/ejb30.html.

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

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

1

Getting Started

This tutorial describes how to design and build a Java EE customer relationship management application that tracks customer service requests. The application has Web-based user interfaces and is deployed to Oracle Application Server 10*g*. Data components are EJB3.0 objects.

This chapter contains the following topics:

- ServiceCompany Overview
- Using This Tutorial
- Setting Up Your Environment
- Summary

ServiceCompany Overview

ServiceCompany is a large appliance-servicing company that provides service support for household appliances (dishwashers, washing machines, microwaves, and so on). The company supports a wide variety of appliances and tries to solve most customer issues by providing answers to questions via service requests. ServiceCompany has found that customers can eventually resolve most issues when they have the correct information. This approach has proven to save time and money for both the company and its customers.

A service request generally has the following flow:

1. A customer issues a request.
2. The company assigns the request to a service technician.
3. The service technician answers the request or asks the customer for more information.
4. The customer checks the request and either closes it or provides further information.

Business Problem

Currently this process is initiated by telephone and requires the services of a clerical staff to log and follow up on service requests. As volume has increased, assignments to service technicians are sometimes delayed until clerks can perform data entry of the service requests. It is difficult for managers to keep track of the work that technicians do to ensure efficient performance. These factors have resulted in declining customer satisfaction and increasing costs per request for the company.

ServiceCompany wants to implement a self-service application so that customers can log and track their own requests, and so that managers can better monitor the work of technicians.

Goal

ServiceCompany has the following goals:

- A customer interface that customers can use to add, update, and check the status of their service requests
- A business interface with which the company can create, update, and manage customer service requests, which includes adding service information and assigning requests to the appropriate service technician.
- Various reporting tools to ensure timely resolution of service requests

Business Solution

ServiceCompany decides to use Oracle JDeveloper to create a Java EE customer relationship management application. The technical aspects of the application include the following:

- The user interfaces are Web-based to enable deployment of the application both externally to customers and internally to employees.
- The technology scope is Java EE compliant.
- The application server is Oracle Application Server 10*g*. ServiceCompany uses the Oracle Application Server platform to enable service-oriented architecture (SOA) in its enterprise, so the application can be deployed there.
- The data components are EJB3.0 objects. Enterprise JavaBeans (EJB) 3.0 provides the business service layer of the application. An Enterprise JavaBean is a reusable, portable Java EE component. The EJB 3.0 specification makes it easier than ever to develop Enterprise JavaBeans, and JDeveloper enables wizard-driven creation of EJBs.

- Java ServerFaces (JSF) is used for the application's UI.

The following is a list of functions and pages to be used in the application. These will change as the application evolves. The first list shows the functions that are needed. The second is a list of required pages.

Functions

The following functionality is required in the application:

- **Customer log-in validation**
- **Service request creation**
- **Service request maintenance**
- **Service history maintenance** (with the ability to hide internal notes from customers)

The following functions are not part of this tutorial but could be done as an optional exercise:

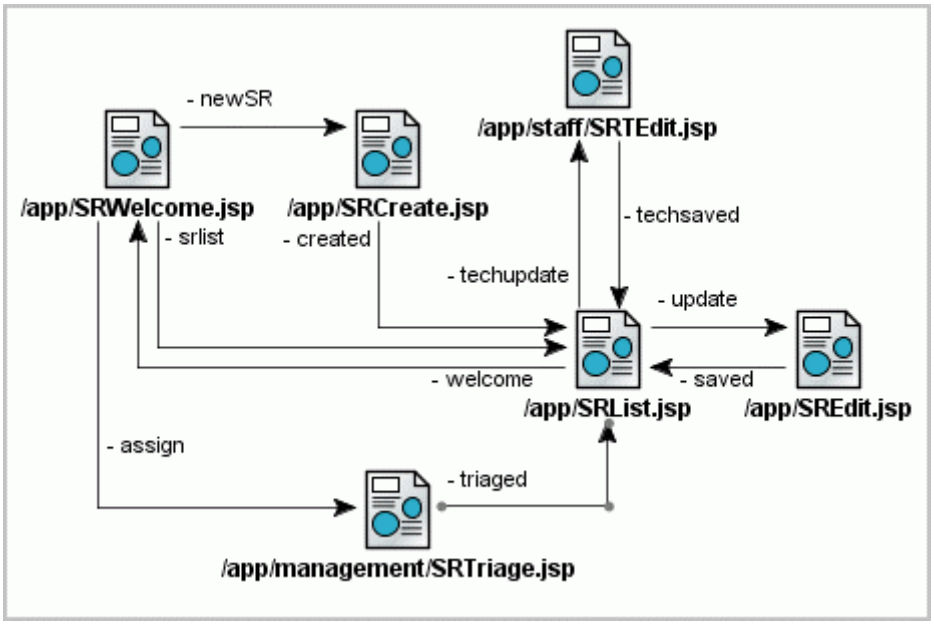
- **History of service request assignments:** As an optional exercise, you could populate a history row with each change of assignment.
- **History of service request status:** As an optional exercise, you could populate a history row with each change of status.

Pages

The application consists of the following pages:

- **Login page:** Each user is a customer, service technician, or manager.
- **Service Request Insert page:** Both customers and staff can use this page to create new service requests (SRs).
- **Service Request List page (customer-facing):** This is the page that customers use to view existing SRs. This page shows all of the SRs that the customer has created and the status of each. Customers can click an existing SR and be directed to an edit page where they can add information to the request (Service Request page).
- **Service Request Edit page (customer-facing):** This is the page that customers use to create or update SRs. The history may include sensitive information that the company does not want customers to see. These entries can be flagged as "internal only" and will be excluded from customers' view.
- **Triage page (company-facing):** This page is used by management to display all SRs. Management can assign or reassign SRs from this page. The page is dynamic to show all requests, unassigned requests, or any other status.
- **Service Request List page (company-facing):** This page displays all of the SRs assigned to the logged-on service technician and the status of each request. The technician can click an existing SR and be directed to a page to add information to the request (Service Request Edit page).
- **Service Request Edit page (company-facing):** This page is where the service technician updates the SR with resolution information. The page will show the original request along with the full SR history (master-detail).

The following graphic provides an overview of the page flow for the application:

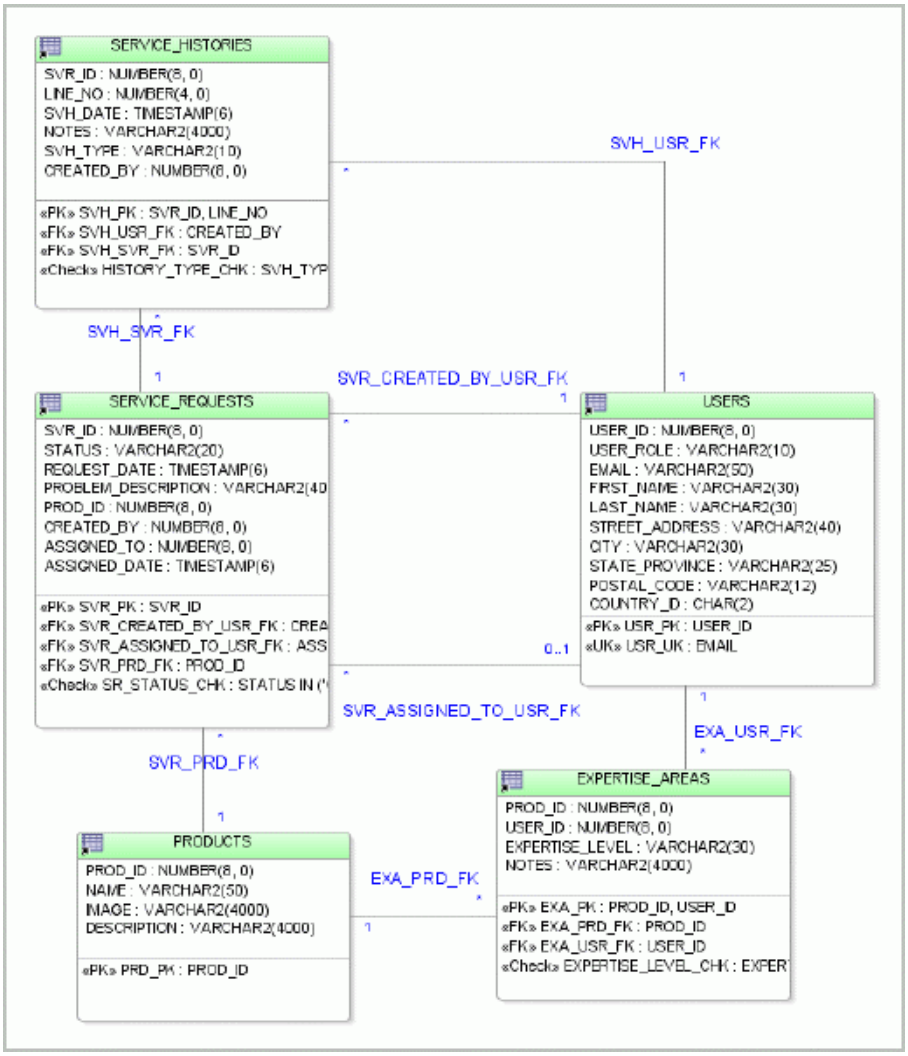


The application flow is as follows:

1. Users invoke the application and are presented with a Welcome page, where they enter log-in information. Users can be customers or ServiceCompany employees. If the log-in information is correct, they navigate to different pages depending on whether they are a customer, a technician, or a manager.
2. If the user who logs in is a customer, the Welcome page displays options to view previously logged SRs or create a new one.
3. If the user who logs in is a technician, the Welcome page displays options to view assigned SRs or create a new one.
4. If the user who logs in is a manager, in addition to being able to view and create SRs, the manager can navigate to pages for assigning an SR or querying for a specific SR or type of SR.
5. If users choose to create an SR, they navigate to a page where they enter SR details.
6. If users choose to view SRs, they navigate to a page that lists the applicable SRs. This list contains links that enable them to update their SRs.
7. If users choose to update, they navigate to a page where they can update SR details.
8. If managers choose to query, they navigate to a page where they can enter query details and display a restricted record or set of records.
9. If managers choose to assign SRs, they navigate to a page that enables them to associate a technician with an SR.

Schema

The initial schema has been designed to provide the needs of the business model. The schema consists of five tables and three database sequences, diagrammed as follows:



The five tables represent creating and assigning an SR 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 an SR is created, the name of the individual who opened it, the product it is for, and the date of the request are all recorded, along with a short description of the problem. When the SR is assigned to a technician, the name of the technician and date of assignment are also recorded. An artificial ID uniquely identifies each SR.

SERVICE_HISTORIES: There may be many events recorded for each SR. 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 an SR are also tracked. The SR ID and the service history sequence number uniquely identify each service history record.

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 each product.

EXPERTISE_AREAS: To better assign technicians to SRs, each technician's specific areas of expertise are defined.

Sequences

USERS_SEQ: Populates the ID for new users

PRODUCTS_SEQ: Populates the ID for each product

SERVICE_REQUESTS_SEQ: Populates the ID for each new SR

Using This Tutorial

This tutorial is divided into sixteen separate chapters. Each chapter builds on the previous one. You must complete each chapter in the order described in this tutorial.

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

Setting Up Your Environment

In this section, you prepare your working environment to support the tutorial. You examine and install the Service Request tables and configure JDeveloper. You perform the following key tasks:

1. Download the tutorial schema setup.
2. Create the `SRDEMO` schema owner and the ServiceCompany schema.
3. Download and install JDeveloper 10*g* Release 3 (10.1.3).
4. Create a JDeveloper database connection.
5. Define an application and projects for the tutorial.

Downloading the Tutorial Setup Files

To download the setup files for this tutorial, perform the following steps:

1. Download `JavaEE_tutorial_setup.zip` from http://download.oracle.com/otndocs/products/jdev/10131/JavaEE_tutorial_setup.zip and extract the zip file to a directory on your computer. This tutorial refers to this directory as the `\setup` directory.
2. From the `\files` subdirectory of the `\setup` directory, unzip `SRSetups.zip` to a temporary directory (such as `D:\temp`) to expose files used to create the user and schema.

Creating the `SRDEMO` Schema Owner and the ServiceCompany Schema

The `SRDEMO` user owns the data displayed in the application. Access to an Oracle `SYS` user or equivalent is required to create the user account and 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. 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 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.

To create the schema owner and schema, perform the following steps:

1. Navigate to the directory where you unzipped the `SRSetups.zip` file.
2. Invoke SQL*Plus from that directory and log on as `SYS` or another DBA-level user. You may need to ask your DBA to run the scripts for you.
3. In the SQL*Plus window start the `build.sql` script:
   `SQLPLUS>`**`start build.sql`**

The `build.sql` script calls the `createSchema.sql` script to create the `SRDEMO` user, and then it logs on to the `SRDEMO` account. After connection, the script automatically calls the `createSchemaObjects.sql` script to create all the tables, constraints, and database sequences. When this operation is complete, the `populateSchemaTables.sql` script is called to insert sample data into the tables.

When control returns to the `build.sql` script, a list displays the created objects and 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 JDeveloper 10*g* Release 3 (10.1.3)

Follow these instructions to prepare JDeveloper Studio.

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

1. In Windows Explorer, navigate to the directory where JDeveloper is installed and find the `jeveloper.exe` file. Create a shortcut to it on the desktop by right-clicking the `jdeveloper.exe` file and choosing **Send To > Desktop** from the context menu.
2. Double-click the **Shortcut to JDeveloper** icon on the desktop to invoke JDeveloper. If this is the first time you are running JDeveloper, a “Do you wish to migrate?” window appears. Click **No** to continue.
3. On startup, a “Tip of the Day” window is displayed. These tips are things you can do to make development more productive. Click **Close** when you’ve finished looking at the tips.

Creating a JDeveloper Database Connection

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

Note: In this 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 along with the names used in the steps.

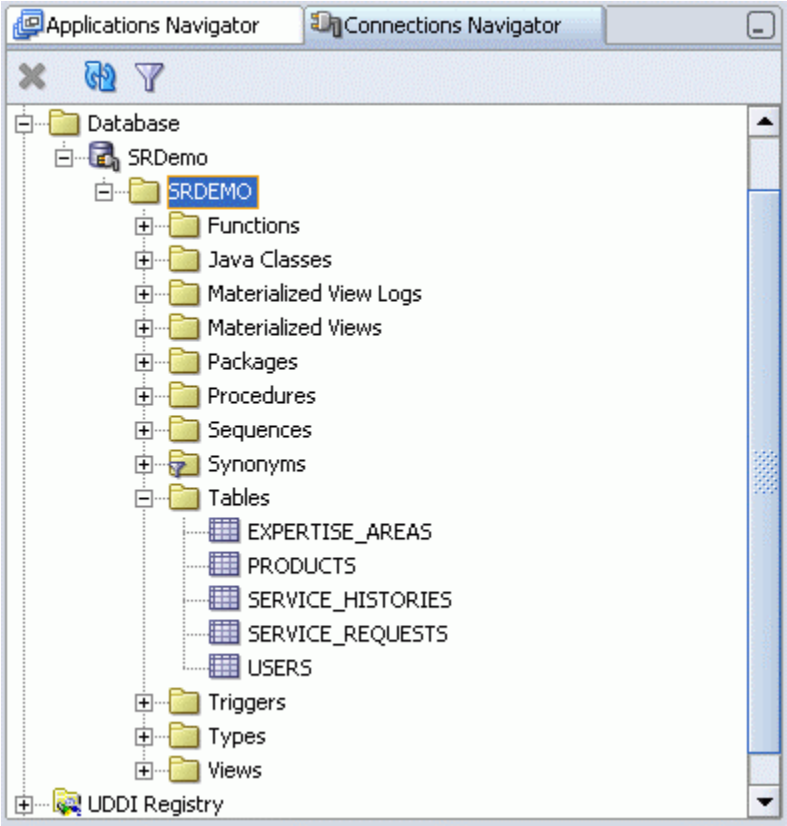
1. In JDeveloper, choose the menu option **View > Connection Navigator**.
2. Right-click the **Database** node and choose **New Database Connection** from the context menu.
3. Click **Next** on the Welcome page.
4. In the Connection Name field, type a name for the connection. The tutorial uses the name **`SRDemo`** (this is case sensitive). Click **Next**.
5. On the Authentication page, enter the following values. 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	Type the name (or IP address) of the computer where the database is located. If the database is on the same machine as JDeveloper, the default value of `localhost` is fine.
JDBC Port	Enter the port used for access to the database. The default value is `1521`. If you do not know this value, check with your DBA.
SID	Enter the SID used to connect to the database. The default value is `ORCL`. If you do not know this value, check with your DBA.

7. Click **Test Connection**. If the database is available and the connection details are correct, you can continue. Otherwise, click the **Back** button and check the values.
8. Click **Finish**. The connection now appears below the Database node in the Connection Navigator.
9. Examine the schema from JDeveloper. In the Connection Navigator, expand **Database > SRDemo**. Browse the database elements for the schema and confirm that they match the following schema definition:



Defining an Application and Projects for the Tutorial

In JDeveloper, you always work in projects contained within applications.

The application is the highest level in the control structure. An application keeps track of any projects you are using. When you open JDeveloper, the last application is opened by default so that you can resume work where you left off.

A JDeveloper project is an organizational structure used to logically group related files. You can add multiple projects to your application to easily organize, access, modify, and reuse your source code.

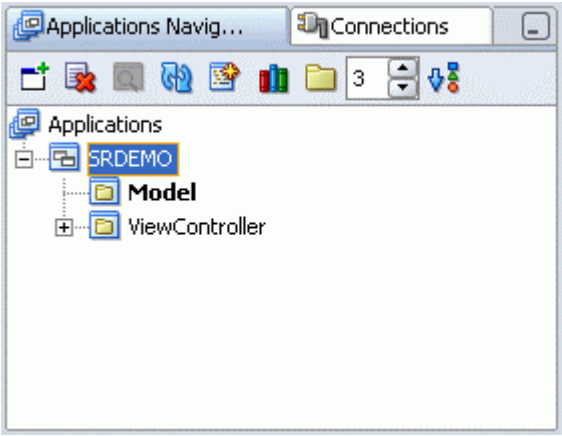
Before you create application components, you must first create the application and its projects. Follow these instructions to create a new `SRDEMO` application and projects for the business model and the view.

> **Note:** Do not include special characters in the project name (such as periods) or in any activity or element names. 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 Application node, and select **New Application** from the context menu.
2. In the **Create Application** dialog box, enter the following values. When finished, click **OK**.

Field	Value
Application Name	`SRDEMO`
Directory name	`D:\JDeveloper\jdev\mywork\SRDEMO` Leave this field at its default value. If you used the recommended directory structure, your path should match this one.
Application Package Prefix	`org.srdemo` This value becomes the prefix for all package names. You can override it later if necessary.
Application Template	`Web Application [JSF, EJB]` In this tutorial, you access technologies related to Java ServerFaces and Enterprise JavaBeans. New templates can be created and added to restrict the technologies that are available during development.

The application is created with two default projects: Model and ViewController.



3. The ViewController project has a dependency on the model because it needs to have access to resources that are defined in the model. However, the model doesn't need components coming from the view side, so no dependency is required for the Model project.

   To set the dependency, right-click the **ViewController** project and select **Project Properties** from the context menu.

4. Select **Dependencies** from the tree at the left. In the Project Dependencies list at the right of the dialog, select the check box next to **Model.jpr**, and then click **OK**.

   Setting this dependency also determines the order in which projects are built. The view needs certain classes that are contained in the model. Setting this dependency thus ensures that the model is compiled before the view.

5. Click **SaveAll** to save your work.

Summary

This chapter introduced you to ServiceCompany and its plan to implement a Web-based customer relationship management application that meets the following requirements:

- Customers can log and update their own service requests.
- Technicians can keep track of and update service requests that are assigned to them.
- Managers can query and assign service requests.

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:

- Downloaded the tutorial schema setup files
- Created the `SRDEMO` schema owner and installed the ServiceCompany schema
- Downloaded and installed JDeveloper
- Created a JDeveloper database connection
- Defined an application and projects for the tutorial

2

Developing Persistence Entities

This chapter of the tutorial describes how to develop persistence entities.

If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh01.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

- Creating EJB 3.0 Java Persistence API (JPA) Entities
- Reviewing Objects That Have Been Created
- Defining Named Queries
- Specifying a Cascade Type for Master-Detail Entities
- Using a Database Sequence to Generate IDs
- Setting Default Values
- Mapping Attributes to Match Subsequent Code
- Summary

Creating EJB 3.0 Java Persistence API (JPA) Entities

In this section, you learn about EJB 3.0 entities and how to create them with JDeveloper.

What Is EJB?

The Enterprise JavaBeans (EJB) architecture is suitable for the development and deployment of component-based business applications. Applications written using the EJB architecture are scalable, transactional, and multiuser secure.

An Enterprise JavaBean (EJB) is a server-side component that encapsulates business logic. The EJB specification defines a server-side component model and programming interfaces for application servers. Servers and containers built under this specification take care of the low-level application programming, such as transaction management, security, and persistence. Because much of the difficult application programming is taken care of by the server, the developer is free to work on the business logic of the application.

Types of EJBs

An EJB can be an entity, a session bean, or a message-driven bean. With EJB 3.0, you no longer need to implement the `EntityBean`, `SessionBean`, or `MessageDrivenBean` interfaces. Instead, you can use annotations to define Java classes as one of the following types of EJBs:

- An **entity** represents a business object that exists in the database.
- A **session bean** performs a distinct, decoupled task such as checking credit history for a customer. Session beans are classified based on the maintenance of the conversation state:
  - **Stateless session beans** do not have an internal state and therefore do not keep track of the information that is passed from one method to another. They are used for business methods that are independent of previous invocations, such as calculating shipping charges. After the shipping charge is calculated and returned to the calling method, there is no need for the session bean to store its state for future invocations.
  - **Stateful session beans**, in contrast, maintain conversational state across invocations and thus can be used for applications such as online shopping carts.
- A **message-driven bean** is used to receive asynchronous JMS messages.

More About Entities

An EJB 3.0 entity is an object that manages persistent data, potentially uses several dependent Java objects, and can be uniquely identified by a primary key. EJB 3.0 entities implement the EJB 3.0 Java Persistence API, which enables you to define plain old java objects (POJOs) as entities by using the `@Entity` annotation. Note the following additional requirements for an entity class:

- It must have a `no-arg` constructor that is public or protected.
- It must be a top-level class.
- It must not be final.
- No methods or persistent instance variables of the entity class may be final.
- It must implement the `Serializable` interface if an entity instance is to be passed by value as a detached object, such as through a remote interface.

EJB 3.0 entities represent persistent data from the database, such as a row in a customer table or an employee record in an employee table. Entities are also sharable across multiple clients. For example, various clients can use an employee entity to calculate the annual salary of an employee or to update the employee address. Specific fields of the entity object can be made persistent. All fields in the entity not

marked with the `@Transient` annotation are considered persistent.

A key feature of EJB 3.0 is the ability to create entities that contain object-relational mappings by using metadata annotations rather than deployment descriptors as in earlier versions. For example, to specify that an entity's `empId` attribute is mapped to the `EMPNO` column of the `EMPLOYEES` table, use the `@Table(name="Employees")` annotation for the table name and the `@Column(name="EMPNO")` annotation for the attribute. You will see and use such annotations in this tutorial. For more information about mapping annotation syntax, see "How-To Use EJB 3.0 O-R Mapping Annotations" on Oracle Technology Network (OTN).

Remote and Local Access

Clients can access EJB entities locally or remotely. Local access is for clients running in the same virtual machine as the EJB, while remote access is for clients running in a different virtual machine, as with a Web application.

Creating the Entities

To create EJB 3.0 entities from a database table, you use the Create CMP Entity Beans from Tables wizard. Note that you must have a database connection to use this wizard. The name of this wizard is a holdover from previous EJB versions, where you created container-managed persistence entity beans.

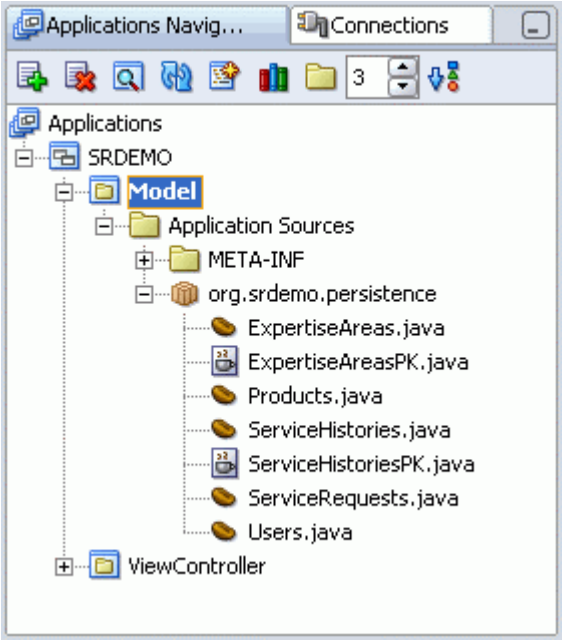
To create EJB 3.0 entities, perform the following steps:

1. Right click the **Model** project and select **New** from the context menu.
2. In the New Gallery, expand the **Business Tier** node. Select **EJB** in the Categories list, and in the Items list select **Entities from Tables (JPA/EJB 3.0)**. Click **OK**.
3. The Create Entities from Tables wizard is displayed. If the Welcome page of the wizard appears, click **Next**.
4. On the Database Connection Details page of the wizard, select **SRDemo** from the Connection dropdown list, and then click **Next**.
5. On the Select Tables page of the wizard, click **Query** to display the available tables. Click **Add All** to shuttle all the tables from the Available list to the Selected list, and then click **Next**.
6. On the Entities from Tables page of the wizard, enter **`org.srdemo.persistence`** as the Package Name. The classes that you create in this package are responsible for persisting application data.
   In the Collection type for Relationship Fields drop down list, select **java.util.List**. Leave other values at their defaults and click **Next**.
7. The Specify Entity Details page of the wizard enables you to change the names of the entities and classes that map to each of the database tables that you selected. Accept the default mappings by clicking **Next**.
8. Click **Finish** on the Summary page of the wizard to create the entities. This may take a few moments. When done, a message appears in the log window to let you know that generation is complete.
9. Click Save All to save your work.

Reviewing the Objects That Have Been Created

You have now created EJB 3.0 entities for each table that you selected and also for composite primary keys. The objects that have been created are plain old java objects (POJOs) that are organized under the

name of the package that you specified in the wizard, as in the following illustration:



What makes these objects different from other Java files are the annotations that identify them as EJB entities and perform object-relational mapping to database tables. These metadata annotations specify to the container how these entities should be managed.

You can open and examine the files to see the annotations by performing the following steps:

1. In the Applications Navigator, expand the **Model** project, and then also expand **Application Sources** and the package **org.srdemo.persistence**. You are able to see all the objects that have been created.
2. Double-click **ServiceHistories.java** to open it in the editor window.
3. You can see the following annotations in the file (you may need to scroll down to see some of them):

Annotation	Description
`@Entity`	Identifies the file as an EJB 3.0 entity
`@NamedQuery`	A query that can be used at run time to retrieve data
`@Table`	Specifies the primary table for the entity
`@IdClass`	Specifies the class for the composite primary key
`@Id`	Can define which property is the identifier for the entity
`@Column`	Specifies a mapped column for a persistent property or field
`@ManyToOne`	Specifies a type of foreign key relationship between tables
`@JoinColumn`	Specifies the join column and referenced column for a foreign key relationship

Defining Named Queries

Named queries enable you to define queries at design time and then use them at run time. You can use annotations to store the query on the entity. These annotations use EJB Query Language (EJBQL) to define the queries.

Including parameters in the query enables the reuse of a query to return different results. Parameters are denoted in EJBQL as strings that are preceded by colons. If you specify a parameter, you must give it a value when calling the query at run time.

For further information about defining EJB 3.0 queries, see "How-To Queries using EJB 3.0" on Oracle Technology Network (OTN).

By default, when JDeveloper creates an EJB 3.0 entity from a table, it adds a named query for retrieving a list of all the data in the table, such as the `ServiceHistories.findAll` named query that you saw in the `ServiceHistories.java` file. At this point, you are not sure what additional queries the application needs, except that users log in with their e-mail address and the application should issue a query to retrieve the corresponding `Users` object. To define the named query that provides the ability to query users by e-mail address, perform the following steps:

1. In the Applications Navigator, double-click the **`Users.java`** file to open it in the editor window.
2. Just above the line with the **`@NamedQuery`** annotation, add the following annotation:
   `@NamedQueries({`

   When prompted, press **[Alt]+[Enter]** to import `javax.persistence.NamedQueries`.
3. On a new line just after the first **`@NamedQuery`** annotation, add the following annotation to the source code:
   ```
   @NamedQuery(name="findUserByEmail",query="select" +
   " object (user) from Users user  where user.email = :email")
   ```
4. Add a comma separator after the first named query (as shown in the example following step 6).
5. Close the set of named queries by adding the following line:
   `})`

   Notice that the first named query does not use parameters, while the second passes a parameter (string preceded by a colon) to the `WHERE` clause of the query. These annotations are shown in the following example:

   ```
   @Entity
   @NamedQueries({
   @NamedQuery(name = "Users.findAll", query = "select o from Users o"),
   @NamedQuery(name="findUserByEmail",query="select" +
   " object (user) from Users user  where user.email = :email")
   })
   public class Users implements Serializable {
       @Column(nullable = false)
   ```
6. Click **SaveAll** to save your work.

You define additional named queries as you continue to develop the application.

Specifying a Cascade Type for Master-Detail Entities

EJB 3.0 introduces the EntityManager API that can persist new entities, remove (delete) existing entities, refresh entity state from the data store, and merge detached entity state back into the persistence context. A

bean instance may be detached and may be updated by a client locally and then sent back to the EntityManager to be merged and synchronized with the database.

Cascading is used to propagate persistence operations to related entities. The default behavior is no cascading.

There are four cascade types:

- `PERSIST` cascades the persist (create) operation to associated entities.
- `MERGE` cascades the merge operation to associated entities.
- `REMOVE` cascades the remove operation to associated entities.
- `REFRESH` cascades the refresh operation to associated entities.

These cascade types can be used alone or in combination with one another. An additional option, `ALL`, can be used to signify a combination of all four cascade types.

Because the ServiceRequests entity has the associated entity ServiceHistories, you want to be sure that the detail Service Histories records are persisted whenever the create operation is called on the master service request. To do so, perform the following steps:

1. Open **ServiceRequests.java** in the editor window, or switch to it if it is already open by clicking its tab.
2. Scroll to the `@OneToMany` annotation that establishes a relationship with the ServiceHistories entity and add **`cascade={CascadeType.PERSIST}`** to the beginning of its attribute list (enclosed in parentheses as shown below). Be sure to include the comma separator.

   When prompted, press **[Alt]+[Enter]** to import `javax.persistence.CascadeType`.

```
@JoinColumn(name = "ASSIGNED_TO", referencedColumnName = "USER_ID")
private Users users1;
@OneToMany(cascade={CascadeType.PERSIST}, mappedBy = "serviceRequests")
private List<ServiceHistories> serviceHistoriesList;
@ManyToOne
```

3. Click **SaveAll** to save your work.

Using a Database Sequence to Generate IDs

New service request records should use the `SERVICE_REQUESTS_SEQ` database sequence to generate the value for the primary key. EJB 3.0 enables you to use annotations to specify how primary keys are to be generated. To specify a generator strategy for `SVR_ID`, perform the following steps:

1. With ServiceRequests.java open in the editor window, scroll to the `@ID` annotation just above the `@Column` annotation for the `SVR_ID` column. On the next line after the `@ID` annotation, add the following code to specify that the entity should use a sequence request generator named `SERVICE_REQUEST_SEQ_GEN` to generate the ID for the `SVR_ID` column:
   `@GeneratedValue(strategy= GenerationType.SEQUENCE, generator="SERVICE_REQUESTS_SEQ")`
2. When prompted, press [Alt]+[Enter] to import `javax.persistence.GeneratedValue` and `javax.persistence.GenerationType`.

```
import javax.persistence.GeneratedValue (Alt-Enter)
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
    generator="SERVICE_REQUESTS_SEQ")
    @Column(name="SVR_ID", nullable = false)
```

Now you have set the sequence generator for the `SVR_ID` column to use the sequence that has been defined in the database, which is `SERVICE_REQUESTS_SEQ`.

3. Click **SaveAll** to save your work.

Setting Default Values

Service request and service history records contain date fields that should default to the current date when they do not contain values. You also need to set line numbers to the next line item for new service history records. To set these default values for null fields, perform the following steps:

4. With ServiceRequests.java open in the editor window, double-click the **setAssignedDate** method in the Structure window to navigate to that method in the editor.
5. Change the line `this.assignedDate = assignedDate;` to the following:

   ```
   this.assignedDate = (assignedDate==null)?new
        Timestamp(System.currentTimeMillis()):assignedDate;
   ```

6. Double-click the **setRequestDate** method in the Structure window to navigate to that method in the editor.
7. Change the line `this.requestDate = requestDate;` to the following:

   ```
   this.requestDate = (requestDate==null)?new
        Timestamp(System.currentTimeMillis()):requestDate;
   ```

8. Open the **ServiceHistories.java** file in the editor (or switch to it if it is already open), and then double-click the **setSvhDate** method in the Structure window to navigate to that method in the editor.
9. Change the line `this.svhDate = svhDate;` to the following:

   ```
   this.svhDate = (svhDate==null)?new
        Timestamp(System.currentTimeMillis()):svhDate;
   ```

10. To default the line number of new records to the next line item, you need to add a new method to the ServiceHistories entity that retrieves the next line item number.

    Add the following method at the end of the file just above the closing right curly bracket, pressing **[Alt]+[Enter]** when prompted to import `java.util.List`:

    ```
    public Long getNextLineItem() {
      // Need to loop through the SRHistrories collection and

      //find the largest value for the Lineno rather than
      // working on the size of the collection -DRM
          Long maxLineNo = new Long(0);
          final ServiceRequests servReqs = getServiceRequests();
          if (servReqs != null)
          {
              final List<ServiceHistories> histList =
                  servReqs.getServiceHistoriesList();
              if (histList != null)
    ```

```
                {
                    for (ServiceHistories svh:histList)
                    {
                        //The collection may have empty nodes so
                        // we just skip those
                        final Long testLineNo = svh.getLineNo();
                        if (testLineNo != null){
                          if (testLineNo > maxLineNo) {
                              maxLineNo = testLineNo;
                          }
                      }
                    }
                }
            }
            return ++maxLineNo;
        }
```

11. Call that method when setting the line number. Modify the `setLineNo()` method by changing the line `this.lineNo = lineNo;` to the following:

    ```
    this.lineNo = (lineNo==null )?getNextLineItem():lineNo;
    ```

12. Click **SaveAll** to save your work.

Mapping Attributes to Match Subsequent Code

The ServiceRequests entity contains two mappings of `Users` objects to the database columns `CREATED_BY` and `ASSIGNED_TO`. To ensure that code in the remainder of this tutorial works properly, `ServiceRequests.users` must be mapped to `CREATED_BY`, and `ServiceRequests.users1` must be mapped to `ASSIGNED_TO`. However, sometimes the mappings are created in the reverse order.

To check that these columns are mapped so that subsequent code works correctly, perform the following steps:

1. With ServiceRequests.java open in the editor window, scroll to the mapping for the **`ASSIGNED_TO`** column. If the attribute that is mapped to that column is named `users`, then change its name to **`users1`**, as shown in the following screenshot:

   ```
   @ManyToOne
   @JoinColumn(name = "ASSIGNED_TO", referencedColumnName = "USER_ID")
   private Users users1;
   ```

2. Now scroll to the mapping for the **`CREATED_BY`** column. If the attribute that is mapped to that column is named `users1`, then change its name to **`users`**, as shown in the following screenshot:

   ```
   @ManyToOne
   @JoinColumn(name = "CREATED_BY", referencedColumnName = "USER_ID")
   private Users users;
   ```

3. Click **Rebuild** to compile the project. The Messages tab of the log window should show successful compilation.

Summary

You have now built a persistent model that uses EJB 3.0 entities. You performed the following key tasks:

- Used the Create CMP Entity Beans from Tables wizard to create EJB 3.0 entities for all tables in the `SRDEMO` schema
- Examined the resulting POJOs to see the annotations that identify them as entities and perform object-relational mapping to database tables
- Used annotations to add named queries, cascade types, and an ID-generation strategy
- Defaulted null date fields to the current date and null line numbers to the next line item
- Ensured that attributes are mapped to match the code in the remainder of this tutorial

3

Developing the Session Facade

This chapter of the tutorial describes how to develop session facade beans and interfaces.

If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh02.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

- Creating a Data Model Session Facade
- Modifying the Query Method to Return a Single Record
- Adding a Method to Return Related Records
- Summary

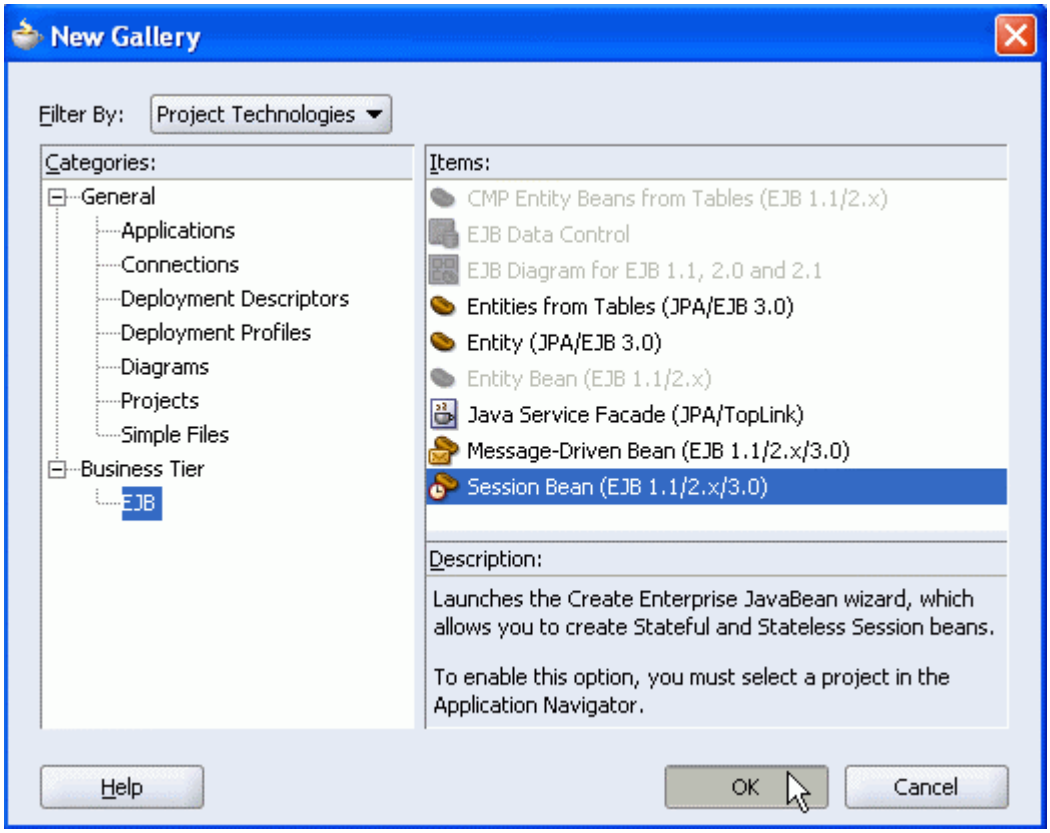
Creating a Data Model Session Facade

A session facade presents client objects with a unified interface to the underlying EJBs (Enterprise Java Beans). The client interacts only with the facade, which resides on the server and invokes the appropriate EJB methods. As a result, dependencies and communication between clients and EJBs are reduced.

If you are performing remote access without a session facade, numerous remote calls are needed for the clients to access EJB 3.0 entities directly over the network. This results in a large amount of network traffic that negatively affects performance. In addition, without a facade the client depends directly on the implementation of the business objects, so that if the interface of an EJB changes, client objects have to be changed as well.

In this section you create a session facade for the EJB data model that you built in the previous chapter by performing the following steps:

1. Right-click the **Model** node in the Applications Navigator and select **New** from the context menu.
2. In the New Gallery, expand **Business Tier** in the Categories list and select **EJB**. Then select **Session Bean (EJB 1.1/2.x/3.0)** from the Items list (as shown in the following screenshot) and click **OK** to invoke the Create Session Bean wizard.



3. If the Welcome page of the Create Session Bean wizard appears, read the information and then click **Next**.

4. On the EJB Name and Options page of the wizard, enter **`ServiceRequestFacade`** as the EJB Name.

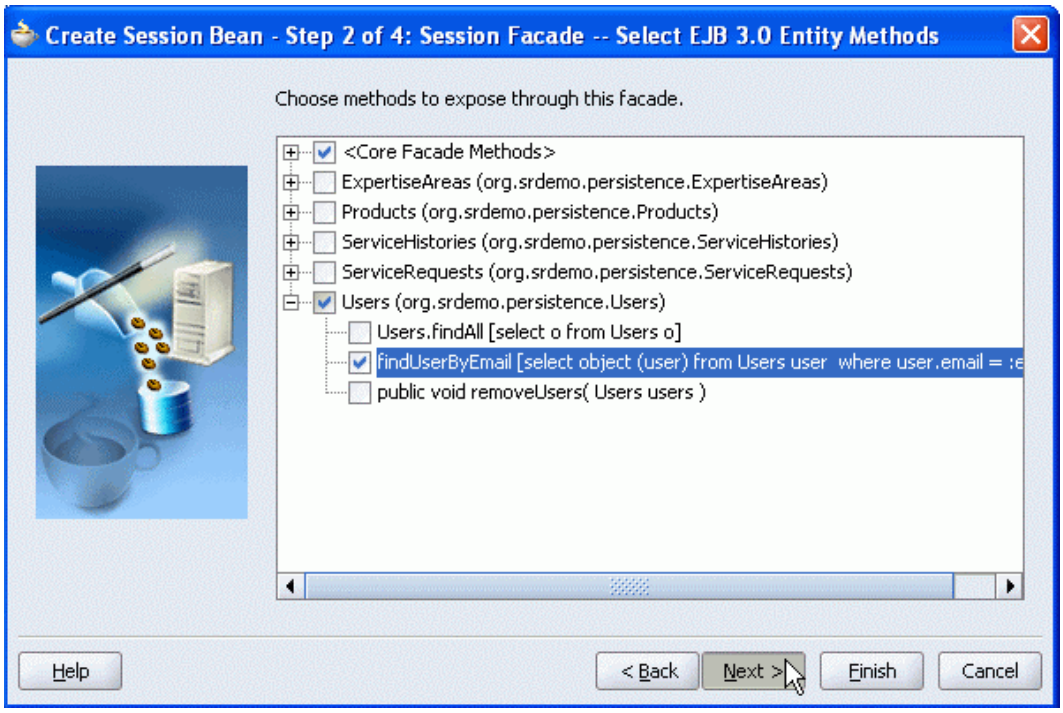
   Ensure that Session type is **Stateless** and Transaction Type is **Container**. By specifying Container as the Transaction Type, you enable the container to manage transactions so that you do not have to code transaction mechanisms. You should change this only if you want to write your own transaction mechanisms.

   Ensure that the check box next to **Generate Session Facade Methods** is selected.

   Leave the other values on this page at their defaults. Then click **Next**.

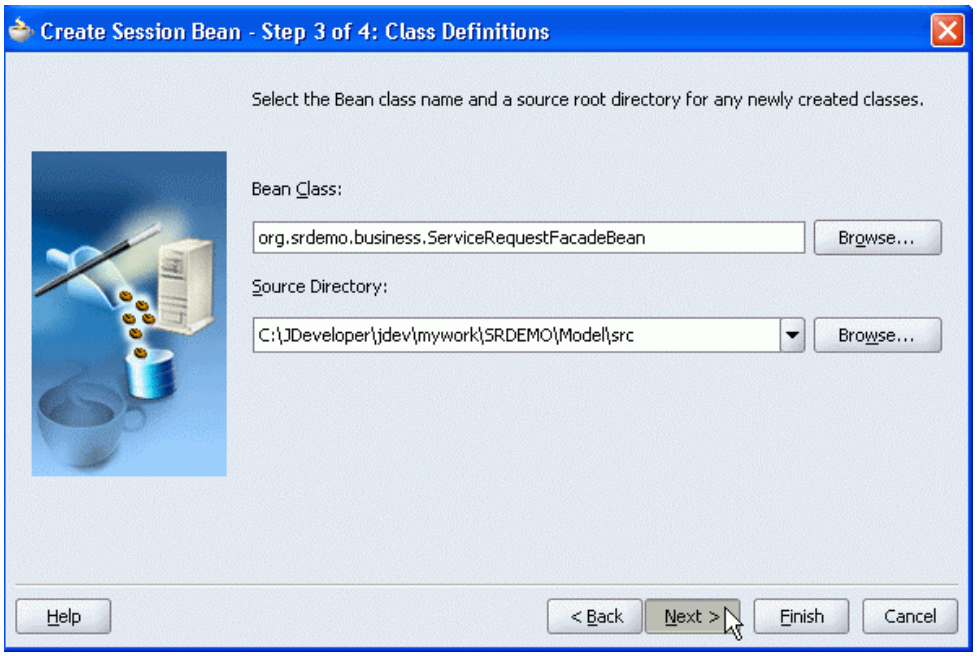
5. On the Select EJB 3.0 Entity Methods page of the wizard, clear all check boxes except the one next to **<Core Facade Methods>**. These are core CRUD methods to support transactions.

   Expand **Users** and select the check box next to **findUserByEmail**, as shown. Then click **Next**.

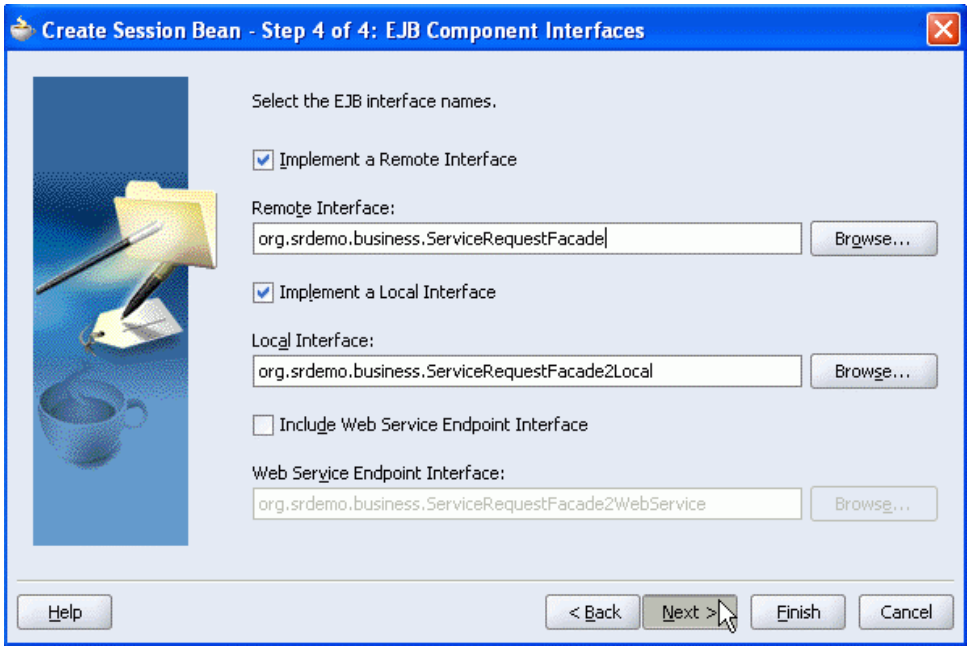


   Note: If you do not see the findUserByEmail named query, then cancel the Create Session Bean wizard, click **Rebuild** , and invoke the wizard again.

6. On the Class Definitions page of the wizard, change the Bean Class to **`org.srdemo.business.ServiceRequestFacadeBean`** (as shown in the screenshot), and then click **Next**.



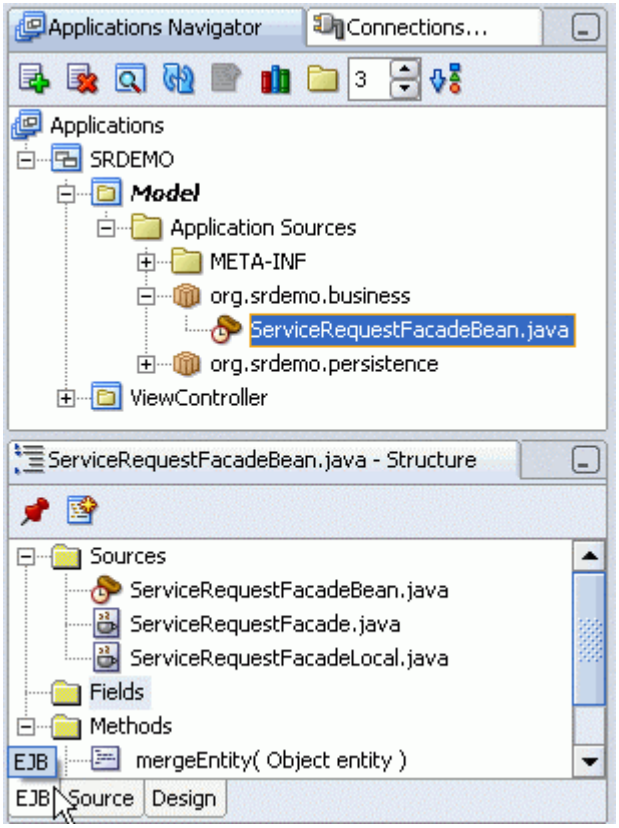
7. On the EJB Component Interfaces page, ensure that both **Implement a Remote Interface** and **Implement a Local Interface** are selected (as shown in the following screenshot) and then click **Next**.



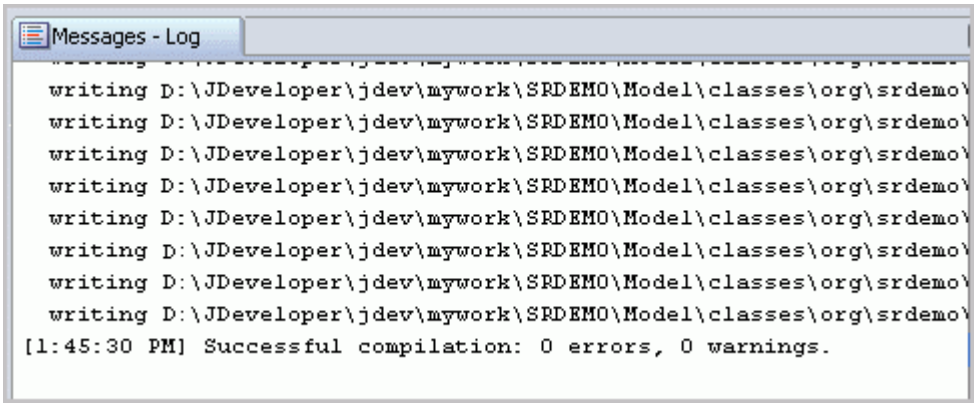
You implement the remote interface so that you will be able to use a command line test client. The local interface is needed for clients that run in the same Java Virtual Machine (JVM), such as JSPs.

8. On the Summary page, click **Finish**. The `org.srdemo.business` package is created.

9. In the Applications Navigator, expand the **org.srdemo.business** package and select **ServiceRequestFacadeBean.java**. In the Structure window, expand **Sources** to see the objects that have been created, which are shown as follows:
   - Session bean (`ServiceRequestFacadeBean.java`)
   - Remote interface (`ServiceRequestFacade.java`)
   - Local interface (`ServiceRequestFacadeLocal.java`)



10. Compile the project by selecting the **Model** node in the Applications Navigator and clicking **Rebuild** . The Messages Log window should display a message that compilation was successful, as shown in the following screenshot:

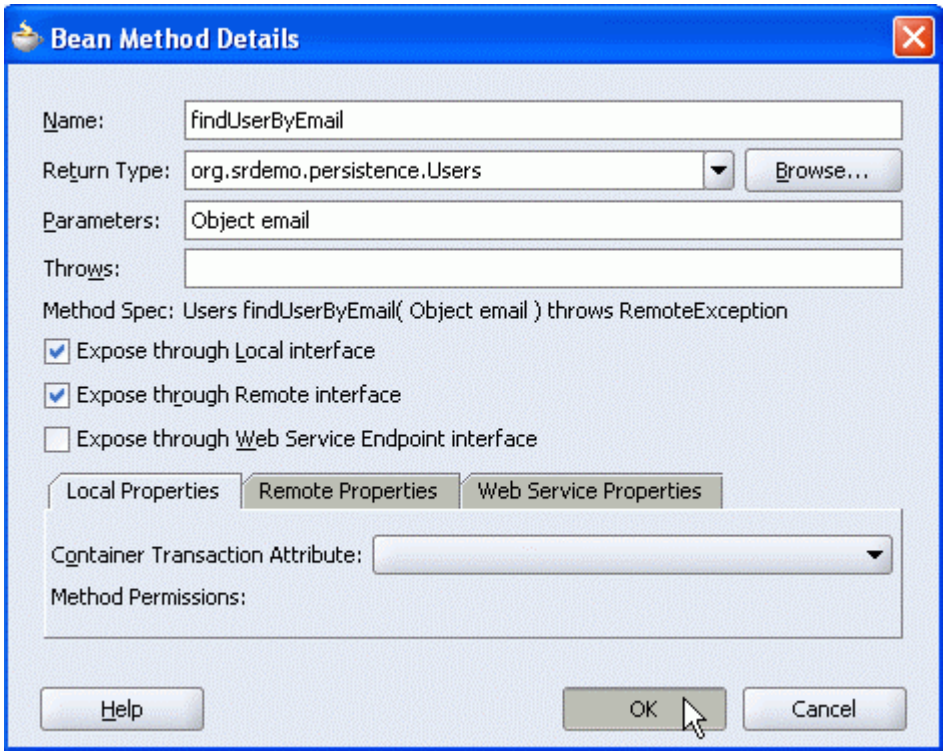


11. If it is not already open, double-click **ServiceRequestFacadeBean.java** to open it in the editor window. Scroll through the code to examine what has been created by default. You see methods to create and persist entity objects. There is also a method to implement the named query that you added to the Users entity object. Note that the `findUserByEMail()` method takes a parameter, as you specified when you defined the named query.

Modifying the Query Method to Return a Single Record

In the previous chapter, you defined a named query in the Users entity that can be used at run time to query a user by e-mail address. You included this named query when generating the session facade, which automatically generated a method to return a list of users. However, you need the method to return only one user. In this section, you modify the session facade bean and interfaces to return only one record, rather than a list of records, by performing the following steps:

1. Open **ServiceRequestFacadeBean.java** in the editor window, or click its tab if it is already open.
2. In the Structure window, right-click the **`findUserByEmail()`** method and select **Properties** from the context menu. When you set properties on the method, the bean and its local and remote interfaces pick up the changes and remain synchronized.
3. In the Bean Method Details dialog box, change the Return Type to **`org.srdemo.persistence.Users`**. Then click **OK**.



4. In the editor, scroll to the end of the file and make changes in the following lines:

```
public Users findUserByEmail(Object email) {
return
em.createNamedQuery("findUserByEmail").setParameter("email",
email).getResultList();
}
```

Change these lines to:

```
public Users findUserByEmail(Object email) {
return
(Users)em.createNamedQuery("findUserByEmail").setParameter("email
", email).getSingleResult();
}
```

Here are the applied changes:

```
/** <code>select object (user) from Users user where user.email = :email</code> */
public Users findUserByEmail(Object email) {
    return
    (Users)em.createNamedQuery("Users.findUserByEmail").setParameter("email",
    email).getSingleResult();
}
```

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

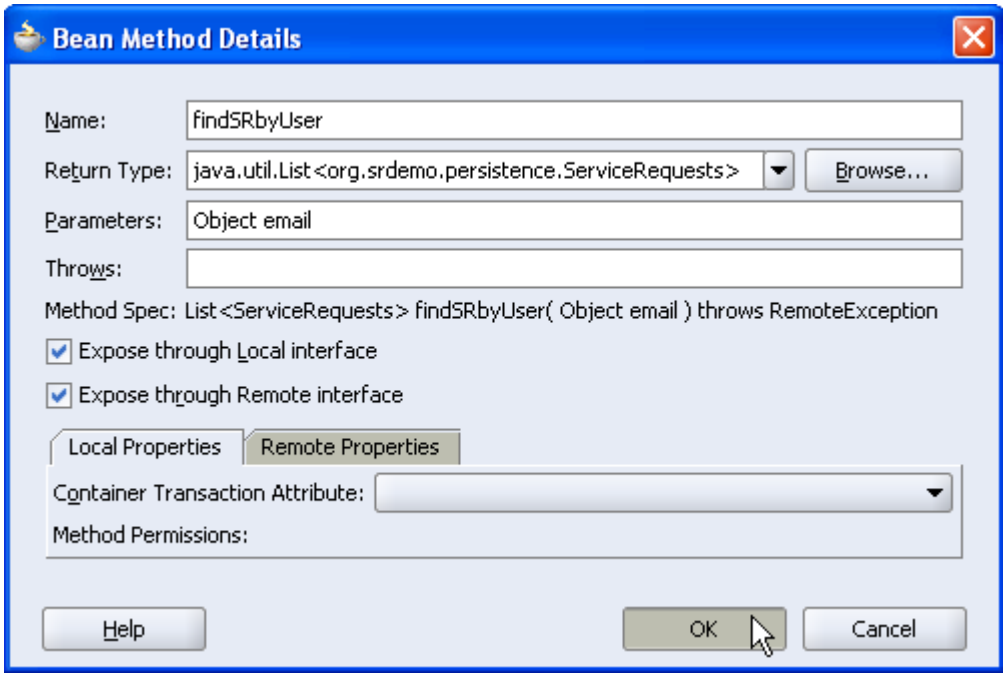
Adding a Method to Return Related Records

Now that the session facade calls a query to return the requested user by e-mail address, you need to add a method that returns a list of service requests that were created by that user. This is the last modification you make to the facade at this time. (As you continue to develop the application, more modifications are required.)

To add a method to return the related set of records, perform the following steps:

1. Open **ServiceRequestFacadeBean.java** in the editor, or click its tab if it is already open.
2. In the Structure window, right-click the **Methods** node and select **New Method** from the context menu. When you add methods in this way, the bean and its local and remote interfaces pick up the changes and remain synchronized.
3. In the Bean Method Details dialog box, enter the following values. Then click **OK**.

Field	Value
Name	`findSRbyUser`
Return Type	`java.util.List<org.srdemo.persistence.ServiceRequests>`
Parameters	`Object email`



4. In the editor, locate the method that you just added and change the line:

```
return null;
```

to

```
return findUserByEmail(email).getServiceRequestsList();
```

```
    public List<ServiceRequests> findSRbyUser(Object email) {
        return findUserByEmail(email).getServiceRequestsList();
    }
}
```

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

Summary

You have now developed session facade beans and interfaces. You performed the following key tasks:

- Used the Create Session Bean wizard to create an EJB 3.0 session bean, along with remote and local interfaces
- Modified the session bean's call to a named query to return a single Users object rather than a list of users
- Added a method to the session bean to return a list of related service requests that were created by the specified user

4

Deploying and Testing the Data Model

This chapter of the tutorial describes how to start OC4J and deploy the session bean that you created in the preceding chapter. It also shows you how to test the data model by creating and running a sample client that inserts records into the database.

If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh03.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

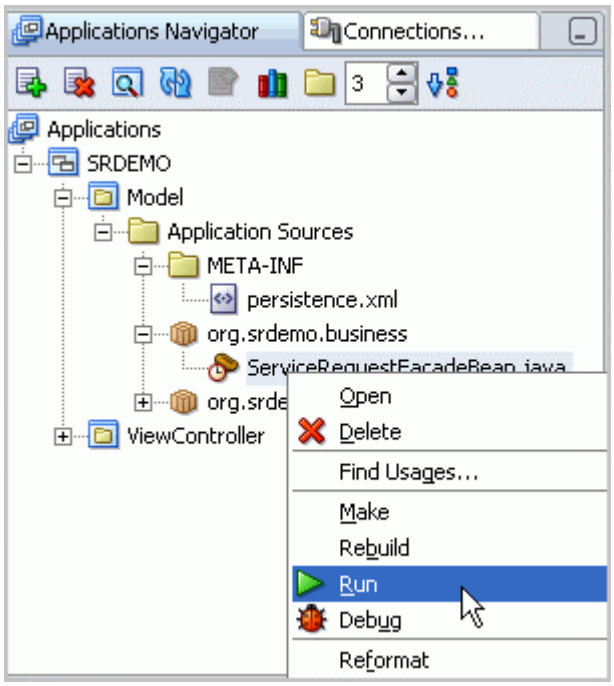
- Starting OC4J and Deploying the Application
- Adding a Named Query
- Creating a Sample Client
- Testing the Data Model
- Summary

Starting OC4J and Deploying the Application

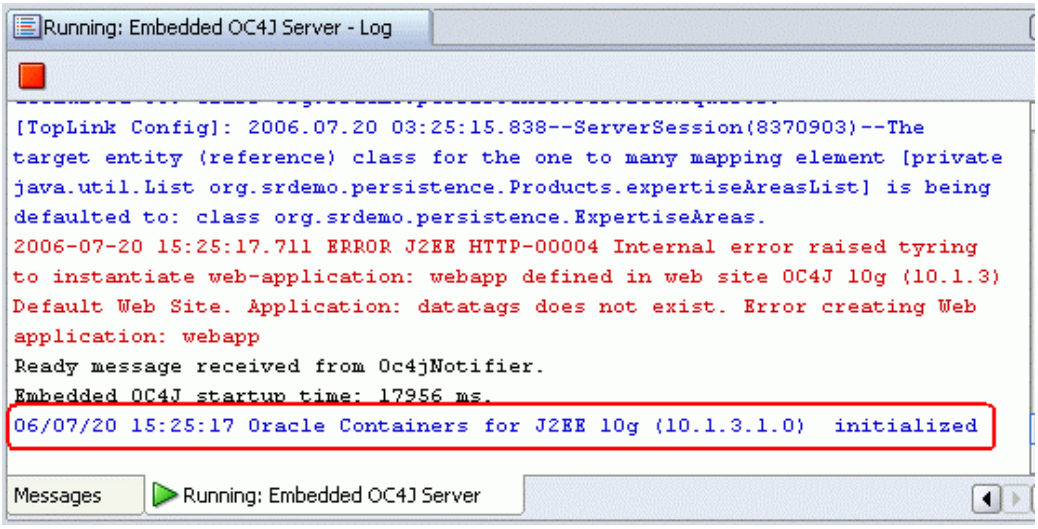
In the preceding chapter, you created a session facade bean and interfaces. In this section of the tutorial, you deploy the session bean to the embedded OC4J server that is included with JDeveloper. When you run the bean, it is deployed and runs in the OC4J server that is embedded in the JDeveloper IDE.

To start OC4J and deploy the application, perform the following steps:

1. In either the editor window or the Applications Navigator, right-click **ServiceRequestFacadeBean.java** and select **Run** from the context menu:



2. Be sure that the log window displays no errors and that OC4J is successfully initialized, as shown in the following screenshot. (If you have previously run the session bean, OC4J is reinitialized, so you do not receive the message about it being initialized.)



Adding a Named Query

You have now created the session facade bean, which provides the necessary services to communicate with database objects. Before you develop the View part of the application, however, you can test the Model to ensure that it is working correctly.

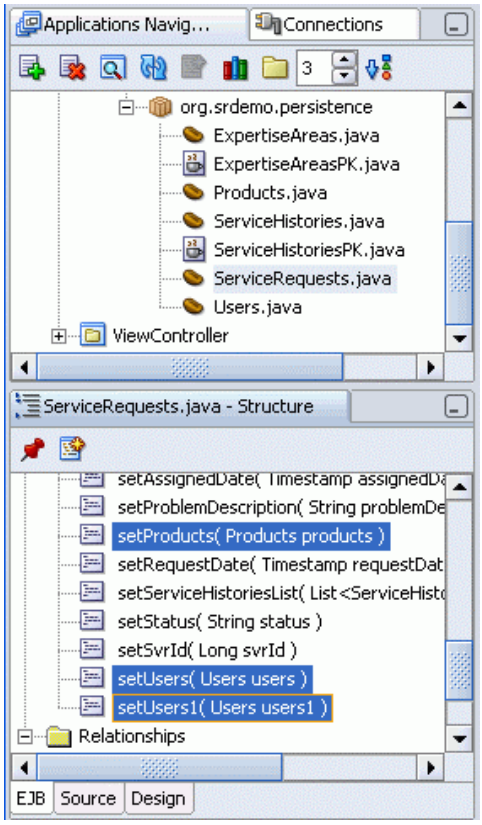
JDeveloper provides the ability to create a sample client class for testing the functionality of an EJB. You decide to test the application's ability to add a new service request to the database.

As you think about the code needed for creating a service request, you do the following:

- Examine `ServiceRequests.java`.
- Determine that another named query is needed.
- Add the named query to the entity bean.
- Modify the session bean.

To accomplish these tasks, perform the following steps:

1. In the Applications Navigator, select **ServiceRequests.java**.
2. In the Structure window, expand the **Methods** node and scroll down to examine the setter methods (as shown in the following screenshot). You can see that the methods **`setUsers()`**, **`setUsers1()`**, and **`setProducts()`** require that you pass objects to the methods.



3. In the Applications Navigator, select **ServiceRequestFacadeBean.java** and look at its methods in the Structure window. You can see the method to retrieve a `Users` object, `findUserByEmail()`, that was created to access the named query that you previously added to the `Users` object. However, there is no method to find a `Products` object. You decide to add a method to retrieve a `Products` object based on its ID so that you can pass that object to the `setProducts()` method when you create a new service request.

4. In the Applications Navigator, double-click **`Products.java`** to open it in the editor window.

5. Add the following named query, as shown in the screenshot below:

```
@NamedQuery(name="findProductById", query="select" +
" object (product) from Products product where" +
" product.prodId = :prodId")
```

Note: Because this file now contains multiple named queries, you must separate them with a comma and surround them with the **`@NamedQueries({ })`** annotation, pressing **[Alt+Enter]** when prompted to import `javax.persistence.NamedQueries`. If you are unsure how to do this, refer to Chapter 2 of this tutorial.

```
import javax.persistence.NamedQueries (Alt-Enter)
@NamedQueries({
@NamedQuery(name = "Products.findAll", query = "select o " +
"from Products o"),
@NamedQuery(name="findProductById", query="select" +
" object (product) from Products product where" +
" product.prodId = :prodId")
})
```

6. Now that you have added a named query to the entity, you must modify the session bean. First you must compile `Products.java` so that the added named query is available to the session bean.

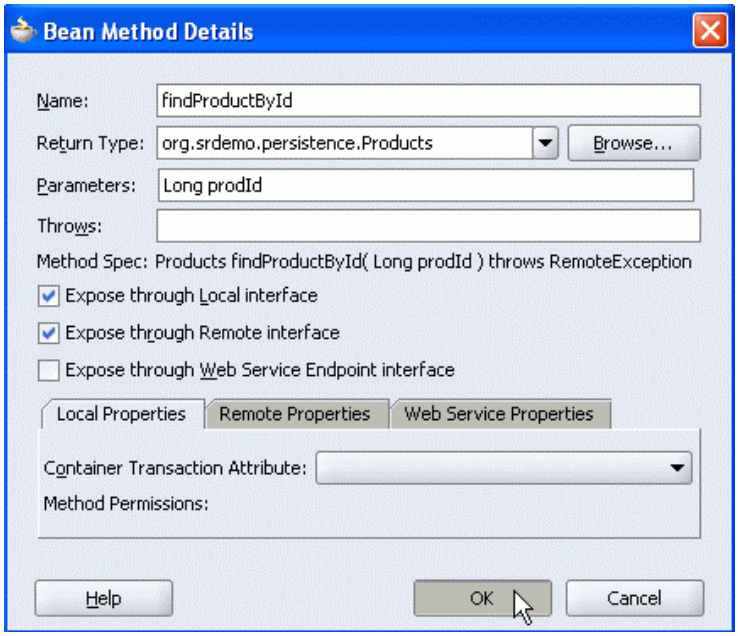
   Click **Make** .

7. In the Applications Navigator, right-click **ServiceRequestFacadeBean.java** and select **Edit Session Facade** from the context menu.

8. In the Session Façade Options dialog box, expand **Products**. Select the **findProductById** method and click **OK**. JDeveloper generates code in the session façade bean and interfaces to access the new named query.

9. In the Structure window, right-click the **findProductById** method and select **Properties** from the context menu. When you set properties on the method, the bean and its local and remote interfaces pick up the changes and remain synchronized.

10. In the Bean Method Details dialog box, change the Return Type to **`org.srdemo.persistence.Products`** and the Parameters to **`Long prodId`**, and then click **OK**.



11. In the editor, scroll to the end of the file and change the following line:

```
return
em.createNamedQuery("findProductById").setParameter("prodId",
prodId).getResultList();
```

Change this line to:

```
return
(Products)em.createNamedQuery("findProductById").setParameter
("prodId", prodId).getSingleResult();
```

Here are the changes:

```
public Products findProductById(Long prodId) {
    return (Products)em.createNamedQuery("Products.findProductById").setParameter
    ("prodId", prodId).getSingleResult();
}
```

12. Click **Save All** to save your work.
13. Now that you have modified the facade, redeploy it by right-clicking **ServiceRequestFacadeBean.java** in the Applications Navigator and then selecting **Run** from the context menu.

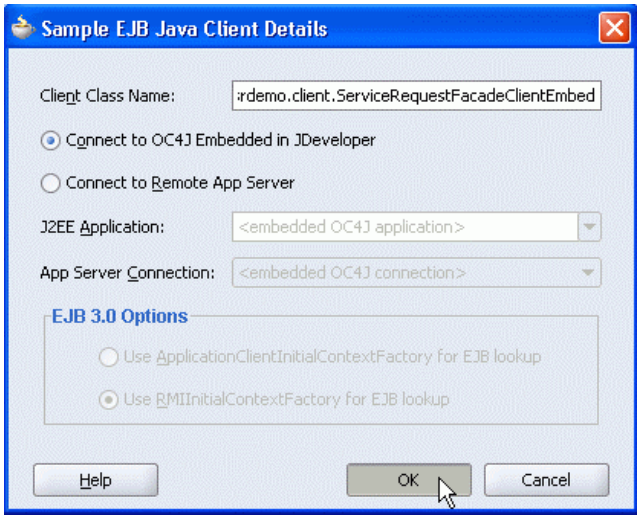
    Ensure that the Embedded OC4J Server Log window displays "`Ready message received from Oc4jNotifier`" along with a startup time.

Creating a Sample Client

To create a sample client, perform the following steps:

1. In the Applications Navigator, right-click **ServiceRequestFacadeBean.java** and select **New Sample Java Client** from the context menu.
2. In the Sample EJB Java Client Details dialog box, change the Client Class Name to **`org.srdemo.client.ServiceRequestFacadeClientEmbed`**.

   Ensure that **Connect to OC4J Embedded in JDeveloper** is selected (as shown in the following screenshot), and then click **OK**.



3. The `ServiceRequestFacadeClientEmbed.java` file is created in the `org.srdemo.client` package and opens in the editor window. If you examine the code, you can see that the generated sample client simply gets a handle to the initial session context.
4. Next you modify the sample client so that it inserts a Service Request record and a Service Histories record and also displays informative messages as processing occurs. You add code to

create a new `ServiceRequest` object and set its `Status`, `RequestDate`, `Products`, `ProblemDescription`, and `Users` properties.

In the `main()` method, locate the following line:

```
ServiceRequestFacade serviceRequestFacade =
  (ServiceRequestFacade)context.lookup("ServiceRequestFacade");
```

After this line, add the following code:

```
System.out.println("Creating a service request");
ServiceRequests sr = new ServiceRequests();
System.out.println("setting the status");
sr.setStatus("Open");

System.out.println("setting the timestamp for request date");
Timestamp requestDate = new
   Timestamp(System.currentTimeMillis());
System.out.println(requestDate);
sr.setRequestDate(requestDate);

System.out.println("getting product object with id  = 100");
Products product =
  serviceRequestFacade.findProductById(new Long(100));

System.out.println("setting the product for the request");
sr.setProducts(product);

System.out.println("setting problem description");
sr.setProblemDescription("This is my first problem request");

System.out.println("need to retrieve user object");
Users user =
serviceRequestFacade.findUserByEmail("ghimuro");

System.out.println("setting the user");
sr.setUsers(user);
```

5. Press **[Alt]+[Enter]** when prompted to import:
   `org.srdemo.persistence.Products`
   `org.srdemo.persistence.ServiceHistories`
   `org.srdemo.persistence.ServiceRequests`
   `org.srdemo.persistence.Users`
   `java.sql.Timestamp`

6. Add code to create a `ServiceHistories` object and set its properties. Add the following code immediately after the code that you previously added:

```
System.out.println("create service history for notes");
ServiceHistories historyNotes = new ServiceHistories();
historyNotes.setNotes("This is my first problem request");
historyNotes.setLineNo(new Long (1));
historyNotes.setServiceRequests(sr);
historyNotes.setSvhType("Customer");
historyNotes.setSvhDate(requestDate);
historyNotes.setUsers(user);
```

7. Press **[Alt]+[Enter]** when prompted to import:
   `org.srdemo.persistence.ServiceHistories`

8. Now that you have created an instance of the `ServiceHistories` object, you can set the `serviceHistoriesCollection` property of the `ServiceRequests` object, first creating a collection to use to set this property. Add the following code just below the code that you previously added:

```
System.out.println("adding history to collection");
List<ServiceHistories> historyList = new
   ArrayList<ServiceHistories>();
historyList.add(historyNotes);
System.out.println("setting the history list in the " +
   " ServiceRequest object");
sr.setServiceHistoriesList(historyList);
```

9. Press **[Alt]+[Enter]** to import:
   `java.util.List`
   `java.util.ArrayList`

10. Call a method from the facade to create the service request in the database. Because of the `CascadeType` (`PERSIST`) that you set on the `ServiceRequest` entity in a previous chapter of this tutorial, the `persist()` operation is cascaded to the associated `ServiceHistories` entity. Add the following code just below the code that you previously added:

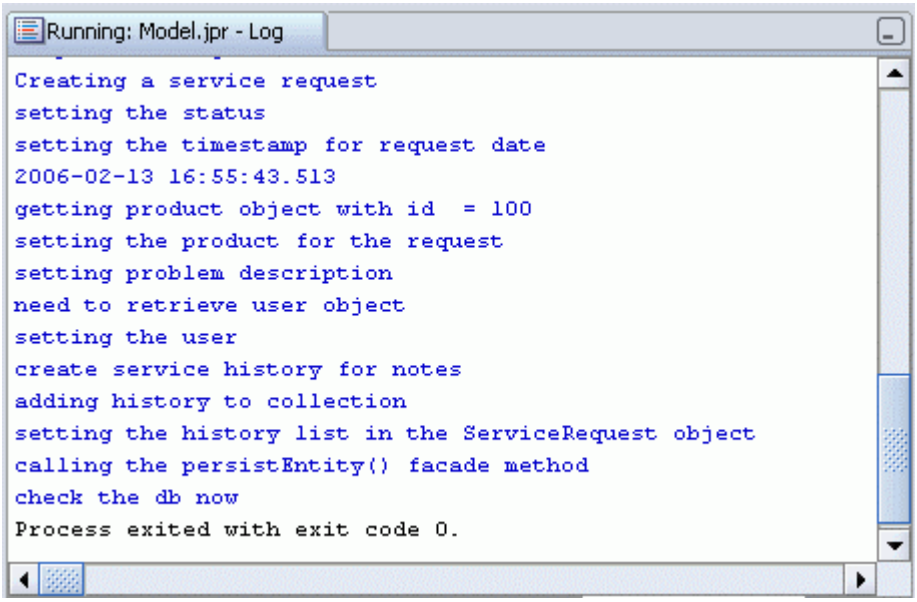
```
System.out.println("calling the persistEntity() facade " +
      "method");
serviceRequestFacade.persistEntity(sr);
System.out.println("check the db now");
```

11. Click **Save All** to save your work.

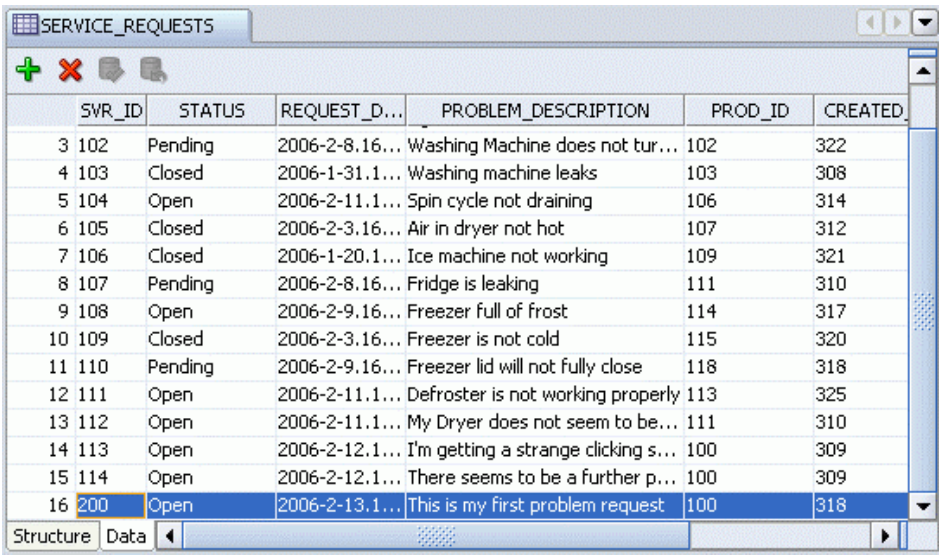
Testing the Data Model

Now that you have built the sample client, you are ready to test the data model. First you run the EJB 3.0 client, and then you check the database to see that the records were inserted.

1. In the Applications Navigator or in the editor window, right-click **ServiceRequestFacadeClientEmbed.java** and select **Run** from the context menu. The log window should show the messages that you coded into the client, as displayed here:



2. Check the database to see that records were inserted. Click the **Connections Navigator** tab to view the Connections Navigator.
3. In the Connections Navigator, expand **Database**, **SRDemo**, **SRDEMO**, and **Tables**.
4. Double-click the **`SERVICE_REQUESTS`** table to open it in the editor.
5. In the editor, click the **Data** tab. You should see the record that has been added, as shown in the following screenshot (the record may not be the last one in the list).



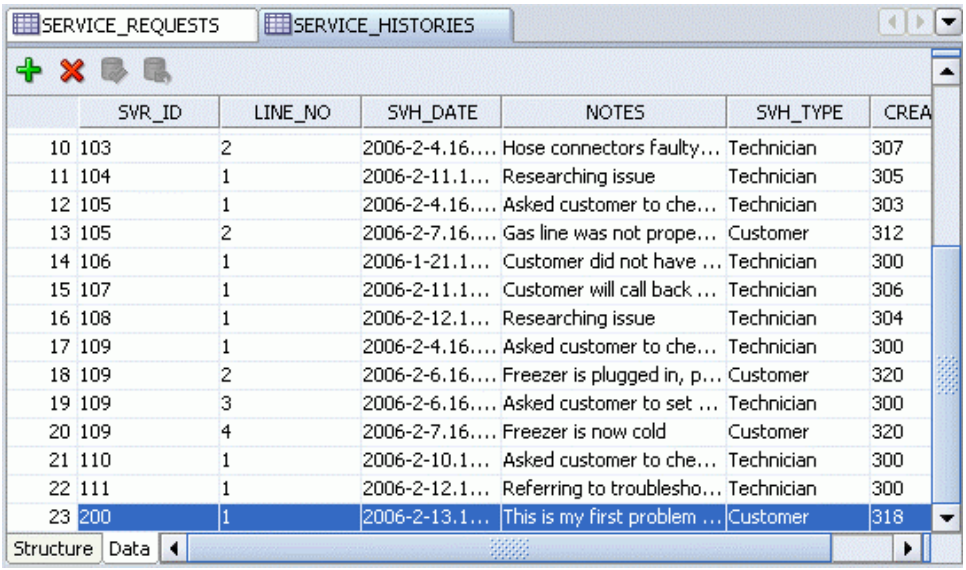
SERVICE_REQUESTS

	SVR_ID	STATUS	REQUEST_D...	PROBLEM_DESCRIPTION	PROD_ID	CREATED_
3	102	Pending	2006-2-8.16...	Washing Machine does not tur...	102	322
4	103	Closed	2006-1-31.1...	Washing machine leaks	103	308
5	104	Open	2006-2-11.1...	Spin cycle not draining	106	314
6	105	Closed	2006-2-3.16...	Air in dryer not hot	107	312
7	106	Closed	2006-1-20.1...	Ice machine not working	109	321
8	107	Pending	2006-2-8.16...	Fridge is leaking	111	310
9	108	Open	2006-2-9.16...	Freezer full of frost	114	317
10	109	Closed	2006-2-3.16...	Freezer is not cold	115	320
11	110	Pending	2006-2-9.16...	Freezer lid will not fully close	118	318
12	111	Open	2006-2-11.1...	Defroster is not working properly	113	325
13	112	Open	2006-2-11.1...	My Dryer does not seem to be...	111	310
14	113	Open	2006-2-12.1...	I'm getting a strange clicking s...	100	309
15	114	Open	2006-2-12.1...	There seems to be a further p...	100	309
16	200	Open	2006-2-13.1...	This is my first problem request	100	318

Structure Data

Note that `SVR_ID` was generated by a database sequence, as you specified in the `ServiceRequest` entity bean.

6. In a similar fashion, examine the database records for the `SERVICE_HISTORIES` table. You should see the new record that was inserted, as shown in the following screenshot (again, it may not be last in the list):



SERVICE_REQUESTS SERVICE_HISTORIES

	SVR_ID	LINE_NO	SVH_DATE	NOTES	SVH_TYPE	CREA
10	103	2	2006-2-4.16....	Hose connectors faulty...	Technician	307
11	104	1	2006-2-11.1...	Researching issue	Technician	305
12	105	1	2006-2-4.16....	Asked customer to che...	Technician	303
13	105	2	2006-2-7.16....	Gas line was not prope...	Customer	312
14	106	1	2006-1-21.1...	Customer did not have ...	Technician	300
15	107	1	2006-2-11.1...	Customer will call back ...	Technician	306
16	108	1	2006-2-12.1...	Researching issue	Technician	304
17	109	1	2006-2-4.16....	Asked customer to che...	Technician	300
18	109	2	2006-2-6.16....	Freezer is plugged in, p...	Customer	320
19	109	3	2006-2-6.16....	Asked customer to set ...	Technician	300
20	109	4	2006-2-7.16....	Freezer is now cold	Customer	320
21	110	1	2006-2-10.1...	Asked customer to che...	Technician	300
22	111	1	2006-2-12.1...	Referring to troublesho...	Technician	300
23	200	1	2006-2-13.1...	This is my first problem ...	Customer	318

Structure Data

Summary

You have now deployed and tested the application. You performed the following key tasks:

- Ran the session bean, which automatically started the embedded OC4J server
- Created a sample EJB Java client and modified it to insert data into the database
- Tested the data model by running the sample client and checking the database to see that records were inserted

5

Planning the User Interface

In this chapter of the tutorial, you plan the user interface for the application.

If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh04.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

- Planning Pages
- Specifying Navigation Rules
- Planning a Standard Look and Feel
- Summary

Planning the Pages

Background

So far in this tutorial, you have defined the data model for the application. The Model project that you have been working on provides the business model and persistence services that the application requires.

You first developed entity beans that represent the five tables in the `SRDEMO` database schema. You added named query annotations to the beans and made other modifications, such as:

- Specifying a cascade type for master-detail entities
- Using a database sequence to generate IDs

You then created a session facade bean and interfaces. You modified the files that were generated by the Create Session Bean wizard in order to:

- Return a single `Users` record rather than the list of users that is returned by default
- Return all service requests related to a particular user

You deployed the session bean to JDeveloper's embedded OC4J. You then created a sample client to test the application. You added functionality to the default sample client so that it would insert a record into two of the tables, using the ID generation strategy that you specified for the entity bean. After running the sample client, you examined the database tables to determine that the insertion of new records was successful.

Overview

Now that the data model has been implemented, you can begin to develop the user interface for the application. So far you have been working in the Model project; now you begin to work on the other project that you set up initially.

You can use a JSF Navigation (page flow) Diagram to plan the pages that the application should have and the navigation rules that define how users navigate between pages. When you add the JSF technology scope to your project, JDeveloper creates an empty navigation diagram, `faces-config.xml`, for you.

Although you can create pages without using this diagram, it is helpful to plan the pages and navigation right from the start. You can design the entire application and then add pages as needed.

In this section of the tutorial, you create the pages for the application and then you specify navigation rules between the pages.

Note: If you already have existing pages, you can add them to the diagram by dragging them from the Applications Navigator.

Using the JSF Navigation Diagram

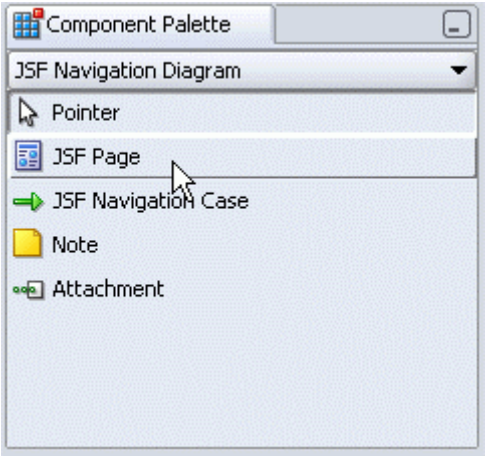
In this tutorial, you create the pages directly on the JSF Navigation Diagram by performing the following steps:

1. Right-click the **ViewController** project and select **Open JSF Navigation** from the context menu.

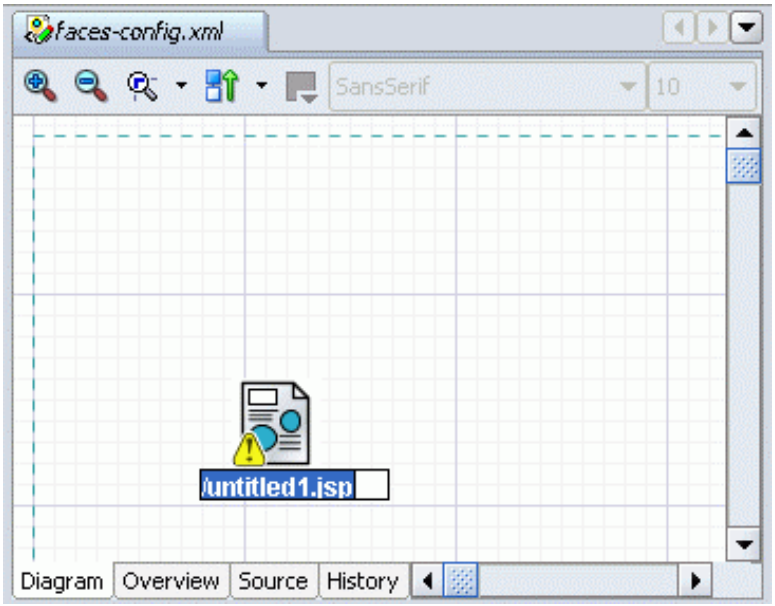
   The `faces-config.xml` file, which is visually depicted as a blank JSF Navigation Diagram, opens in the editor. A Component Palette and a Property Inspector are displayed at the right side of the IDE.

   Note: If you do not see the blank diagram, click the Diagram tab at the bottom of the editor. If you do not see the Component Palette or the Property Inspector, you can open them from the View menu.

2. To assist in laying out the pages in the diagram, you can display grid lines. Right-click in the diagram and select **Visual Properties** from the context menu. Select the **Show Grid** check box and click **OK**.
3. In the Component Palette, click **JSF Page**:



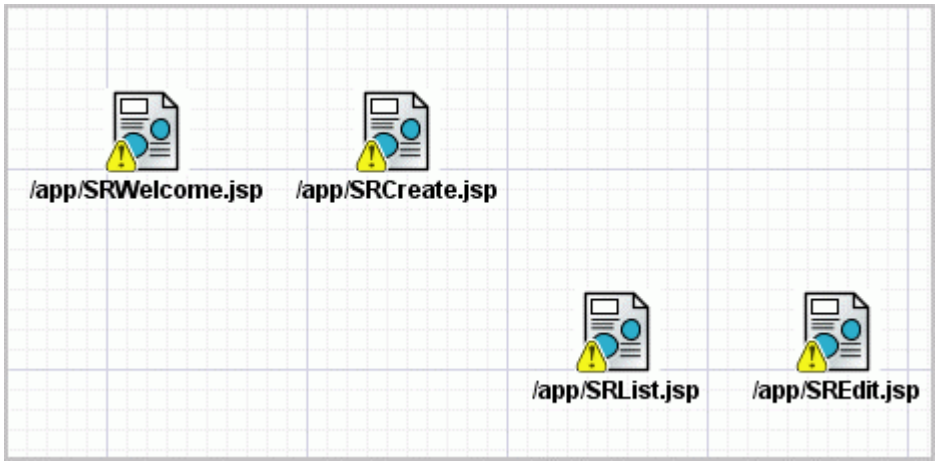
4. Click in the diagram where you want the page to appear. An icon representing a page appears on the diagram with the title **/untitled1.jsp**, as shown below. Notice that the icon has a yellow warning attached to it because you have not yet defined the page but merely created it on the diagram. In this chapter, you create the pages that you define in later chapters. After you define the page, the yellow warning disappears.



5. Click the icon label and change its name to **`/app/SRWelcome.jsp`** (the leading forward slash and the `.jsp` extension are parts of the name; if you omit them, JDeveloper automatically adds them). This is the first page that all users see when they invoke the application.
6. Add the following additional JSP pages, arranged as shown in the screenshot:

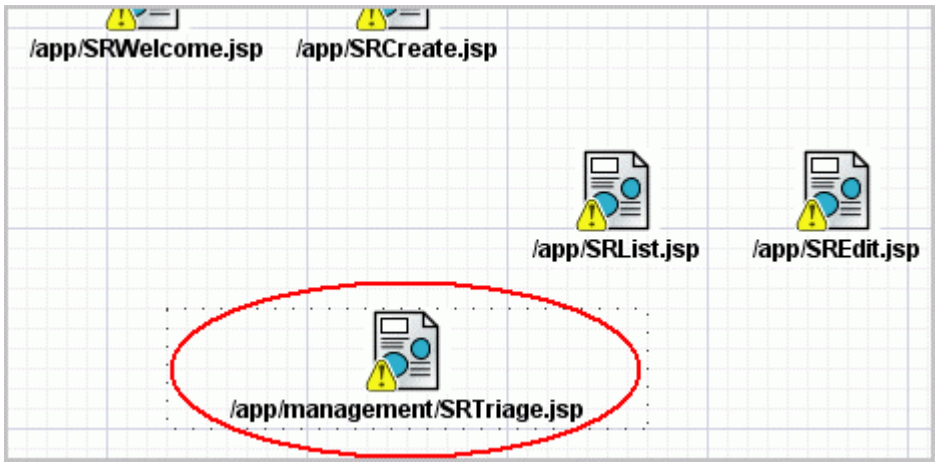
```
/app/SRCreate.jsp
/app/SRList.jsp
/app/SREdit.jsp
```

These are pages that can be accessed by all users of the application (customers, managers, and technicians).



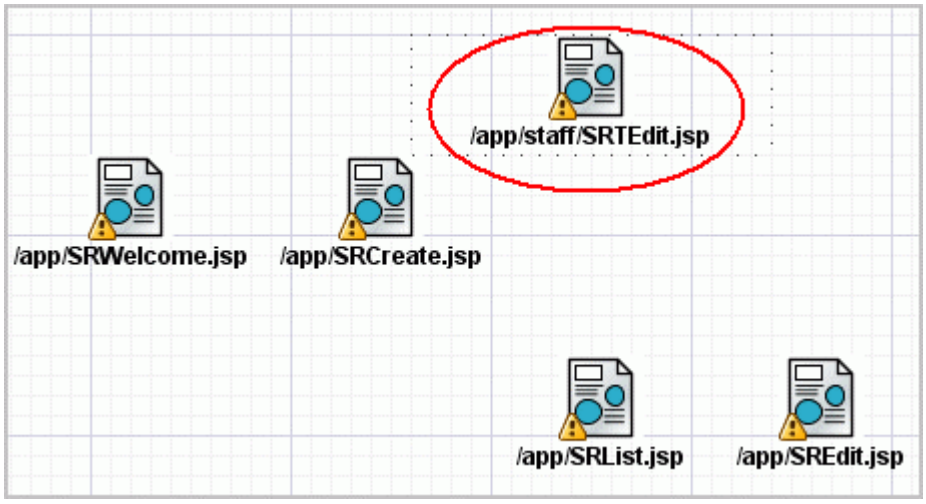
7. Add the following page that only managers can access (arranged as shown in the screenshot):

```
/app/management/SRTriage.jsp
```



8. Add the page that can be accessed by all employees (managers and technicians), arranged as shown in the following screenshot:

```
/app/staff/SRTEdit.jsp
```



9. Click **Save All** to save your work.

Specifying Navigation Rules

Now that you have created the pages of the application, you need to define how a user navigates among them.

With central control of navigation, you can easily define and change how users navigate the pages. This is part of the Model-View-Controller (MVC) design pattern, a best practice that JDeveloper enables you to easily implement in your application. The `faces-config.xml` file can define the control of navigation so that you avoid hard-coded navigation in the pages themselves.

JSF navigation is defined by a set of rules for choosing the next page to be displayed when a user clicks a UI component. There may be several ways in which a user can navigate from one page, and each of these would be represented by a different navigation case within the rule for that page. You can define these rules by using the JSF Navigation Diagram

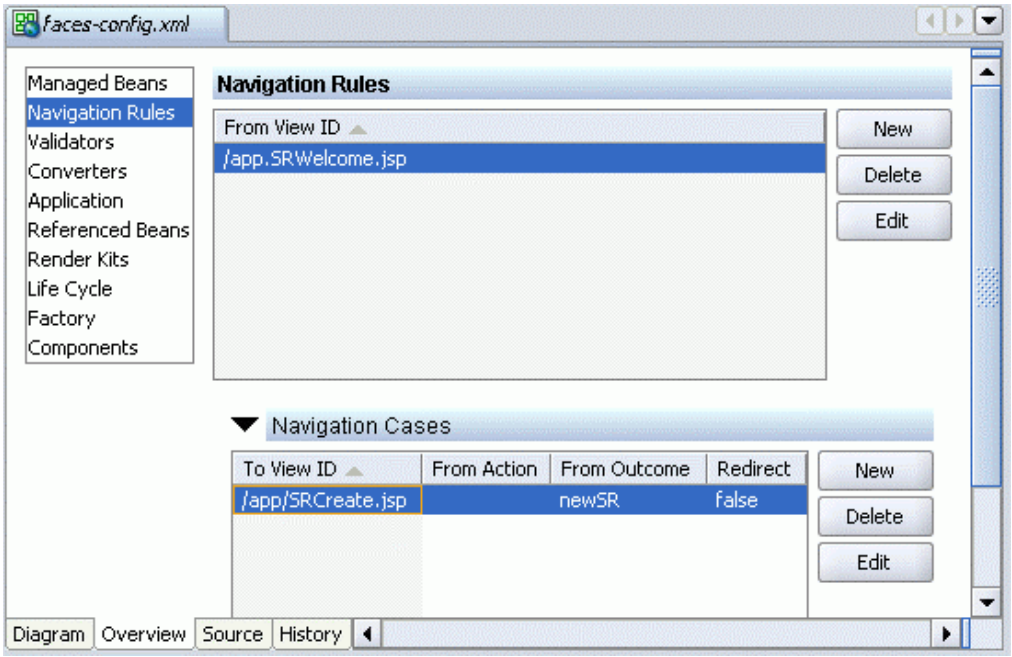
For the SRDemo application, you draw navigation cases on the diagram by performing the following steps:

1. In the Component Palette, click **JSF Navigation Case**.
2. In the diagram, first click the icon for the source page, **`/app/SRWelcome.jsp`**, and then click the icon for the destination page, **`/app/SRCreate.jsp`**.

   This creates a navigation case, shown as a solid arrow on the diagram from the source to the destination. A default `<from-outcome>` value of **`- success`** is the label for the navigation case, as shown here:

3. Modify the label by clicking it and typing **`newSR`** over the default label. Alternatively, you can change the label by modifying the **From Outcome** property in the Property Inspector. Note that JDeveloper adds a preceding hyphen and space automatically on the diagram.
4. Now that you have created a navigation case, examine the XML code that has been generated. Click the **Overview** tab at the bottom of the editor, and then click **Navigation Rules** in the list at the left of the window. The rule that you just created in the diagram is listed in the Navigation Rules box, and the navigation case is listed in the Navigation Cases box, as shown

in the following screenshot.



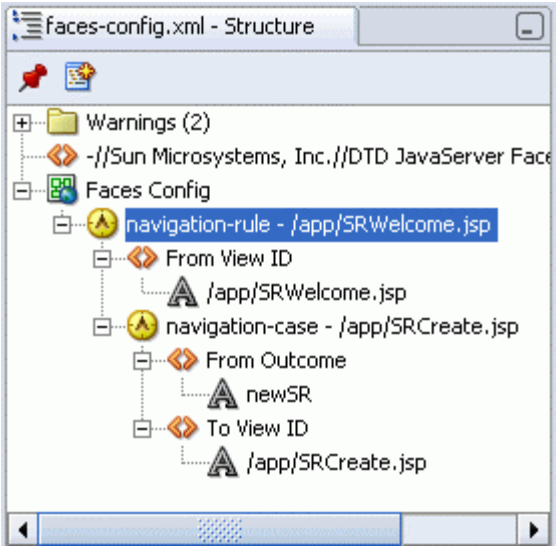
5. Click the **Source** tab to see the XML code for the selected rule.

   The `<from-view-id>` tag identifies the source page, and the `<to-view-id>` tag identifies the destination page. The wavy lines under the page names remind you that the pages have not yet been fully defined.

```
<navigation-rule>
  <from-view-id>/app/SRWelcome.jsp</from-view-id>
  <navigation-case>
    <from-outcome>newSR</from-outcome>
    <to-view-id>/app/SRCreate.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

6. You can also see the navigation rule in the Structure window, located in the lower left of the JDeveloper window below the Applications Navigator, as shown in the following screenshot. Click the **Source** tab.

   Note: If you do not see the Structure window, select View > Structure from the menu.



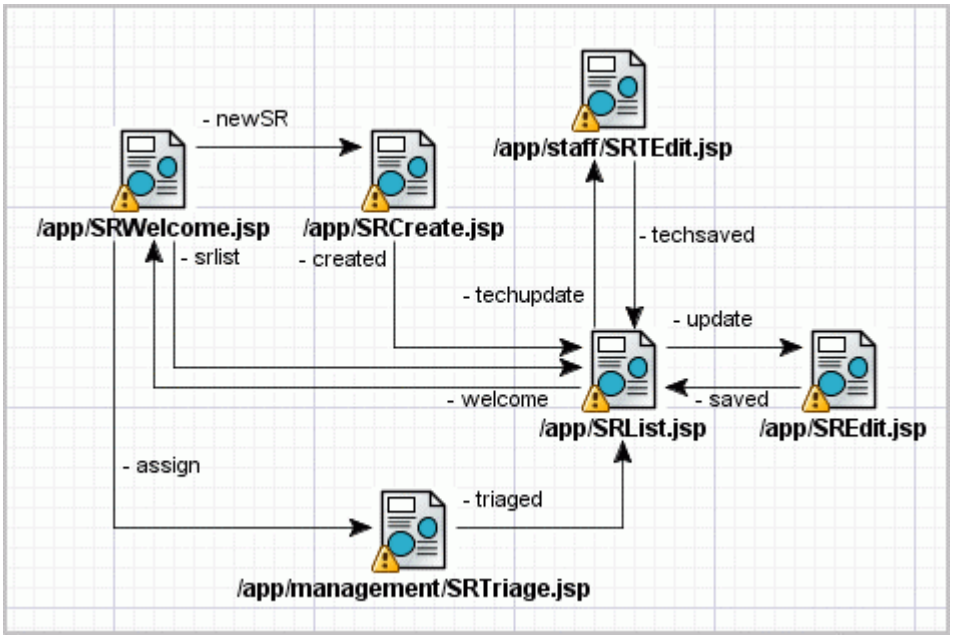
7. Repeat the steps to create more navigation cases on your diagram. The following table shows how users can navigate between pages. For example, a user can click a link on the SRWelcome page to navigate to the SRList page.

   To make a bend in the line on the diagram, click the intermediate point where you want the line to bend before clicking the destination page. To bend an existing line, shift-click the line where you want it to bend and drag the bend to the desired position.

   Note: For the label, you can simply type the word and JDeveloper adds a preceding hyphen and space automatically.

Source	Destination	Label
SRWelcome	SRTriage	`- assign`
SRWelcome	SRList	`- srlist`
SRCreate	SRList	`- created`
SRList	SRWelcome	`- welcome`
SRList	SRTEdit	`- techupdate`
SRList	SREdit	`- update`
SREdit	SRList	`- saved`
SRTEdit	SRList	`- techsaved`
SRTriage	SRList	`- triaged`

8. Click **Save All** to save your work. The diagram should be similar to the following screenshot.

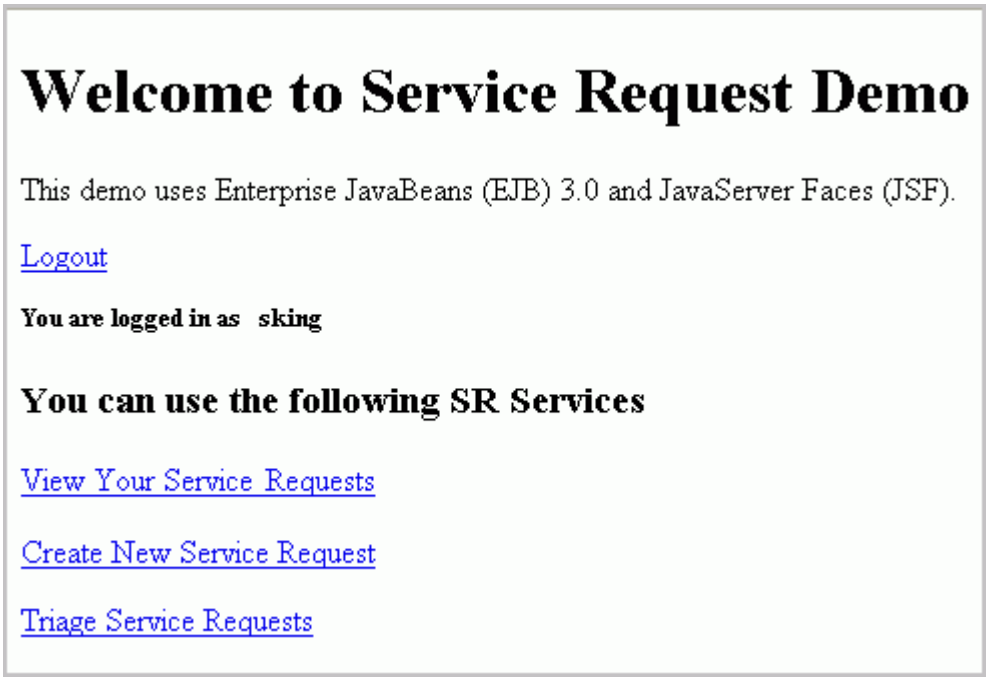


Planning a Standard Look and Feel

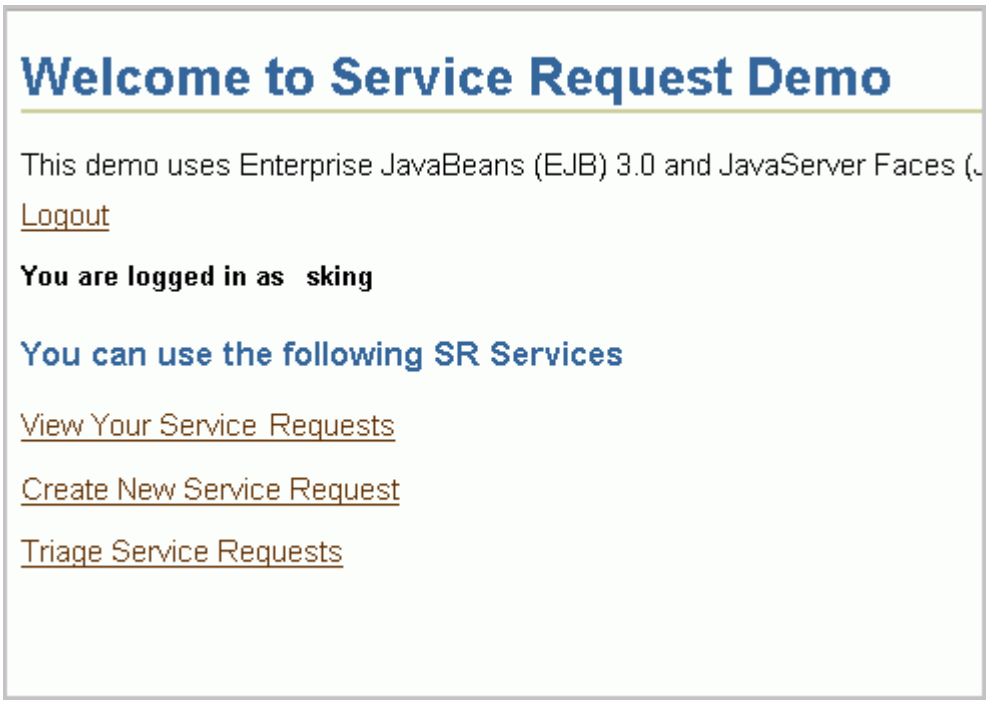
There are many ways to impose a standard look and feel across multiple pages. For example, you can define a template to govern the layout of the pages, or you can include a standard header and footer on each page.

A simple method that is built in to JDeveloper is to apply a style sheet to the pages. You can define your own style sheet, if desired, or use one that is included in the Component Palette by default.

You decide that the standard JDeveloper style sheet provides an acceptable look and feel for your application. For example, consider a page that looks like this:



Applying the JDeveloper style sheet changes the page to look like this:



To achieve a uniform look and feel, you instruct developers to use standard heading styles and apply the JDeveloper style sheet to all pages. You apply the style sheet when you create the actual pages in later chapters of this tutorial.

Summary

To plan the user interface for the application, you performed the following key tasks:

- Created pages in a JSF Navigation Diagram
- Defined navigation rules that specify how pages in the JSF Navigation Diagram are linked
- Planned to use the JDeveloper style sheet to apply a standard look and feel to pages

6

Defining Login Logic

Before creating pages, you decide to address how users log in to the application (authentication) and how they are allowed to access certain functionality (authorization). In this chapter, you define the security and login logic for the application.

If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh05.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

- Defining Security for the Application
- Creating Classes to Manage Roles
- Summary

Defining Security for the Application

You should implement both authentication and authorization in the environment where the application runs:

- **Authentication** determines which users can access the application. The Java Authentication and Authorization Service (JAAS) is a package that enables services to authenticate and enforce user-access controls. JAAS authentication is defined in the `jazn-data.xml` configuration file, which is referenced by the `jazn.xml` file. In the `jazn-data.xml` file, you can specify user IDs and passwords, create roles, and assign roles to users. In this application, the user IDs are the e-mail addresses of the users, and `welcome` is used for all of the passwords.
- **Authorization** determines what functions users are allowed to perform after they enter the application. You can control this by granting certain functionality to the roles in the Java EE deployment descriptor file, `web.xml`. There are three roles for this application—`user`, `technician`, and `manager`—and each user is assigned only one of these roles. You specify which pages and links are available to each role.

You first set up security for the container, defining users and roles. Then you set up application-specific access, defining which pages can be accessed by specific users.

Configuring Container Security

Java EE container-managed security enables you to configure Web applications to be accessible by authenticated and authorized users only. Container-managed security is easy to use and well integrated in the Java EE platform. It is the Java EE container's responsibility to ensure that all application requests from a specific user are executed within the user's security context.

Container-managed security can use Oracle Internet Directory (OID), LDAP, or a file-based security provider. The default installation uses the file-based provider.

Container security is implemented in the `jazn-data.xml` file. To set up container security, you specify the users who can log in and their passwords. You also define roles and assign them to users.

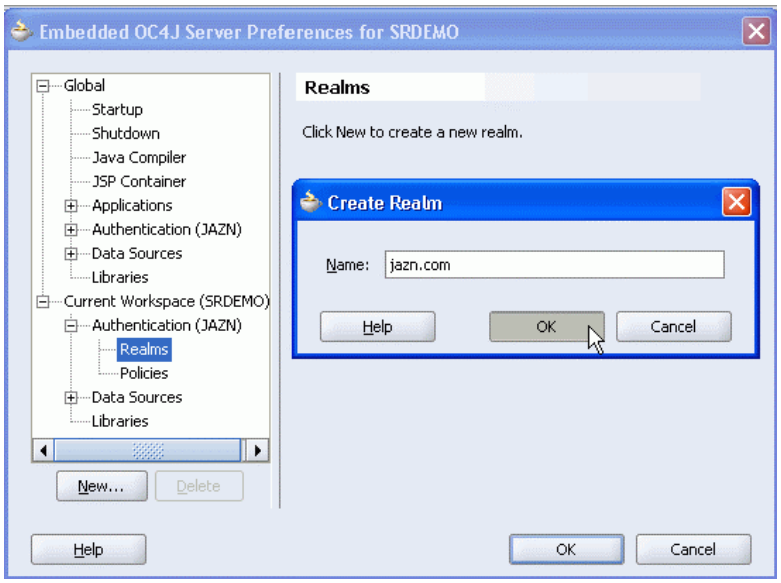
To add the allowed user ID, password, and role information to the `jazn-data.xml` file, perform the following steps:

1. If it is running, stop the OC4J server by selecting **Run > Terminate > Embedded OC4J Server** from the menu.
2. Select the **SRDEMO** application in the Applications Navigator.
3. From the menu, select **Tools > Embedded OC4J Server Preferences**. In JDeveloper 10.1.3, the embedded OC4J server uses the `jazn-data.xml` file that is contained in the current project, if any. If there is no `jazn-data.xml` file in the current project, the `jazn-data.xml` file for the application is used. This enables you to deploy the user and group settings to a production system or stand-alone OC4J for testing.
4. In the Embedded OC4J Server Preferences dialog box, expand **Current Workspace**, **Authentication**, and **Realms** from the tree at the left.

   A *realm* is the general component of security. It is a security domain that represents a security policy and a specific set of users. You can use the same realm for many different applications. Within a realm, applications participate in browser-based single sign-on, so that being authenticated in one application in a realm means that you do not have to be authenticated again in another application in the same realm, as long as you do not close the browser between requests.

   If there is already a `jazn.com` realm under the Realms node, do not create a new one. Instead, proceed to step 6.

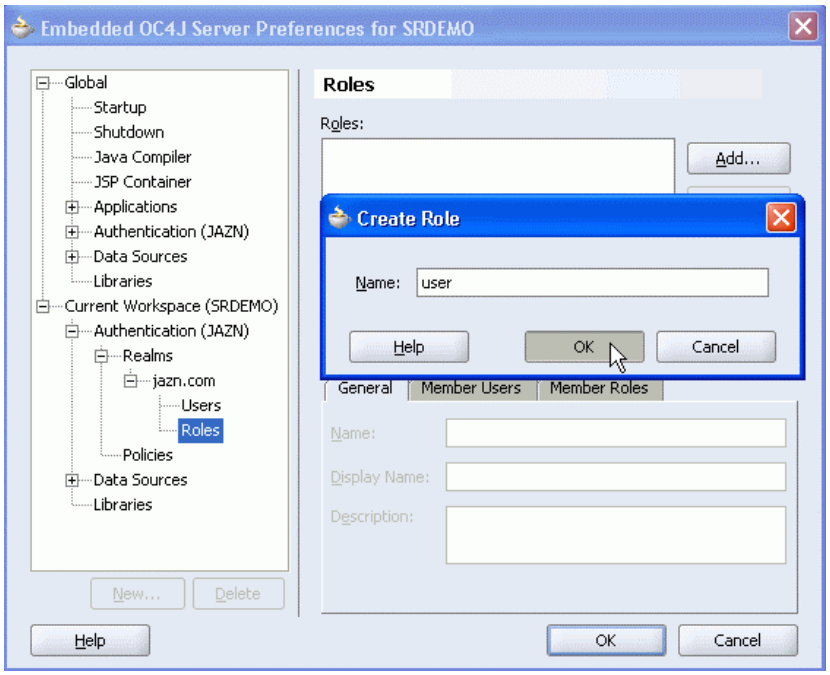
5. If there is no `jazn.com` realm under the Realms node, select the **Realms** node and click **New**. In the Create Realm dialog box, enter the name **`jazn.com`** and click **OK**, as shown in the following screenshot. The default realm in `jazn-data.xml` is `jazn.com`, but you can create your own default realm if desired.



6. Under the **jazn.com** node, select the **Users** node in the tree and click **Add**.
7. Add each of the following users, entering **`welcome`** for the Credentials of each:

   `ghimuro`
   `nkochhar`
   `bernst`
   `daustin`
   `sking`

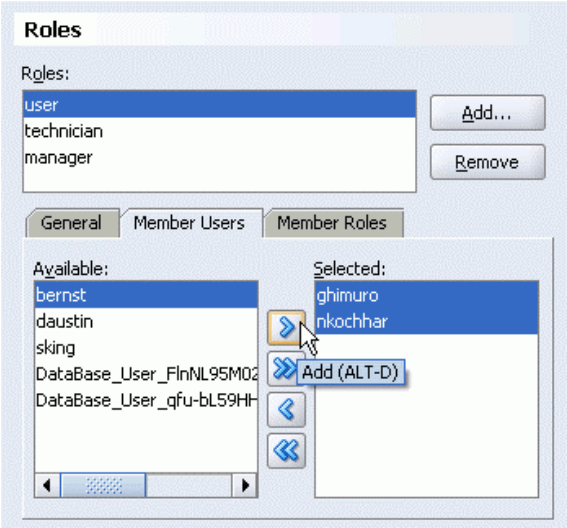
   These are not all of the users in the database, but they are enough to test the application. For production, you must add all users who should have access to the application. In reality, you would use OID or LDAP providers for production systems that have more than just a few users.
8. Select the **Roles** node in the tree (under the jazn.com node) and click **Add**.
9. In the Create Role dialog box, enter **`user`** and click **OK**.



10. Create two additional roles: **`technician`** and **`manager`**.
11. Select **user** in the Roles list and click the **Member Users** tab.

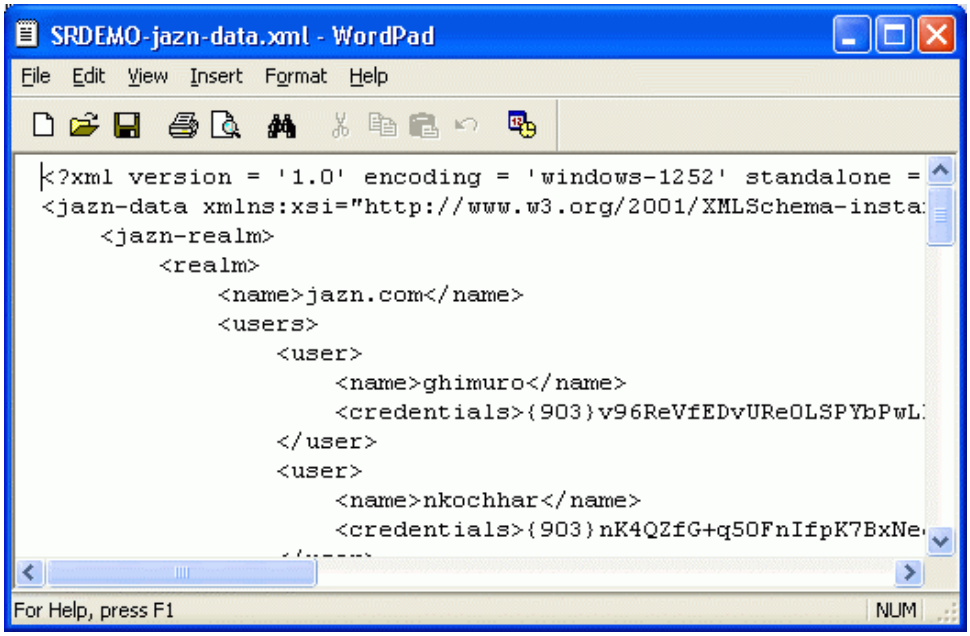
    Select the following users in the Available list and click Add to move them to the Selected list to grant the **`user`** role to them:

    `ghimuro`
    `nkochhar`



12. In a similar fashion, grant the **`manager`** role to **`sking`**. Grant the **`technician`** role to **`bernst`** and **`daustin`**.
13. Click **OK** to dismiss the Embedded OC4J Server Preferences dialog box.

14. All of the data that you entered is kept in a file at the root directory of the application. The convention used to name the file is `<applicationName>-jazn-data.xml`. In this tutorial, the file is named `SRDEMO-jazn-data.xml`. If desired, you can open this file in Wordpad to see the XML that you declaratively created. Passwords in the files are encrypted.



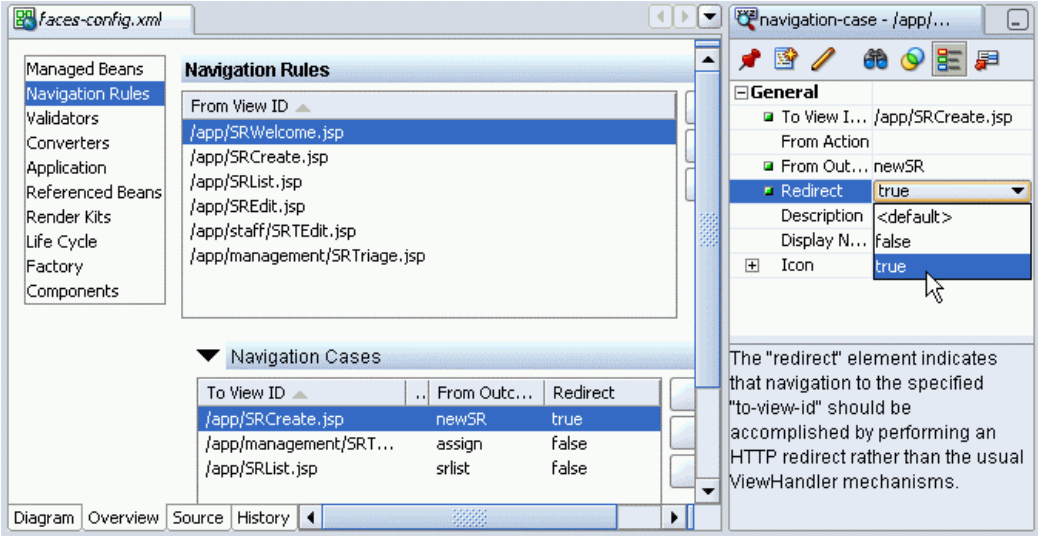
Note: As mentioned previously, the application-specific `jazn-data.xml` file is used only if there is `no jazn-data.xml` file contained in the project.

Enforcing Container-Managed Security

By default, JSF performs server-side navigation, which bypasses container-managed security. To enforce container-managed security for your application, you must set JSF navigation cases to use an HTTP redirect rather than the default ViewHandler mechanisms.

To set this declaratively, perform the following steps:

1. In the editor, open (right-click the **ViewController** project in Applications Navigator and select **Open JSF Navigation**) or switch to **`faces-config.xml`** if it is already open. Click the **Overview** tab.
2. In the list at the left, select the **Navigation Rules** node.
3. In the Navigation Rules list, select **/app/SRWelcome.jsp**.
4. In the Navigation Cases list, select the entry in the list that has **newSR** as the From Outcome.
5. In the Property Inspector, select **true** from the drop-down list for the Redirect property (as shown in the following screenshot).



6. Select each of the other two navigation cases in turn (those with **assign** and **srlist** From Outcomes) and set the Redirect property to **true** for each.
7. Select each of the other navigation rules from the upper list and set the Redirect property for each of its navigation cases to **true**.

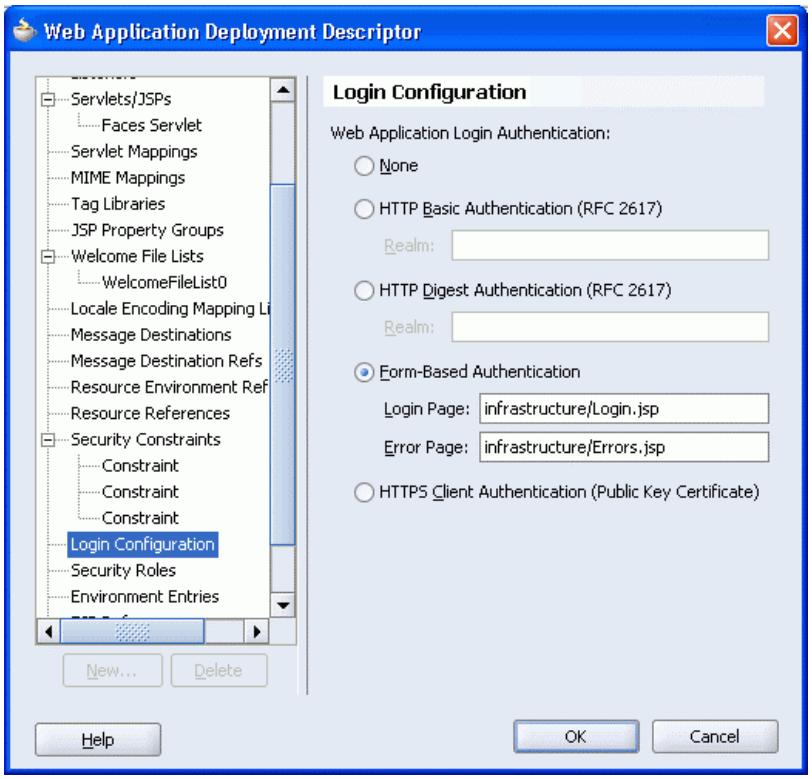
Configuring Application Access

Now that you have created users and roles and have granted roles to be used by the container, you are ready to define the functionality that should be available to each role for this particular application. Because you have separated the pages accessible to each role into different directories, you can use URL patterns to define the users who are authorized to access each set of pages.

To declaratively edit the Java EE deployment descriptor file where such authorization is defined, perform the following steps:

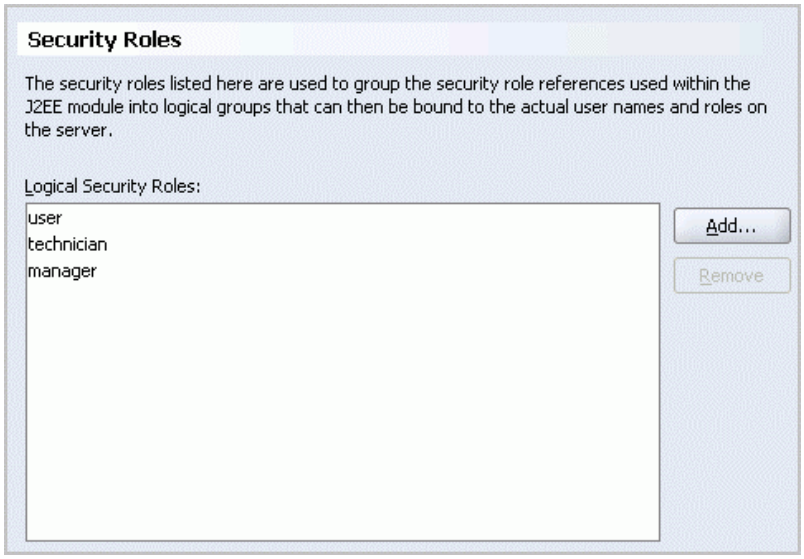
1. In the Applications Navigator, expand the **ViewController > Web Content > WEB-INF** nodes and right-click **web.xml.** Select **Properties** from the context menu.
2. In the tree at the left, select the **Login Configuration** node. In the Login Configuration section of the window, select the **Form-Based Authentication** option. Form-based authentication uses a custom Web application, such as a JSP form, to provide a log-on dialog box.

   Note: Form-based authentication stores the username + password pair in the session, but it is in clear text that is visible to the request object. If you use either form-based or basic authentication, you should use SSL for added security.
3. Set the Login Page to `infrastructure/Login.jsp` and set the Error Page to `infrastructure/Errors.jsp`. These are pages that you create in the next chapter.

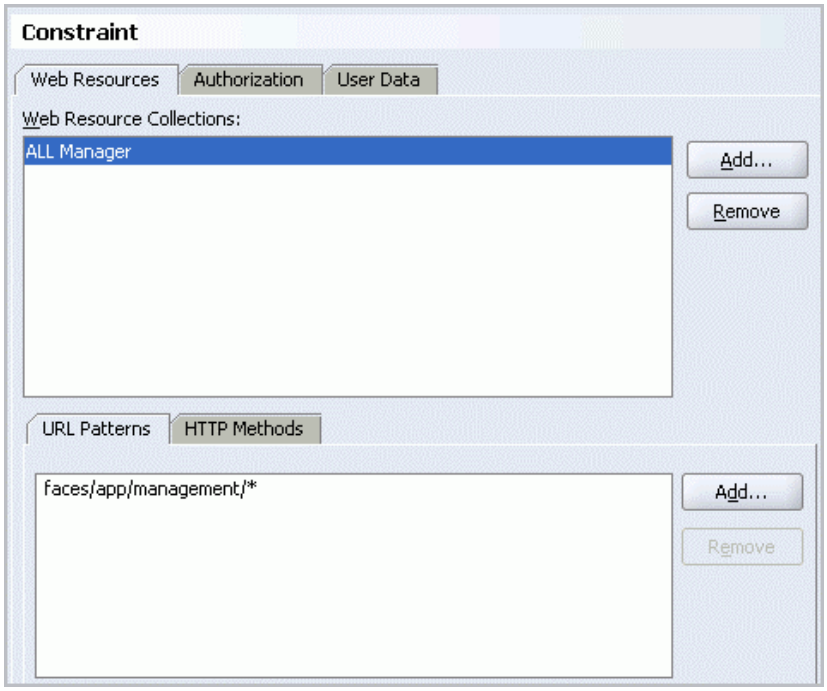


4. Select the **Security Roles** node in the tree at the left, and then click **Add**.
5. In the Create Security Role dialog box, enter **`user`** as the Security Role Name. Then click **OK**.
6. In a similar fashion, add the other two roles: **`technician`** and **`manager`**.

   Note: For authorization to be successful, you must use the same role names as the role names that you used when configuring the realm in OC4J preferences. If you choose different names, you need to define mappings in the `orion-web.xml` file to map the Java EE security role that you define in `web.xml` to the role name in OC4J.



7. Now that you have defined the security roles, they need to be assigned to security constraints to enforce the authorization. Select the **Security Constraints** node and click **New**.
8. On the Web Resources tab, click **Add** and specify **`ALL Manager`** as the Web Resource Name. Click **OK**.



9. With the **`ALL Manager`** collection selected, click the **Add** button in the lower section of the window, to the right side of the URL Patterns tab. In the Create URL Pattern dialog box, enter **`faces/app/management/*`** and click **OK**.
10. With the **`ALL Manager`** collection selected, click the **Authorization** tab and select the **manager** check box.
11. Create two more security constraints by again selecting **Security Constraints** in the tree at the left and then clicking **New**. Use the values in the following table:

Web Resource Collections	URL Patterns	Authorization
`AllStaff`	`faces/app/staff/*`	`technician` and `manager`
`SRDemo Sample`	`faces/app/*`	`user`, `technician`, and `manager`

12. There should be three constraint entries under the Security Constraints node. Click **OK** to dismiss the Web Application Deployment Descriptor dialog box.
13. In the Applications Navigator, double-click **web.xml** to open it in the editor. Notice that there is a Component Palette at the right of the editor. The Component Palette enables you to declaratively edit the `web.xml` file in the editor as an alternative to editing it from its Properties window. You can see that all of the entries after the `<jsp-config/>` line show the security constraints, login configuration, and security roles that you specified.

```
web.xml
        <mime-type>text/plain</mime-type>
    </mime-mapping>
    <jsp-config/>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>ALL Manager</web-resource-name>
            <url-pattern>faces/app/management/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>manager</role-name>
        </auth-constraint>
    </security-constraint>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>AllStaff</web-resource-name>
            <url-pattern>faces/app/staff/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>technician</role-name>
            <role-name>manager</role-name>
        </auth-constraint>
    </security-constraint>
```

14. With the **`web.xml`** file open in the editor, the Structure window also shows all of the elements that you specified.

```
web.xml - Structure
web-app - 2.4
    servlet - Faces Servlet
    servlet-mapping - Faces Servlet
    session-config
    mime-mapping - html
    mime-mapping - txt
    jsp-config
    security-constraint
        web-resource-collection - ALL Manager
        auth-constraint
    security-constraint
        web-resource-collection - AllStaff
        auth-constraint
    security-constraint
        web-resource-collection - SRDemo Sample
        auth-constraint
    login-config
        form-login-config
    security-role - user
    security-role - technician
    security-role - manager
```

15. Click **Save All** to save your work

Creating Classes to Manage Roles

Now that you have implemented security for the container, you need to be able to access user and role information from the application. To do so, you perform the following tasks:

- Create utility classes.
- Create a managed bean.

Creating Utility Classes

The key security artifact for this application is the `UserInfo` bean, which you define next. When this class is first referenced, it reads the container security attributes (such as the `remoteUser` value that matches the `email` column in the `Users` table) and makes the key information available for access via Expression Language (EL).

For example, `UserInfo` gets the logged-in user (`remoteUser`) from the Java EE environment and then finds the corresponding `Users` object, which is mapped to a row in the `Users` table. Because the `UserInfo` bean has session scope, the application always knows the logged-in user and is able to perform CRUD operations that take the `User` object.

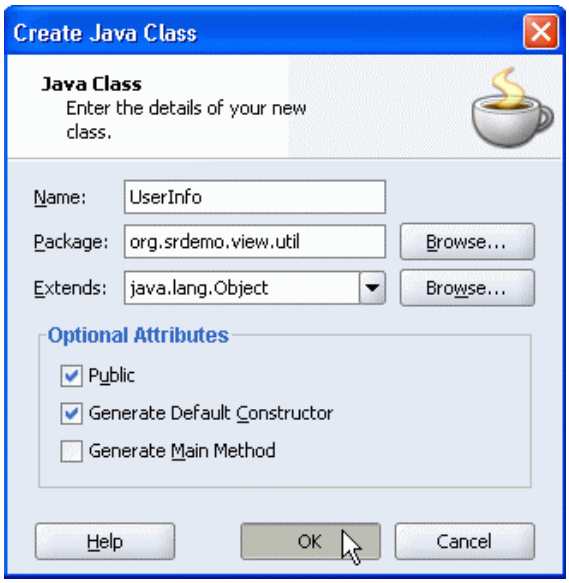
In this section, you create the `UserInfo` class to be used as a managed bean called `UserInfo`. The following table lists the EL expressions in the `UserInfo` class that can be used in the application:

Expression	Value
`#{UserInfo.userName}`	Returns the login ID or the string `Not Authenticated`
`#{UserInfo.userRole}`	Returns a string with the current user's role (for example, `manager`)
`#{UserInfo.staff}`	Returns `true` if the user has the `technician` or `manager` role
`#{UserInfo.customer}`	Returns `true` if the user has the `user` role
`#{UserInfo.technician}`	Returns `true` if the user has the `technician` role
`#{UserInfo.manager}`	Returns `true` if the user has the `manager` role

You also create another utility class, `ServiceLocator.java`, which is used to get a handle to session facade. The locator also caches instances of the facade so that it doesn't have to look up the bean again. Using the service locator class implements a core Java EE design pattern.

To create a set of utility classes to validate users and determine available roles, perform the following steps:

1. In the Applications Navigator, right-click the **ViewController** project and select **New** from the context menu.
2. In the New Gallery, select the **Java Class** item from the General category and click **OK**.
3. In the Create Java Class dialog box, set the Name of the class to **`UserInfo`** and the Package to **`org.srdemo.view.util`**. Click **OK**.



4. In a similar fashion, create in the same package an additional class named **`ServiceLocator`**.
5. From your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory) open the `ServiceLocatorJava.txt` file from the `\files` subdirectory. Copy all the text in this file and then replace the code in the **`ServiceLocator.java`** class with the copied text. You can then close `ServiceLocatorJava.txt`.
6. In a similar fashion, replace the code in the **`UserInfo.java`** class with the code that is contained in `\setup\files\UserInfoJava.txt`.
7. Click **Save All** to save your work.

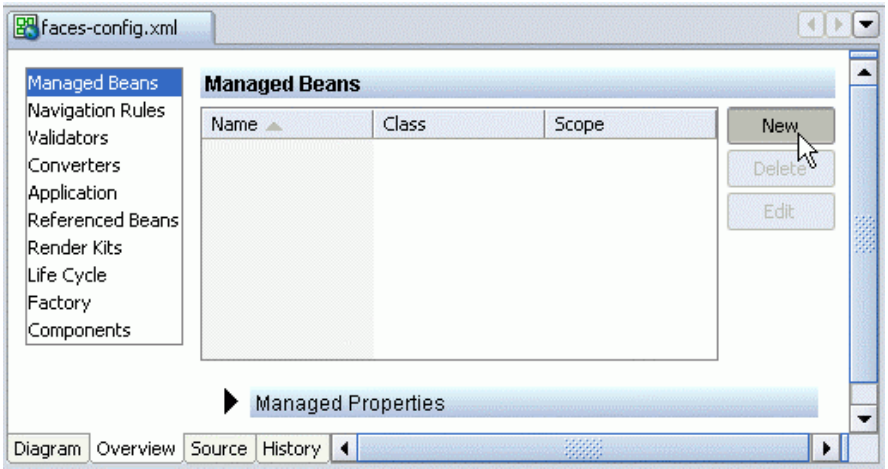
Creating a Managed Bean

Managed beans are objects that are defined in the `faces-config.xml` file. They can be POJOs (plain old java objects), lists, or maps. The classes that can be used as managed beans must have constructors with empty arguments.

The central configuration in the `faces-config.xml` file enables you to declare and initialize properties of all managed beans in one place while restricting the scope of a bean to application, session, or request. You can use JSF EL to access a managed bean and to initialize its properties. You can also change the bean class or its initial property values without changing the code in the bean.

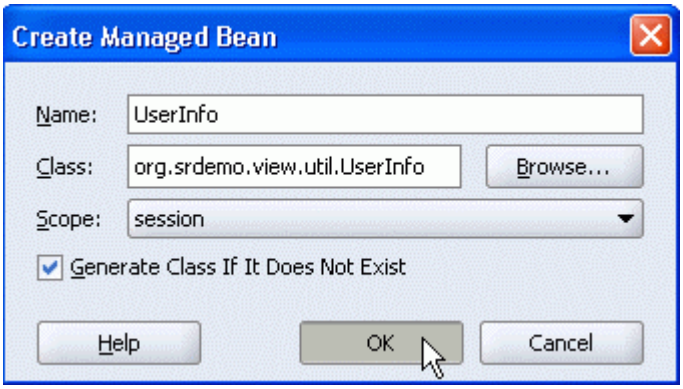
To create the `UserInfo` managed bean, perform the following steps:

1. Switch to the `faces-config.xml` file in the editor if it is already open. If it is not open, right-click the **ViewController** project and select **Open JSF Navigation** from the context menu.
2. Click the **Overview** tab.
3. Select **Managed Beans** in the list at the left and click **New**.



4. In the Create Managed Bean dialog box, set the property values to those in the following table:

Field	Value
Name	`UserInfo`
Class	`org.srdemo.view.util.UserInfo`
Scope	`session`
Generate Class If It Does Not Exist	Select the check box.



5. Click **OK**.
6. Click the **Source** tab. Near the end of the file, you can see the managed bean that you just defined.

```
<managed-bean>
  <managed-bean-name>UserInfo</managed-bean-name>
  <managed-bean-class>org.srdemo.view.util.UserInfo</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

7. Click **Save All** to save your work.

You have now added users and a bean to access user information so that the pages can run and be populated with data. In the next chapter, you create the pages that enable a user to log in and access the functionality designated for that user's role.

Summary

In this chapter, you provided security for the SRDemo application. To accomplish this, you performed the following key tasks:

- Configured container security by defining users and roles
- Enforced container-managed security by setting JSF navigation to use HTTP redirect
- Set up application access by defining the pages that each role can access
- Created a class to manage roles
- Created classes that enable you to access information about users and roles from the application

7

Implementing Login, Display, and Navigation Logic

In the preceding chapter, you defined how users log in to the application (authentication) and are allowed to access certain functionality (authorization). You specified this login logic at the application level.

Now you create pages that implement the login logic. You also create a page to conditionally display links enabling users to navigate to the pages that they are authorized to use.

If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh06.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

- Creating the Index Page
- Creating the Error Page
- Login Page Overview
- Creating the Login Page
- Specifying Where to Save Application State
- Welcome Page Overview
- Creating the Welcome Page
- Adding Navigation and Display Logic
- Testing the Login, Navigation, and Display Logic
- Summary

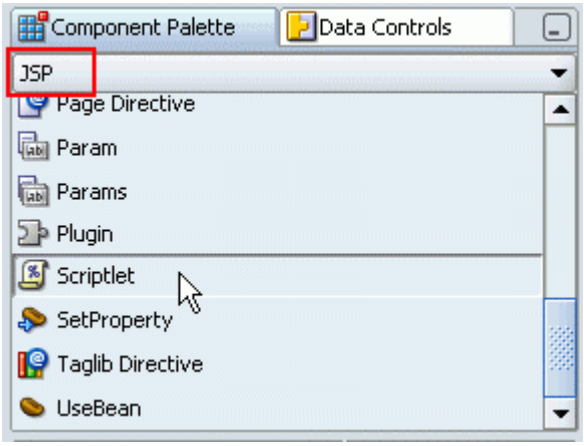
Creating the Index Page

Typically the home page of a Web application is named `index.*` (for example, `index.html`). By default, if no page is specified in the URL, the index page is loaded automatically.

In this application, the main page that you want users to see is the SRWelcome page, so now you create an index page that simply redirects the user to the SRWelcome page. You then specify that the index page should be the entry page of the application by modifying the deployment properties.

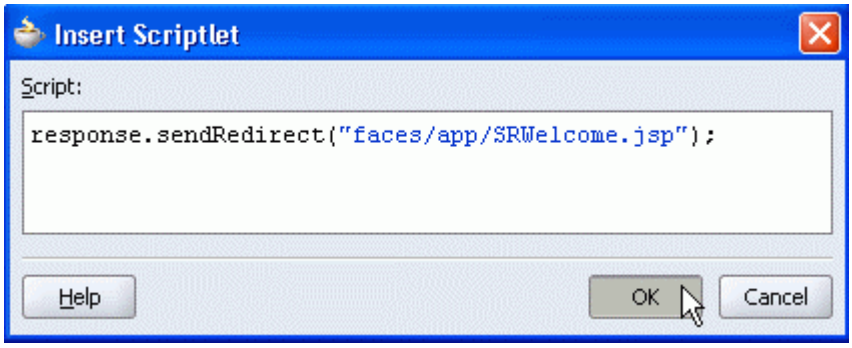
To create the index page and ensure that it is the entry point of the application, perform the following steps:

1. Right-click the **ViewController** project and select **New** from the context menu.
2. In the Categories list, expand **Web Tier** and select **JSP**. Then select **JSP** from the Items list and click **OK**.
3. If the Welcome page of the Create JSP wizard appears, click **Next**.
4. On the JSP File page of the wizard, enter the name of **`index.jsp`**, and then click **Next**.
5. On the Error Page Options page of the wizard, select the option **Do Not Use an Error Page to Handle Uncaught Exceptions in this File**, and click **Finish**.
6. The index.jsp file opens in the visual editor. From the Component Palette drop-down list, select **JSP** and then select the **Scriptlet** component as shown.



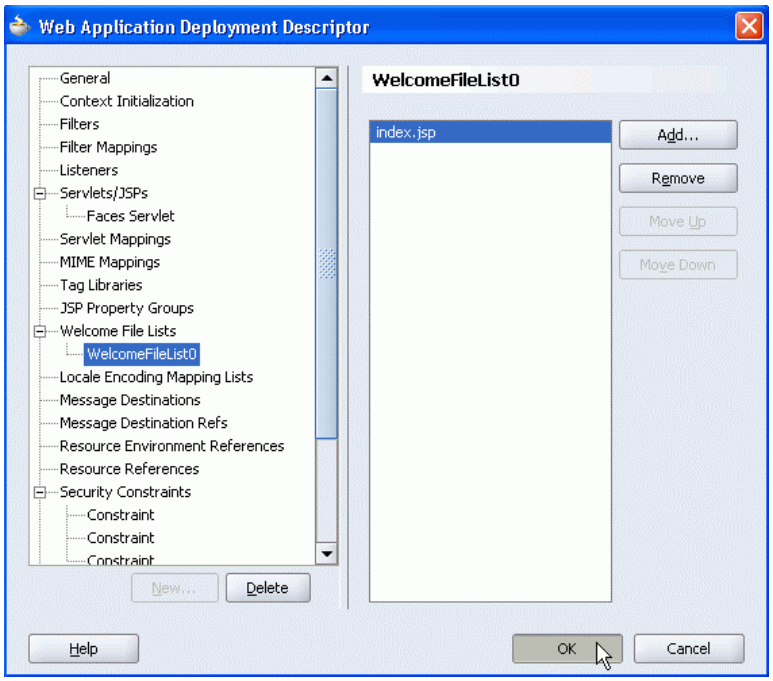
7. In the Insert Scriptlet dialog box, insert the following text:
   `response.sendRedirect("faces/app/SRWelcome.jsp");`

   Then click **OK**.



This scriptlet ensures that when the index page is invoked, the user is redirected to the SRWelcome page.

8. Now you need to set `index.jsp` as the entry point of the application. In the Applications Navigator, right-click **web.xml** (**ViewController** > **Web Content** > **WEB-INF** > **web.xml**) and select **Properties** from the context menu.
9. In the tree at the left side of the page, select **Welcome File Lists** and click **New**.
10. On the right portion of the page, click **Add**.
11. In the Create Welcome File dialog box, enter **index.jsp** and click **OK**.
12. Click **OK** to dismiss the Web Application Deployment Descriptor window (for `web.xml`).



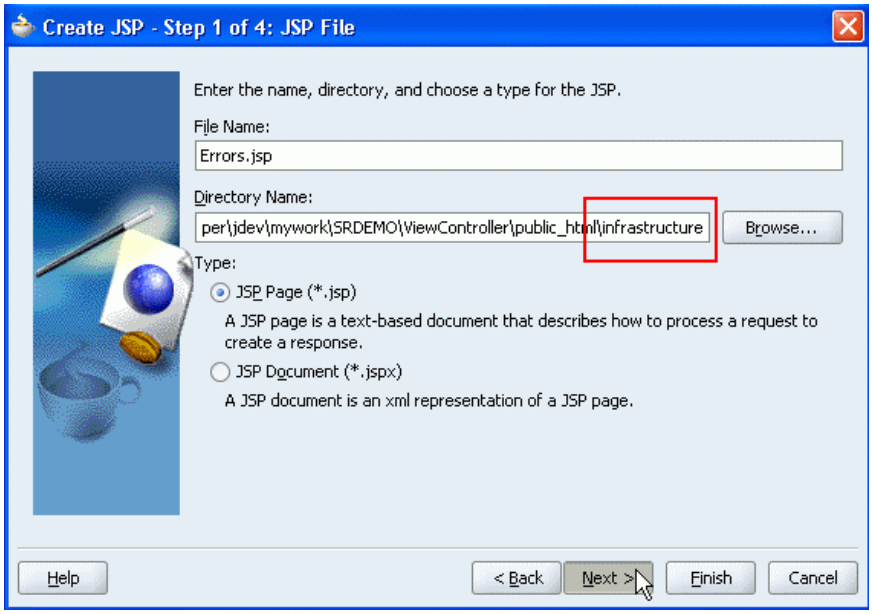
13. Click **Save All** to save your work.

Creating the Error Page

To create the page to display login error messages, perform the following steps:

1. Right-click the **ViewController** project and select **New** from the context menu.
2. In the Categories list, expand **Web Tier** and select **JSP**. Then select **JSP** from the Items list and click **OK**.
3. If the Welcome page of the Create JSP wizard appears, click **Next**.
4. As shown in the following screenshot, on the JSP File page of the wizard, enter the name of **`Errors.jsp`**.

   To the existing directory name, add **`\infrastructure`** and then click **Next**.

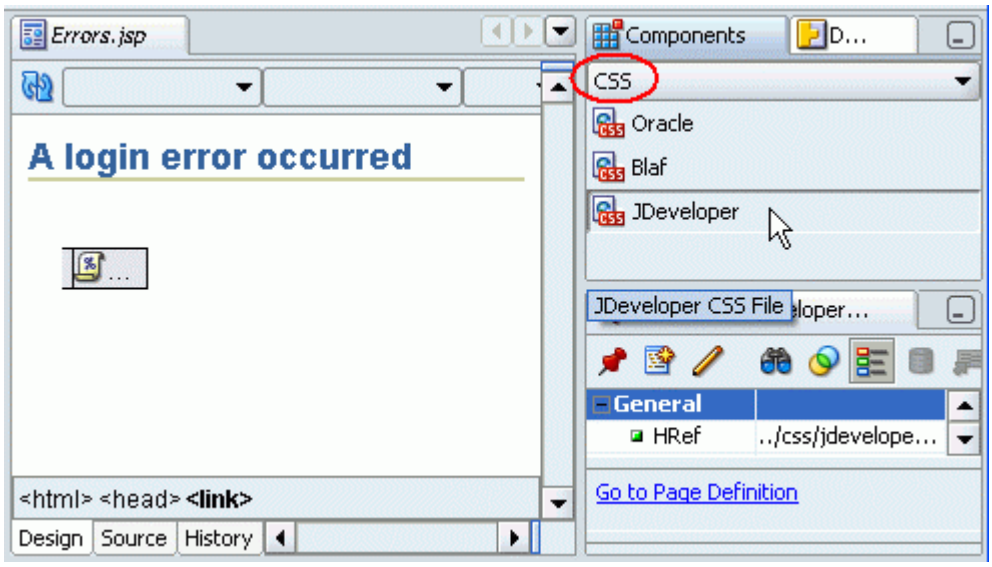


5. On the Error Page Options page of the wizard, select the option **Create This File as an Error Page**, and then click **Finish**.
6. The `Errors.jsp` file opens in the visual editor. Locate the following text:

   `An error occured:`

   Change it to:

   `A login error occurred`
7. From the Block Format drop-down list at the top-left portion of the editor, select **Heading 2**.
8. In the Component Palette, select **CSS** from the drop-down list and then click **JDeveloper**. Your page should now look like the following screenshot:

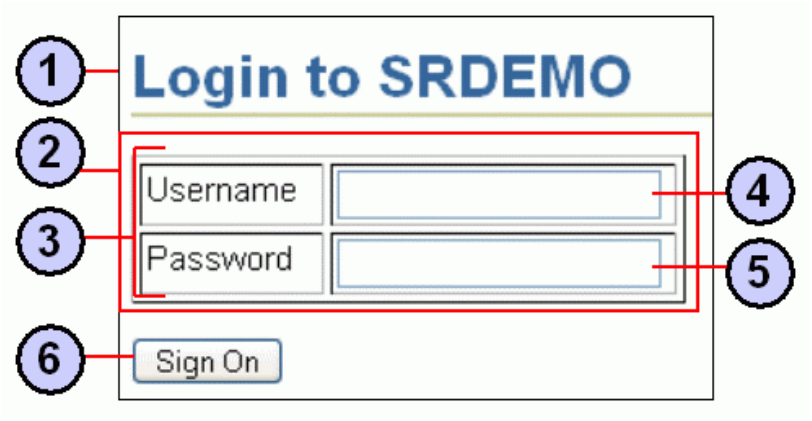


9. Click **Save All** to save your work.

Login Page Overview

In the previous chapter, you specified that the application should use form-based authentication, and that the login page is called Login.jsp. You now define this page, which displays automatically when a user first attempts to access the application.

The following screenshot depicts details of the Login page:



1. Each page that is displayed to users contains a title, which is different for each page.
2. The Login page contains an HTML table to arrange the input and output components in a tabular format.
3. The left cells of the table contain output text components with static values to serve as labels for the cells in the right columns.
4. The top right cell of the table contains an HTM text input component for entering the user name.
5. The bottom right cell of the table contains an HTL password input component for entering a password.
6. The action that is specified for the Sign On button authenticates the user, displaying an error page if unsuccessful. If the login is successful, the first page of the application displays.

Creating the Login Page

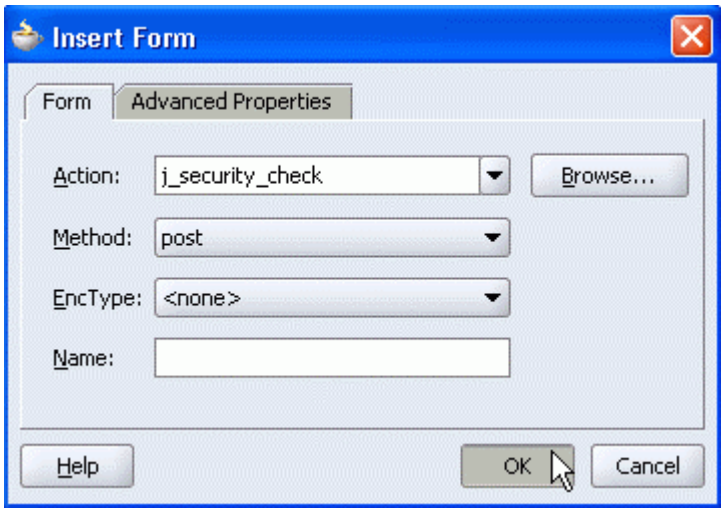
You now create the page that is used for Java EE form-based authentication, which you specified earlier to be used for this application. For this to be successful, the form must contain the following tags:

```
<form action="j_security_check" method="post" >
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

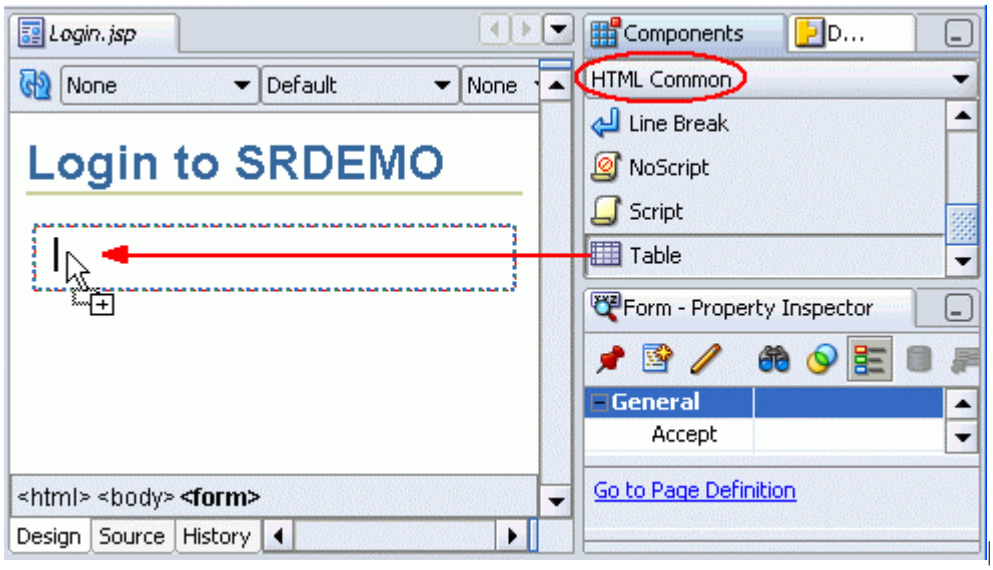
To create the logon page that accepts the user credentials and authenticates the user, perform the following steps:

1. Expand the **ViewController** project and the **Web Content** node. Right-click the **infrastructure** node and select **New** from the context menu.
2. In the Categories list, expand **Web Tier** and select **JSP**. Then select **JSP** from the Items list and click **OK**.
3. If the Welcome page of the Create JSP wizard appears, click **Next**.
4. On the JSP File page of the wizard, enter the name of **`Login.jsp`**. Ensure that the directory name ends with **`/infrastructure`**, and then click **Next**.

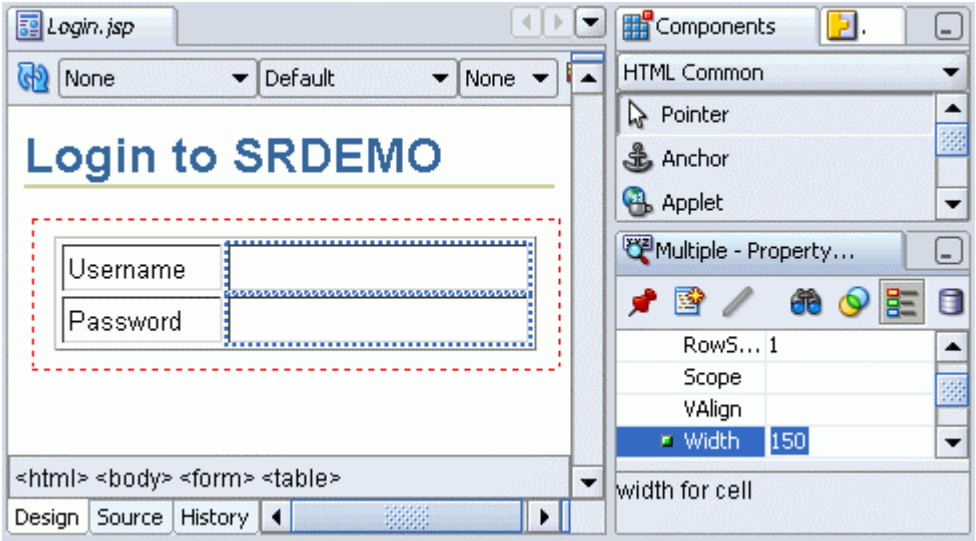
5. On the Error Page Options page of the wizard, select the option **Use an Error Page to Handle Uncaught Exceptions in This File**, and then click **Next**.
6. On the Use Error Page page of the wizard, select **Errors.jsp** and click **Finish**.
7. The `Login.jsp` file opens in the visual editor. Enter the text **`Login to SRDEMO`**.
8. From the Block Format drop-down list in the top-left portion of the editor, select **Heading 1**.
9. In the Component Palette, select **CSS** from the drop-down list and then click **JDeveloper**.
10. Place your cursor on the line under the heading. From the Component Palette drop-down list, select **HTML Forms**.
11. From the component list, click **Form**.
12. In the Insert Form dialog box, enter an Action of **`j_security_check`** and select **post** from the Method drop-down list, as shown in the following screenshot. Then click **OK**.



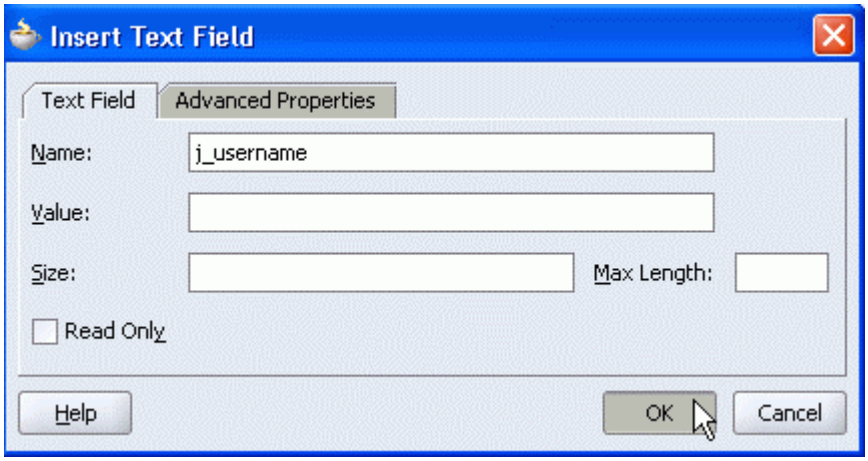
13. In the Component Palette, select **HTML Common** from the drop-down list. Select the **Table** component and drag it to the form that you created previously, as shown here:



14. In the Insert Table dialog box, set both Rows and Columns to **2**. Set Width to **250** and select the **pixels** option, and then click **OK**.
15. Enter the field labels **Username** and **Password** in the left column of the table.
16. Place the cursor in the upper-right cell of the table. Hold down the **[Shift]** key and click in the lower-right cell of the table. In the Property Inspector, set the Width property for the column to **150**. The page should look similar to the following screenshot:



17. Switch back to forms components by selecting **HTML Forms** from the Component Palette drop-down list.
18. Select the **Text Field** component and drag it to the top-right cell of the table. In the Insert Text Field dialog box, enter the Name **j_username** as shown and click **OK**.



19. Select the **Password Field** component and drag it to the bottom-right cell of the table. In the Insert Password Field dialog box, enter the Name **j_password** and click **OK**.
20. Place your cursor to the right of the table and press **[Enter]** to create a blank line within the form, and then click the **Submit Button** component.
21. In the Insert Submit Button dialog box, enter the Name **logon** and the value **Sign On**. Then click **OK**.

22. Click the **Source** tab and examine the code. You can see that you have defined the elements that are necessary for form-based authentication as described at the beginning of this section, as shown in the following screenshot:

```
Login.jsp
        Login to SRDEMO
    </h1><form action="j_security_check" method="post">
      <table cellspacing="2" cellpadding="3" border="1" width="250">
        <tr>
          <td>Username</td>
          <td width="150">
            <input type="text" name="j_username"/>
          </td>
        </tr>
        <tr>
          <td>Password</td>
          <td width="150">
            <input type="password" name="j_password"/>
          </td>
        </tr>
      </table>
      <p>
        <input type="submit" name="logon" value="Sign On"/>
      </p>
    </form></body>
Design  Source  History
```

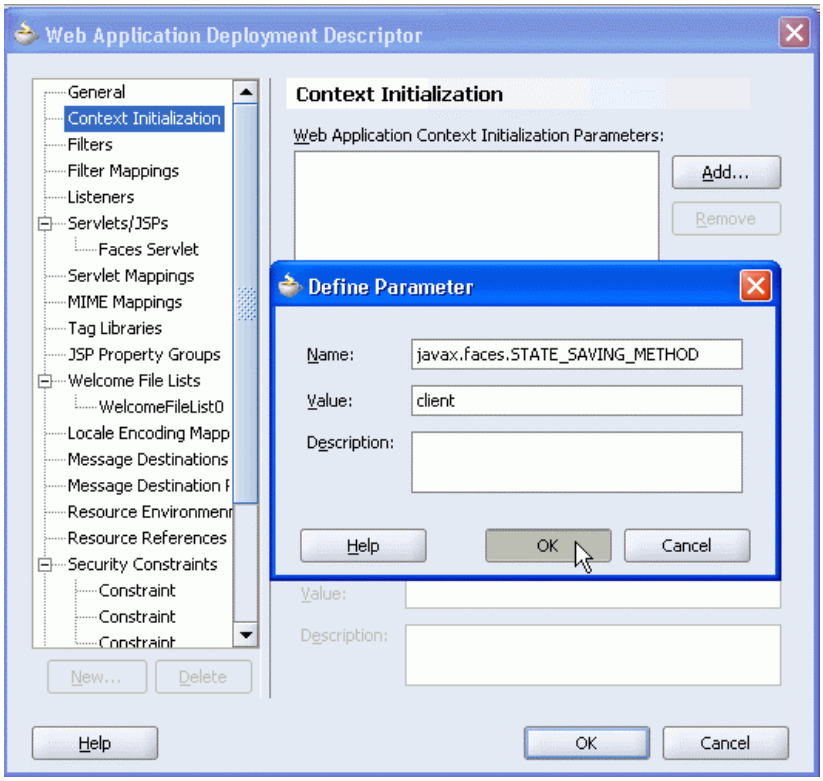
23. Click **Save All** to save your work.

Specifying Where to Save Application State

In JavaServer Faces (JSF), the state of the application is saved on the server by default. To improve performance, you can save application state on the client instead. This saves the state of the application into a hidden field on the page that you can see if you view the page source at run time.

To specify the client for saving application state, perform the following steps:

1. In the editor or the Applications Navigator (**ViewController** > **Web Content** > **WEB-INF** > **web.xml**), right-click **web.xml** and select **Properties** from the context menu.
2. In the tree at the left side of the Web Application Deployment Descriptor window, select **Context Initialization**, and then click **Add** at the right side of the window.
3. In the Define Parameter dialog box, enter the Name `javax.faces.STATE_SAVING_METHOD` and the Value `client`, and then click **OK**.



4. Click **OK** to dismiss the Web Application Deployment Descriptor window.

Welcome Page Overview

Next you create the SRWelcome page that all users see first after logging in to the application. The following screenshot depicts the features of the Welcome page:

1. Each application page contains a title, information about the logged in user, and a Logout link:
   - The title is different for each page.

- Information about the logged in user is derived from the `UserInfo` managed bean by using Expression Language.
- The Logout link logs out the user and returns to the Login page by executing a method from the `Logout` managed bean.

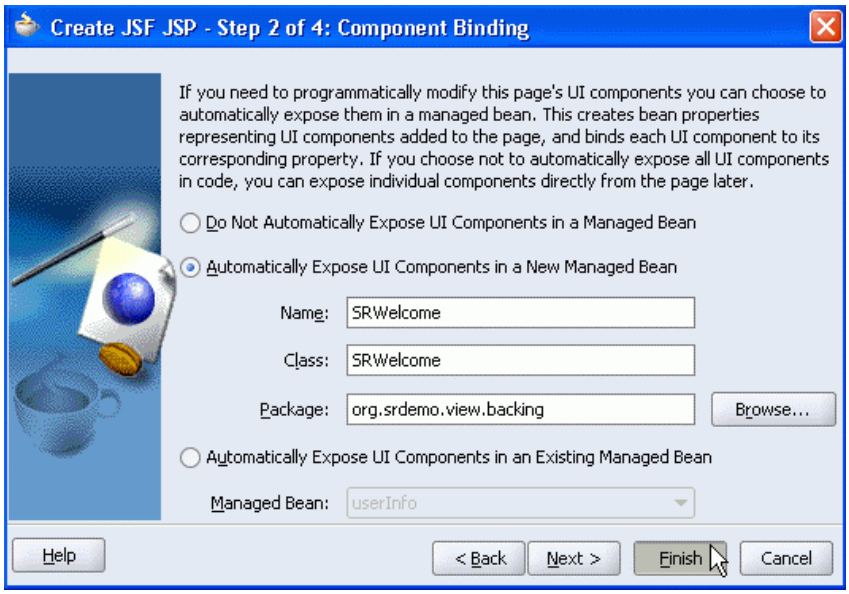
2. The View Your Service Requests link navigates to the SRList page to display service requests that were either created by or assigned to the logged in user. This link is visible to all users.
3. The Create a New Service Request link navigates to the SRCreate page, where users can create a new service request. This link is visible to all users.
4. The Triage Service Requests link navigates to the SRTriage page, where managers assign service requests to technicians. This link is visible only if the logged-in user has the role of `manager`.

To display the links conditionally based on the role of the logged-in user, you add code to the backing bean for the page to set the Rendered property of the link. You also specify an action for each link that corresponds to a From outcome in a navigation rule that you defined earlier.

Creating the Welcome Page

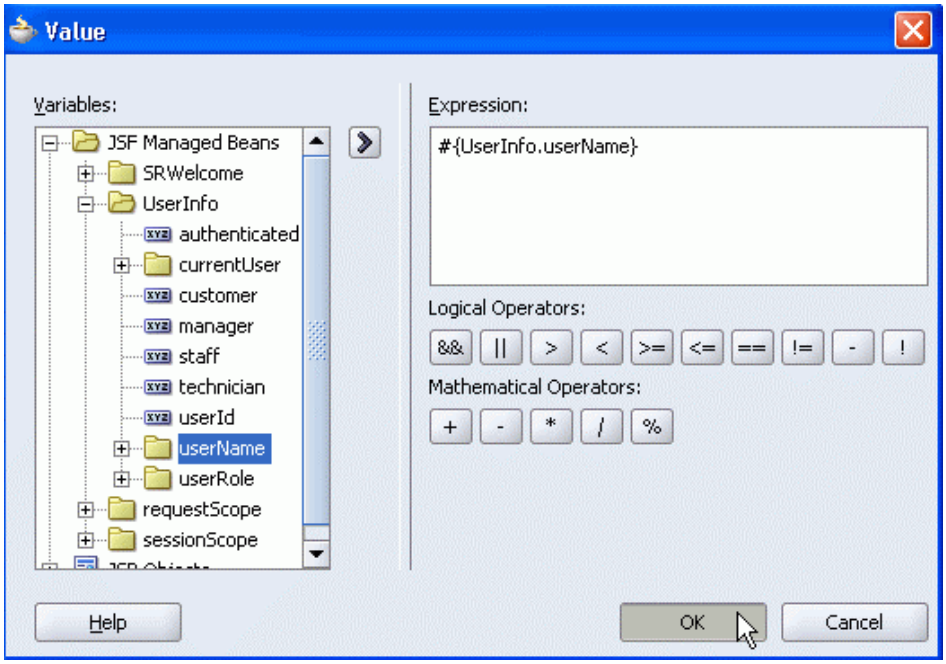
To create the Welcome page, perform the following steps:

1. Open **faces-config.xml** in the editor (if it is not already open), or click its tab if it is open. (To open the file, right-click the **ViewController** project and select **Open JSF Navigation** from the context menu.) Click the **Diagram** tab.
2. Double-click the **/app/SRWelcome.jsp** icon to invoke the Create JSF JSP wizard.
3. If the Welcome page of the wizard appears, click **Next**.
4. On the JSP File page of the wizard, click **Next**.
5. On the Component Binding page of the wizard (as shown in the following screenshot), select the option **Automatically Expose UI Components in a New Managed Bean**. Change the Name of the managed bean to **`SRWelcome`** and the Package to **`org.srdemo.view.backing`**. Click **Finish**.

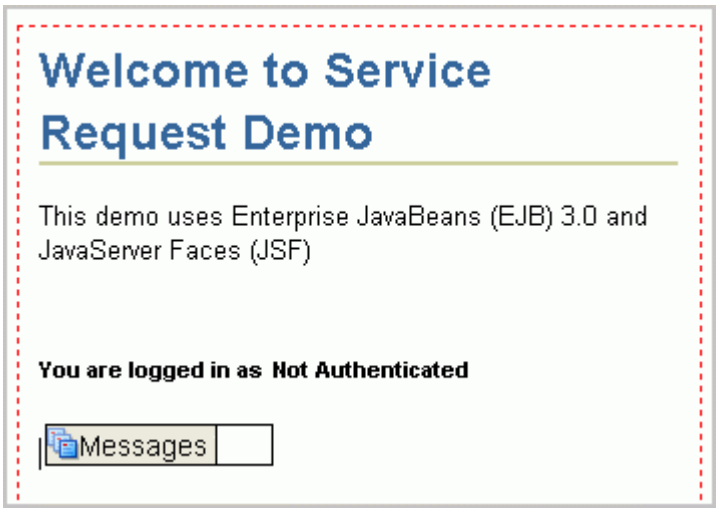


6. The new page opens in the editor. Enter the text **`Welcome to Service Request Demo`** and then select **Heading 1** from the Block Format drop-down list.
7. On the next line, enter the text **`This demo uses Enterprise JavaBeans (EJB) 3.0 and JavaServer Faces (JSF)`**. Ensure that the Block Format is set to **None**. Press **[Enter]** twice to skip a line.
8. Enter the text **`You are logged in as`**, and press **[Space]**. From the Block Format drop-down list, select **Heading 5**.
9. From the Component Palette drop-down list, select **JSF HTML** and then click **Output Text**.
10. An output text field is created in the editor. In the Property Inspector for the output text, select the **Value** property and then click **Bind to Data**.
11. In the Value dialog box, expand **JSF Managed Beans** and **UserInfo**. Select **userName** and click the right arrow to add it to the expression on the right. Then click **OK**.

    This causes the application to display the name of the authenticated user at run time. If you examine the Value property, you can see that it uses Expression Language to obtain the name of the user from the `UserInfo` managed bean that you defined in the previous chapter.



12. Place the cursor on the next line in the editor and select **Messages** from the Component Palette. This provides a placeholder for run-time messages to be displayed.
13. Select **CSS** from the Component Palette drop-down list and then select **JDeveloper**. Your page should now look similar to the following screenshot (instead of "Not Authenticated" you may see "#{UserInfo.userName}"):



14. Click **Save All** to save your work.

Adding Navigation and Display Logic

Now you add links, backing bean code, and actions to perform the display and navigation logic of the welcome page. You then wire components on the welcome page to code in the backing beans.

In this section you create several links and add the code to:

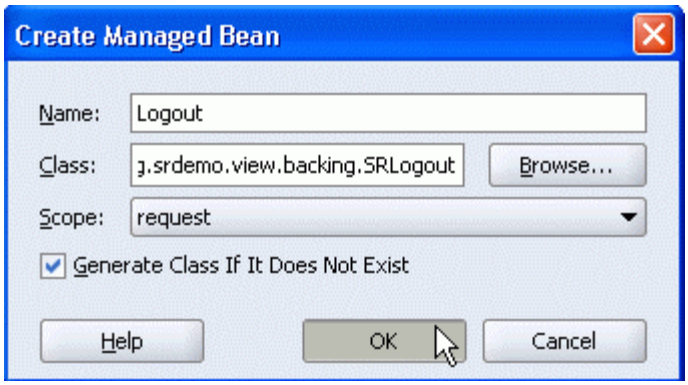
- Enable users to log out
- List the user's service requests
- Create a new service request
- Triage service requests

The Triage link is displayed conditionally to those users who have the authorized role of `manager`.

Enabling Users to Log Out

First you code the logic to enable users to log out by creating a Logout managed bean, and then you call its code from the SRWelcome page. Perform the following steps:

1. Open or switch to **faces-config.xml** and click the **Overview** tab. (To open the file, right-click the **ViewController** project and select **Open JSF Navigation** from the context menu.)
2. In the Managed Bean section, click **New**.
3. In the Create Managed Bean dialog box, enter the Name **`Logout`** and the Class **`org.srdemo.view.backing.SRLogout`**. Ensure that the Scope is set to **request** and that **Generate Class If It Does Not Exist** is checked, as shown in the following screenshot. Then click **OK**.



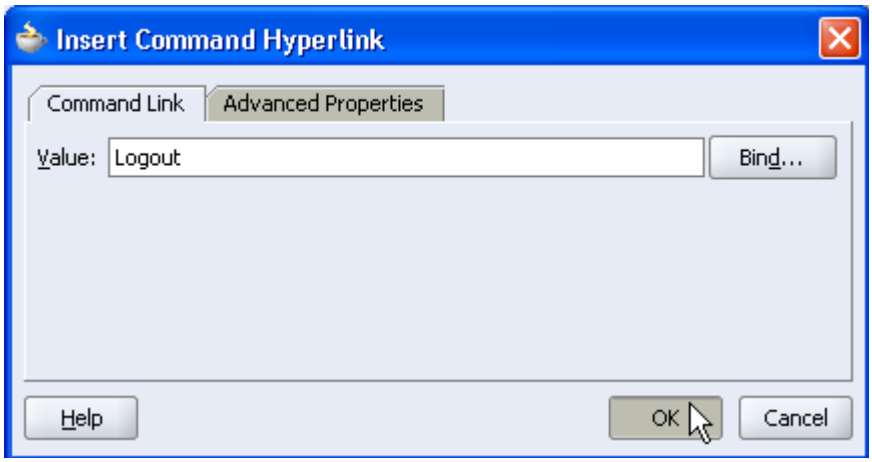
4. The `SRLogout.java` file is created and opened in the editor. Switch to it by clicking its tab, and then add the following method to log out the current user and redisplay the SRWelcome page, as shown here:

```
public String logoutButton_action() throws IOException {
  ExternalContext ectx =
    FacesContext.getCurrentInstance().getExternalContext();
  HttpServletResponse response =
    (HttpServletResponse)ectx.getResponse();
  HttpSession session = (HttpSession)ectx.getSession(false);
  session.invalidate();

  response.sendRedirect("SRWelcome.jsp");
  return null;
}
```

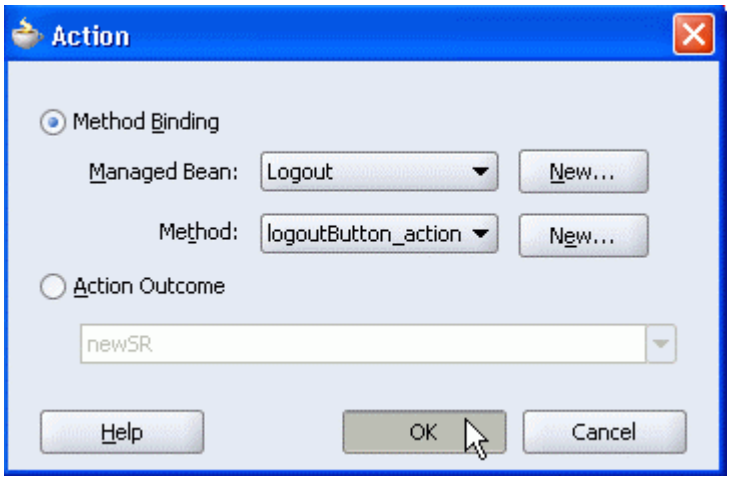
As you add the method, you are prompted several times to add import statements. Press **[Alt]+[Enter]** to import each one when prompted.

5. Add a component to the **SRWelcome** page to call this code. Switch to that page in the editor by clicking its tab.

   Position your cursor on the blank line above the "You are logged in as" text. In the Component Palette, select **JSF HTML** from the drop-down list and click **Command Hyperlink**. In the Insert Command Hyperlink dialog box, enter the Value `Logout` and click **OK**.



6. In the Property Inspector for the command hyperlink, enter the Action `#{}` and press **[Enter]**, and then click back into the Action property. Click the ellipsis ... at the end of the field.
7. In the Action dialog box, ensure that the **Method Binding** option is selected. Select **Logout** from the Managed bean drop-down list, and then select **logoutButton_action** from the Method drop-down list, as shown. Click **OK**.



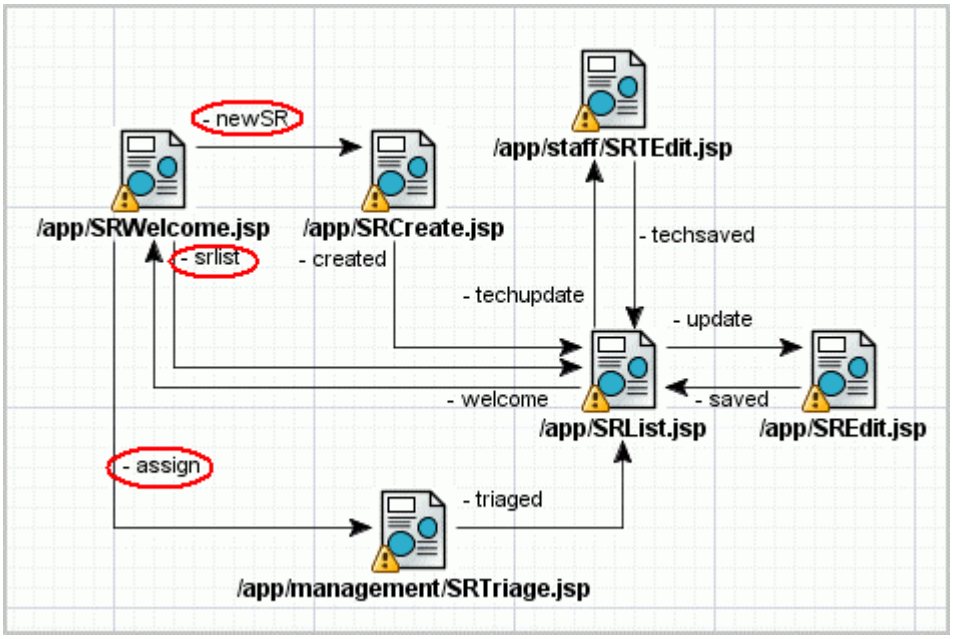
8. Click **Save All** to save your work.

Creating Links to Display Conditionally

Next you create links on the SRWelcome page to other parts of the application. The links to the SRList and SRCreate pages are displayed for all users, but the link to the SRTriage page has logic to display it only for

the `manager` role because only managers are allowed to triage (assign) service requests.

You first add actions to all of the links that correspond to the From outcomes that you defined when setting up navigation rules in the JSF Navigation Diagram (see Chapter 5), as shown in the following screenshot. To implement the link that displays conditionally, you add the conditional logic to the SRWelcome managed bean and then call that code from the link that you place on the SRWelcome page.



You create links to the following pages:

Page	From Outcome	Displays for:
SRList	`srlist`	all users
SRCreate	`newSR`	all users
SRTriage	`assign`	managers

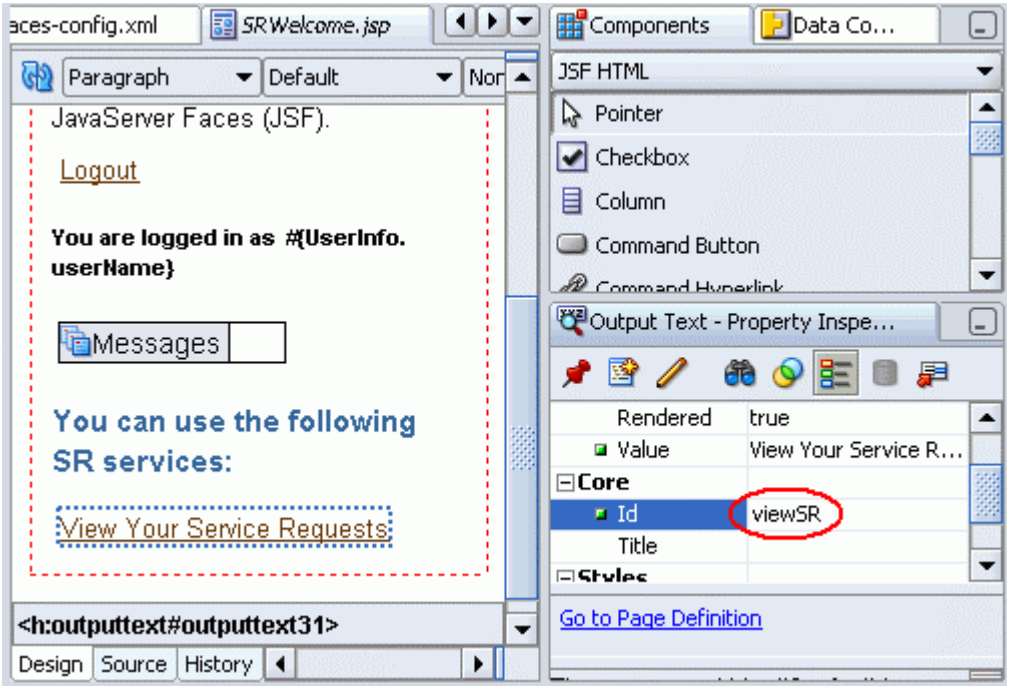
Creating a Link to the SRList Page

First you add a link on the SRWelcome page that navigates to the SRList page. The SRList page is available to all users, but for consistency you add a framework for conditional logic to the SRWelcome bean so that you can easily add display conditions if needed later. The conditional logic sets the Rendered property of the link that controls whether the link is displayed at run time. Finally, you add an action that executes the navigation rule with the `srlist` From outcome.

To create a link to the SRList page that displays the user's service requests, perform the following steps:

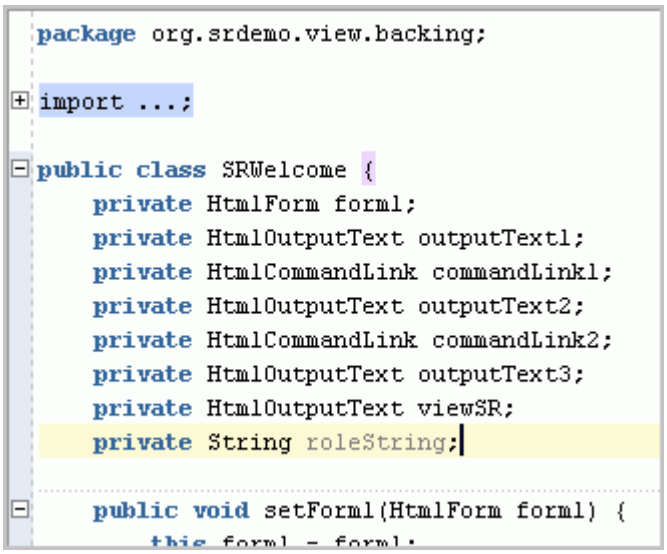
1. Switch to the **SRWelcome.jsp** page by clicking its tab in the editor. On a blank line below the Messages component, enter the text **`You can use the following SR services:`** and then select **Heading 3** from the Block Format drop-down list. Press **[Enter]** to create a blank line.
2. From the Component Palette drop-down list, select **JSF HTML** and then click the **Command Hyperlink** component to create it on the page.
3. In the Insert Command Hyperlink dialog box, enter the Value **`View Your Service Requests`** and click **OK**.

4. From the Block Format drop-down list, select **None**.
5. In the Property Inspector, change the ID of the output text component to **`viewSR`** (be sure that **h:outputText** is selected in the Structure window).



6. In the Applications Navigator, expand **ViewController**, **Application Sources**, **org.srdemo.view**, and **backing**. Double-click **SRWelcome.java** to open it in the editor. SRWelcome.java is the backing bean for the SRWelcome page.
7. As shown in the screenshot, add the following to the local variable declarations:

```
private String roleString;
```



```
package org.srdemo.view.backing;

import ...;

public class SRWelcome {
    private HtmlForm form1;
    private HtmlOutputText outputText1;
    private HtmlCommandLink commandLink1;
    private HtmlOutputText outputText2;
    private HtmlCommandLink commandLink2;
    private HtmlOutputText outputText3;
    private HtmlOutputText viewSR;
    private String roleString;

    public void setForm1(HtmlForm form1) {
        this.form1 = form1;
```

8. Add the following code that obtains the role of the current user:

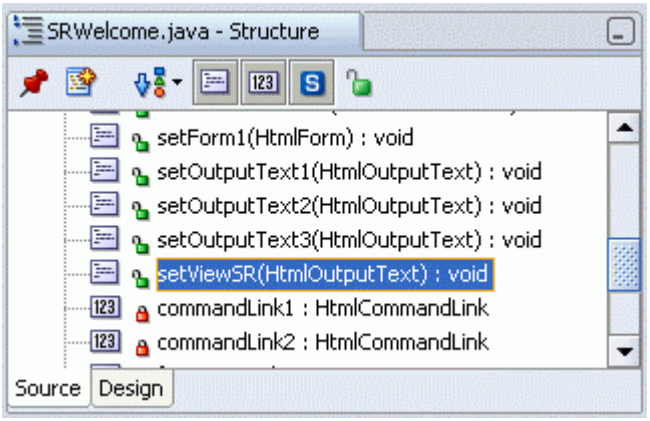
```
public SRWelcome() {
    FacesContext ctx = FacesContext.getCurrentInstance();
    Application app = ctx.getApplication();
    ValueBinding curUser =
      app.createValueBinding("#{UserInfo.userRole}");
    Object obj = curUser.getValue(ctx);
    roleString = obj.toString();
}
```

```
    public HtmlOutputText getViewSR() {
        return viewSR;
    }

    public SRWelcome() {
        FacesContext ctx = FacesContext.getCurrentInstance();
        Application app = ctx.getApplication();
        ValueBinding curUser =
          app.createValueBinding("#{UserInfo.userRole}");
        Object obj = curUser.getValue(ctx);
        roleString = obj.toString();
    }

}
```

9. When prompted, press **[Alt]+[Enter]** to import:

   ```
   javax.faces.context.FacesContext
   javax.faces.application.Application
   javax.faces.el.ValueBinding
   ```

10. In the Structure window, double-click the **`setViewSR()`** method to find it in the editor.



11. Change the method (as shown below) to conditionally display the `viewSR` output text.

    Note: Although all of these conditions produce the same result of rendering the output text for all roles (the default behavior), this provides a structure for easily changing to conditional logic later if needed.

    ```
    public void setViewSR(HtmlOutputText viewSR) {
       if (roleString.equals("user")) {
    ```

```
            viewSR.setRendered(true);
              }
        else if (roleString.equals("technician")) {

            viewSR.setRendered(true);
                }
        else {
            viewSR.setRendered(true);
              }
        this.viewSR = viewSR;
    }
```

```
        return commandLink2;
    }

    public void setViewSR(HtmlOutputText viewSR) {
            if (roleString.equals("user")) {
                viewSR.setRendered(true);
                  }
             else if (roleString.equals("technician")) {

                viewSR.setRendered(true);
                    }
             else {
                viewSR.setRendered(true);
                  }
             this.viewSR = viewSR;
        }

    public HtmlOutputText getViewSR() {
```

12. In the **SRWelcome.jsp** page in the editor, select the command hyperlink that you just created (be sure that **h:commandLink** is highlighted in the Structure window). In the Property Inspector, select **`srlist`** from the Action drop-down list.

    This is the name of the From outcome for the navigation rule that you created previously on the JSF Navigation Diagram, thus ensuring that when the link is clicked at run time, navigation proceeds to the SRList page.

13. Click **Save All** to save your work.

Creating a Link to the SRCreate Page

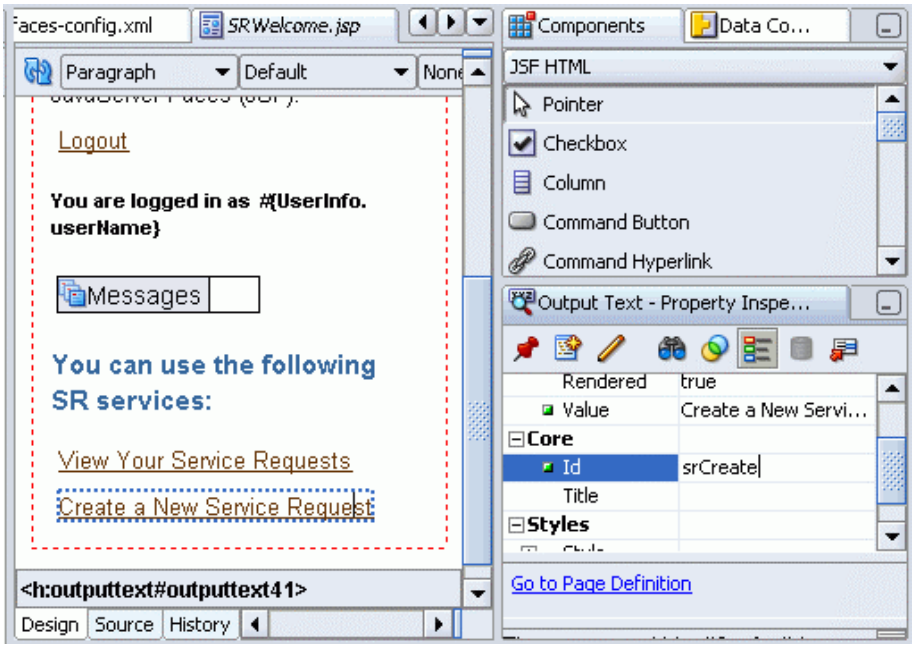
Next you define a link on the SRWelcome page to the SRCreate page, which enables creation of new service requests. As with the previous link that you defined, this link is displayed for all roles. As before, in order to have consistency and to more easily implement conditional logic if it is needed later, you first add code to the SRWelcome managed bean that conditionally sets the Rendered property of the link. You then add the action that specifies that users navigate to the SRCreate page by clicking the link.

Perform the following steps:

1. With **SRWelcome.jsp** open in the editor, place the cursor on the line after the viewSR link and select **Command Hyperlink** from the Component Palette.

2. In the Insert Command Hyperlink dialog box, enter the Value **`Create a New Service Request`** and click **OK**.

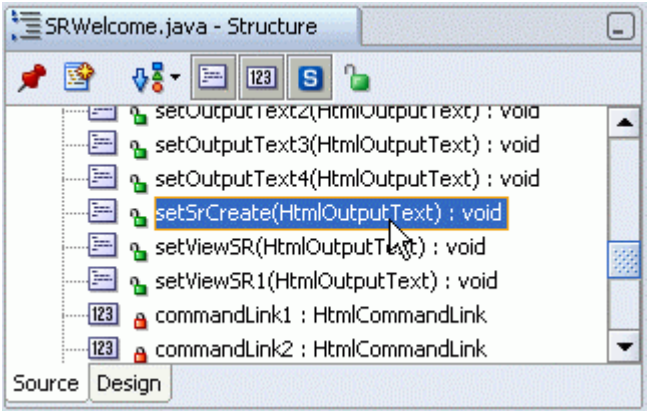
3. In the Property Inspector, change the ID of the output text component to **`srCreate`** (be sure that **h:outputText** is selected in the Structure window).



4. Open or switch to **SRWelcome.java** in the editor as follows:

   ViewController > **Application Sources** > **org.srdemo.view** > **backing** > **SRWelcome.java**

5. In the Structure window, double-click the **`setSrCreate()`** method to find it in the editor.



6. Change the method as shown below to conditionally display the `srCreate` output text

   Note: Although all of these conditions produce the same result of rendering the output text for all roles (the default behavior), this provides a structure for easily changing to conditional logic if needed later.

```
public void setSrCreate(HtmlOutputText srCreate) {
    if (roleString.equals("user")) {
        srCreate.setRendered(true);
```

```
            }
        else if (roleString.equals("technician")) {
            srCreate.setRendered(true);
            }
        else {
            srCreate.setRendered(true);
            }
        this.srCreate = srCreate;
    }
```

```
    }

    public void setSrCreate(HtmlOutputText srCreate) {
        if (roleString.equals("user")) {
            srCreate.setRendered(true);
            }
        else if (roleString.equals("technician")) {
            srCreate.setRendered(true);
            }
        else {
            srCreate.setRendered(true);
            }
        this.srCreate = srCreate;
    }

    public HtmlOutputText getSrCreate() {
```

7. In the **SRWelcome.jsp** page in the editor, select the command hyperlink that you just created (be sure that **h:commandLink** is highlighted in the Structure window). In the Property Inspector, select **`newSR`** from the Action drop-down list.

   This is the name of the From outcome for the navigation rule that you created previously on the JSF Navigation Diagram, thus ensuring that when the link is clicked at run time, navigation proceeds to the SRCreate page.

8. Click **Save All** to save your work.

Creating a Link to the SRTriage Page

The final link that you create is to the SRTriage page, which enables assigning service requests. Only managers are allowed to assign service requests, so this link should be displayed only for the `manager` role. To implement this conditional logic, you first add code to the SRWelcome managed bean that sets the Rendered property based on the user role. You then add the action to the link to ensure that navigation proceeds to the SRTriage page.

Perform the following steps:

1. With **SRWelcome.jsp** open in the editor, place the cursor on the line after the srCreate link. Then select **Command Hyperlink** from the Component Palette.
2. In the Insert Command Hyperlink dialog box, enter the Value **`Triage Service Requests`** and click **OK**.
3. In the Property Inspector, change the ID of the output text component to **`srTriage`** (be sure that **h:outputText** is selected in the Structure window).

4. Open or switch to **SRWelcome.java** in the editor. (**ViewController** > **Application Sources** > **org.srdemo.view** > **backing** > **SRWelcome.java**)
5. In the Structure window, double-click the **`setSrTriage()`** method to find it in the editor.
6. Change the method to the following code to conditionally display the `srTriage` output text (note that the `Rendered` property is set to `true` only for those users whose role is `manager`):

```
public void setSrTriage(HtmlOutputText srTriage) {
     if (roleString.equals("manager")) {
         srTriage.setRendered(true);
           }
     else if (roleString.equals("user")) {
         srTriage.setRendered(false);
           }
     else if (roleString.equals("technician")) {
         srTriage.setRendered(false);
           }
     else {
         srTriage.setRendered(false);
           }
     this.srTriage = srTriage;
 }
```

7. In the **SRWelcome.jsp** page in the editor, select the command hyperlink that you just created (be sure that **h:commandLink** is highlighted in the Structure window) and, in the Property Inspector, select **`assign`** from the Action drop-down list.

   This is the name of the From outcome for the navigation rule that you created previously on the JSF Navigation Diagram, thus ensuring that when the link is clicked at run time, navigation proceeds to the SRTriage page.

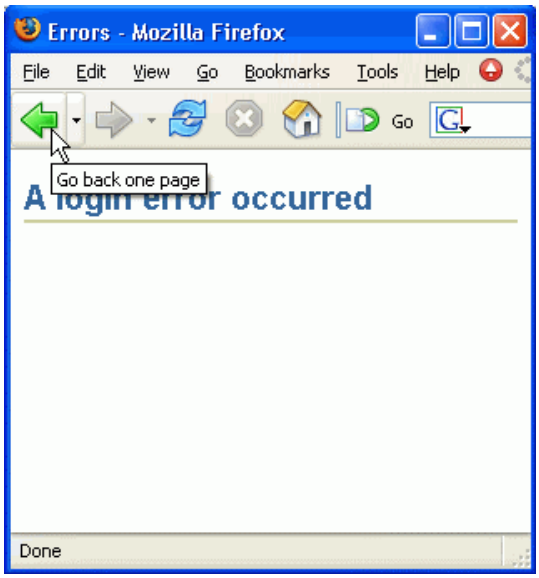
8. Click **Save All** to save your work.

Testing the Login, Navigation, and Display Logic

Now that you have defined the security and login logic and implemented display and navigation logic, you can test the functionality by performing the following steps:

1. In the Applications Navigator, expand the **ViewController** project and the **Web Content** node.
2. Right-click **index.jsp** and select **Run** from the context menu. A browser opens to display the Logon page.
3. Enter any random combination of letters for a Username and Password, and then click **Sign On**.

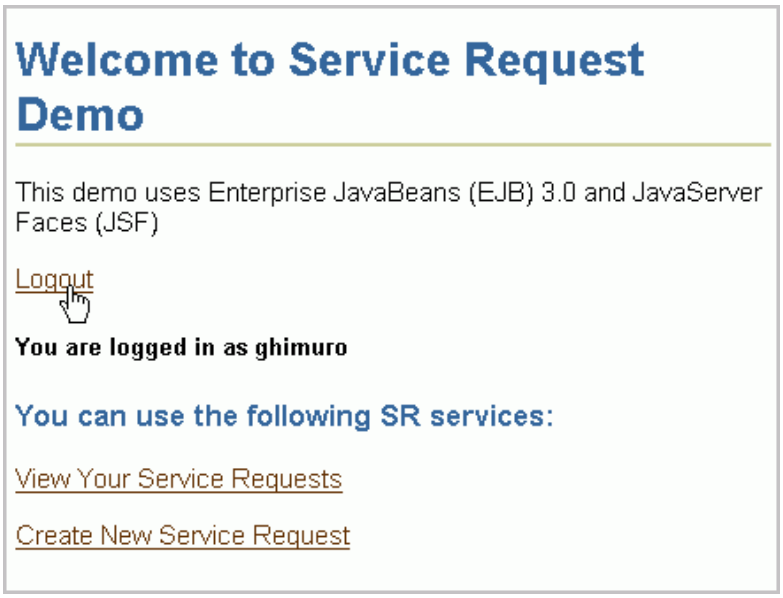
4. The Errors page displays a message that a login error occurred. Click the browser's **Back** button.



5. Now enter the following (for a customer) and click **Sign On**:

Username	Password
`ghimuro`	`welcome`

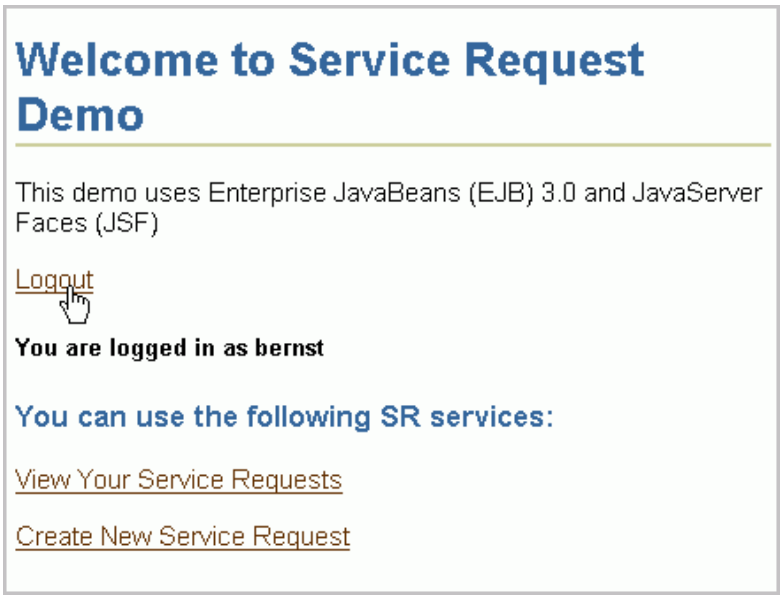
6. The SRWelcome page appears in the browser displaying the user's name and only the two links that customers (`user` role) are allowed to access. Click **Logout**.



7. The Logon page is once again invoked. Enter login information for a technician:

Username	Password
`bernst`	`welcome`

8. The SRWelcome page appears in the browser displaying the user's name and the two links that technicians are allowed to access, which are the same as the links that display for customers. Click **Logout**.



9. The Logon page is once again invoked. Enter login information for a manager:

Username	Password
`sking`	`welcome`

10. The SRWelcome page appears in the browser displaying the user's name and all links, because a manager has access to all of the pages.

11. Now that you have access to all of the links, click each of them in turn to test the navigation. Remember that you have not yet created the pages, so you should receive a 404 Not Found error when attempting to navigate, but you can see from the error message which page is being invoked. Click the browser's Back button to return to the welcome page.
12. Close the browser when you are finished testing.

Summary

In this chapter, you created the pages that provide the entry point of the application, implementing login logic and conditional authorization based on roles. To accomplish this, you performed the following key tasks:

- Created an index page that redirects users to the welcome page
- Created an error page to be displayed when the login is incorrect
- Created a login page where users can log in to the application with one of the user ID/password combinations that you defined in the previous chapter
- Specified that application state should be saved on the client
- Created a welcome page that displays the name of the current user along with a link enabling users to log out and links to other pages in the application
- Added logic to the welcome page to display links conditionally based on roles and to use the navigation rules that you set up previously
- Tested the security and logon logic and the display and navigation logic.

8

Developing a Page to List Records

In the preceding chapter, you implemented the login, display, and navigation logic. You now create the page that lists a user's service requests. There is one page in this application for all users: SRList.jsp.

A user accesses the list page from the welcome page by clicking a link. This accesses the data pertaining to the user's service requests and displays it on the list page.

If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh07.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

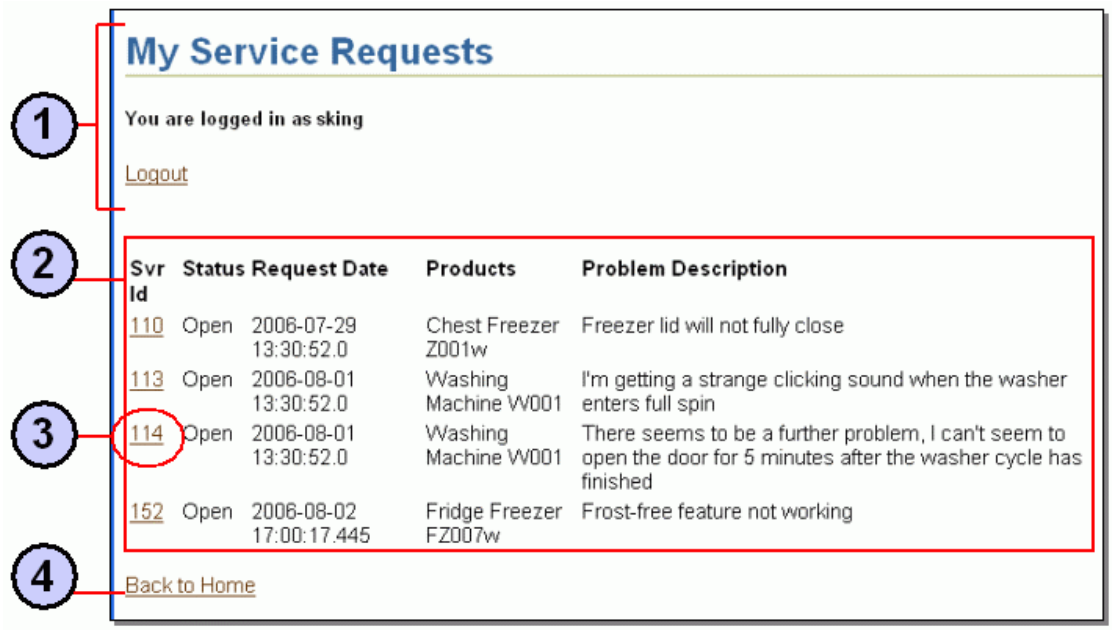
This chapter contains the following topics:

- List Page Overview
- Creating the List Page for Users
- Adding a Named Query
- Exposing the New Named Query
- Updating the Backing Bean to Retrieve the Data
- Displaying Data
- Passing Data Between Pages
- Specifying the Location of the Session Facade
- Testing the Update Logic and the UI Components
- Summary

List Page Overview

The SRList page is a high-level view of all service requests assigned to or created by an individual user. This page is accessed from a link on the SRWelcome page.

The following screenshot depicts details of the SRList page:



1. Each application page contains a title, information about the logged in user, and a Logout link:
   - The title is different for each page.
   - Information about the logged in user is derived from the `UserInfo` managed bean by using Expression Language.
   - The Logout link logs out the user and returns to the Login page by executing a method from the `Logout` managed bean.
2. The list of service requests is a JSF data table component to display service requests that were either created by or assigned to the logged in user. The data table is bound to a new attribute that you create in the page's backing bean. The accessor method for this attribute calls a named query, which you add to the `ServiceRequests` entity, to retrieve the list of service requests.
3. In addition to listing the requests, SRList enables users to drill into a particular service request, passing data to the page where they can edit their service request. The Svr Id column of the data table is a JSF command link component that executes one of two navigation cases that you defined earlier. You add code to the backing bean to determine if the logged in user is a customer or an employee (manager or technician). If the user is a customer, navigation to the edit page for customers (SREdit) occurs. Otherwise, navigation proceeds to the SRTEdit page for employees. In either case, the code passes the current row of data to the appropriate edit page.
4. The Back to Home link executes a navigation rule that returns the user to the SRWelcome page.

Creating the List Page for Users

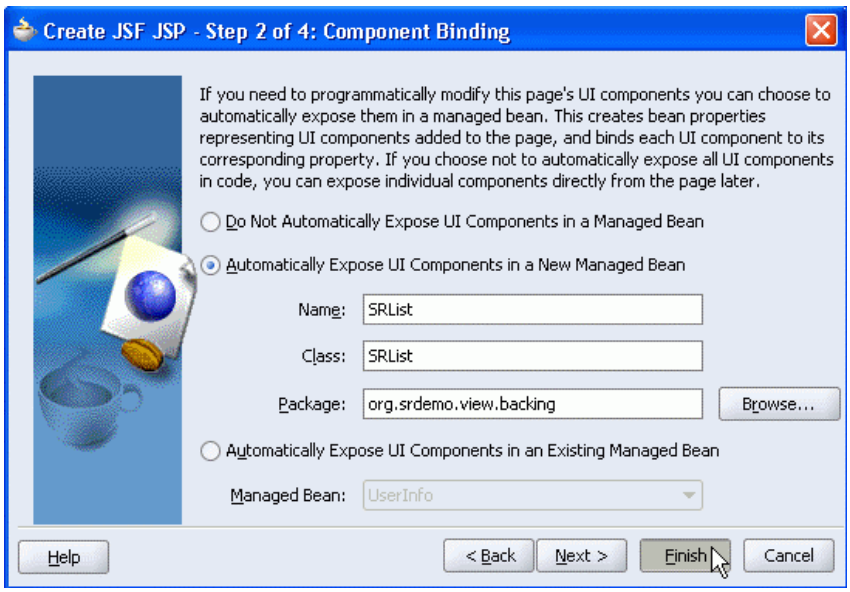
In this and subsequent pages in this application, you expose UI components in a managed bean. Managed

beans that handle the data exchange between the business model and the UI component are called *backing beans*. JavaServer Faces uses its own EL value binding to associate a UI component to a property on the backing bean. A backing bean can have a connection to the data model to retrieve and store data, thus handling the interaction between the View and the Model. Code in the backing bean is automatically synchronized with design changes to the UI components.

In a production application, you would probably not want to expose all UI components in a backing bean, because this generates a lot of extra code that may not be used. Instead, you could create the backing bean, add attributes to it, and wire attributes to UI components only as needed.

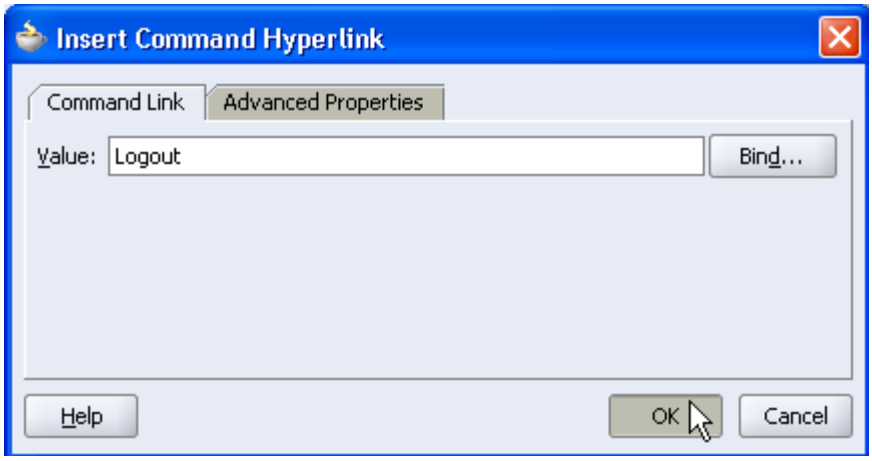
To define the SRList page, perform the following steps:

1. Open **faces-config.xml** in the editor if it is not already open, or click its tab at the top of the editor if it is open. (To open the file, right-click the **ViewController** project and select **Open JSF Navigation** from the context menu.) Click the **Diagram** tab.
2. Double-click the **/app/SRList.jsp** icon to invoke the Create JSF JSP wizard.
3. If the Welcome page of the wizard appears, click **Next**.
4. On the JSP File page of the wizard, click **Next**.
5. On the Component Binding page of the wizard, select the option **Automatically Expose UI Components in a New Managed Bean**. Change the Name of the managed bean to `SRList` and the Package to `org.srdemo.view.backing`. Click **Finish**.

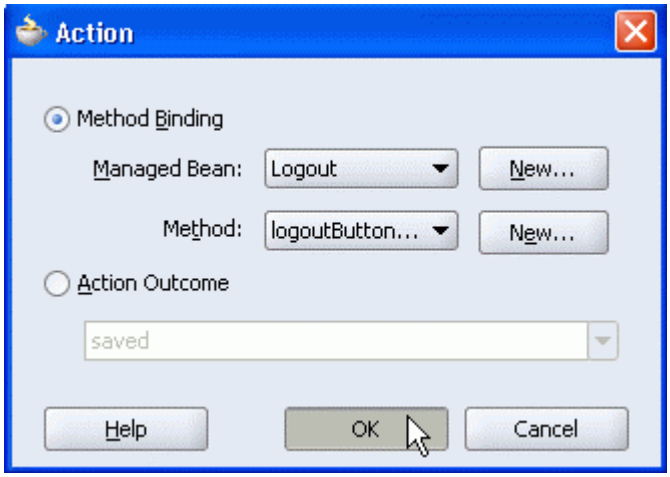


6. The new page opens in the editor. Enter the text `My Service Requests` and then select **Heading 1** from the Block Format drop-down list.
7. On the next line, enter the text `You are logged in as` and then press **[Space]**. From the Block Format drop-down list, select **Heading 5**.
8. In the Component Palette, select **JSF HTML** from the drop-down list and then click the **Output Text** component.
9. An output text field is created in the editor. In the Property Inspector for the output text, select the **Value** property and then click **Bind to Data** .

10. In the Value dialog box, expand **JSF Managed Beans** and **UserInfo**. Select **userName** and click the **right arrow** to add it to the expression on the right. Click **OK**. This causes the application to display the name of the authenticated user at run time.
11. In the visual editor, place the cursor on the next line. Then click the **Command Hyperlink** component in the Component Palette.
12. In the Insert Command Hyperlink dialog box, enter the Value `Logout` and click **OK** as shown.



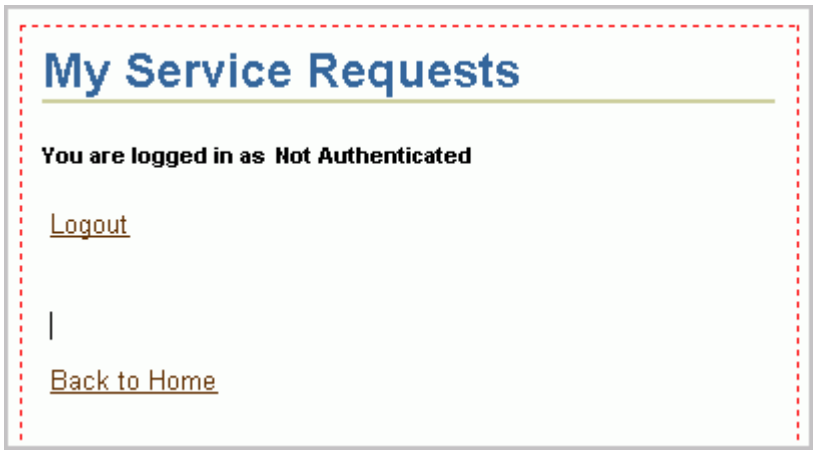
13. In the Property Inspector for the command hyperlink, enter an Action of `#{}` and press **[Enter]**, and then click back into the Action property. Click the **ellipsis** at the end of the field.
14. In the Action dialog box, ensure that the **Method Binding** option is selected. Select **Logout** from the Managed bean drop-down list, and select **logoutButton_action** from the Method drop-down list. Click **OK**.



This ensures that the Logout link executes the code in the Logout managed bean that you created in a previous chapter.

15. Now you add a link to return the user to the welcome page. Press **[Enter]** three times to insert three blank lines after the Logout link. From the Component Palette drop-down list, select **JSF HTML** and then click the **Command Hyperlink** component to create it on the page.
16. In the Insert Command Hyperlink dialog box, enter the Value `Back to Home` and click **OK**.

17. Check the Structure window to be sure that the **h:commandLink** component is selected. In the **Action** attribute within the Property Inspector for the command link, select **welcome** from the drop-down list.
18. Select **CSS** from the Component Palette drop-down list and then click **JDeveloper**. Your page should now look similar to the following screenshot:



19. Click **Save All** to save your work.

Adding a Named Query

The SRList page displays all service requests that were created by or assigned to the user. To support this data retrieval, you must create another named query by performing the following steps:

1. In the Applications Navigator, expand **Model**, **Application Sources** and **org.srdemo.persistence**.
2. Double-click **ServiceRequests.java** to open it in the editor.
3. Just above the line with the **`@NamedQuery`** annotation, add the following annotation:

   ```
   @NamedQueries({
   ```

4. When prompted, press **[Alt]+[Enter]** to import `javax.persistence.NamedQueries`.
5. Add the following named query **`findServiceRequests`** after the named query **`findAllServiceRequests`**:

   ```
   @NamedQuery(name="findServiceRequests", query="select " +
   "object (sr) from ServiceRequests sr where " +
   "sr.users = :user or sr.users1 = :user1")
   ```

6. Add a comma separator after the first named query and close the set of named queries by adding the following line:

   ```
   })
   ```

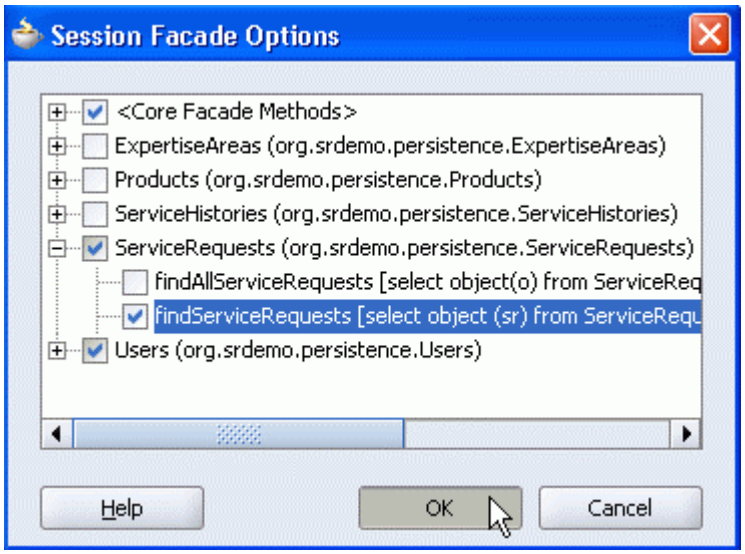
```
@Entity
@NamedQueries({
@NamedQuery(name="findAllServiceRequests", query="select object(o) " +
"from ServiceRequests o"),
@NamedQuery(name="findServiceRequests", query="select " +
"object (sr) from ServiceRequests sr where " +
"sr.users = :user or sr.usersl = :usersl"),
@NamedQuery(name="findTriageRequests",query="select object(sr)" +
" from ServiceRequests sr where " +
"sr.status = :status  or sr.usersl is null " )
})
@Table(name="SERVICE_REQUESTS")
```

7. Click **Save All** to save your work.

Exposing the New Named Query

The new named query cannot be used in the application until you modify the session facade to include a method to access it. To expose the new named query, perform the following steps:

1. The class containing the named query must be compiled in order for them to be visible to the facade editor, so click **Rebuild** to compile all.
2. In the Applications Navigator, expand **Model** > **Application Sources** > **org.srdemo.business**.
3. Right-click **ServiceRequestFacadeBean.java** and select **Edit Session Facade**.
4. In the Session Facade Options dialog box, expand the **ServiceRequests** node and select the check boxes next to the query that you just added: **findServiceRequests**. Click **OK**.



The `ServiceRequestFacadeBean.java` file opens in the editor. Scroll to the end of the file to see the new method that was added to expose the new named query. In the method **`findServiceRequests()`**, locate the following line:

```
return
em.createNamedQuery("findServiceRequests").setParameter("user",
user).setParameter("user1", user1).getResultList();
```

Replace this line with the following four lines (be sure to retain the opening and closing curly brackets):

```
Query query = (Query)em.createNamedQuery("findServiceRequests");
query.setParameter("user",user);
query.setParameter("user1",user1);
return query.getResultList();
```

```
/** <code>select object (sr) from ServiceRequests sr where sr.users =
public List<ServiceRequests> findServiceRequests(Object user,
                                                 Object user1) {
    Query query = (Query)em.createNamedQuery("findServiceRequests");
    query.setParameter("user",user);
    query.setParameter("user1",user1);
    return query.getResultList();
}
```

5. When prompted, press **[Alt]+[Enter]** to import `javax.persistence.Query`.
6. Click **Save All** to save your work.

Updating the Backing Bean to Retrieve the Data

The SRList page displays service requests that are either created by or assigned to the current user. The data is retrieved from the session facade and exposed by a data table in the SRList.jsp page. To modify the backing bean for the page to make this data available, perform the following steps:

1. In the Applications Navigator, expand the **ViewController** project and the **Application Sources**, **org.srdemo.view**, and **backing** nodes.
2. Double-click **SRList.java** to open it in the editor.
3. Add the following attribute:

   ```
   List<ServiceRequests> requestList = new ArrayList();
   ```

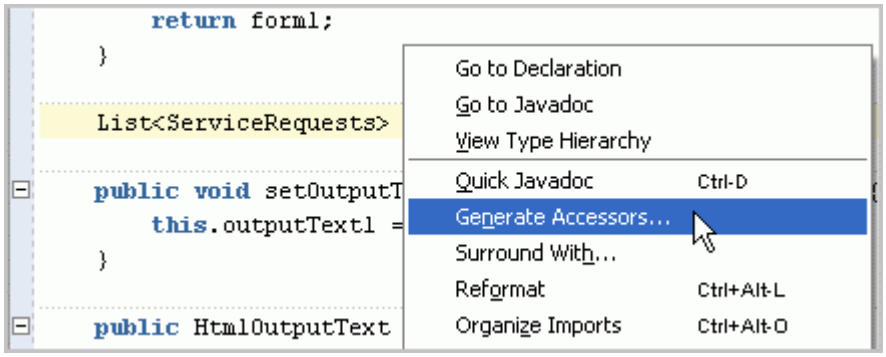
```
public class SRList {
    private HtmlForm form1;
    private HtmlOutputText outputText1;
    private HtmlCommandLink commandLink1;
    private HtmlOutputText outputText2;
    private HtmlOutputText outputText21;
    private HtmlCommandLink commandLink2;
    private HtmlOutputText outputText3;
    List<ServiceRequests> requestList = new ArrayList();
```

The data for this variable is retrieved from the session facade, and its accessors can be used by components in the page.

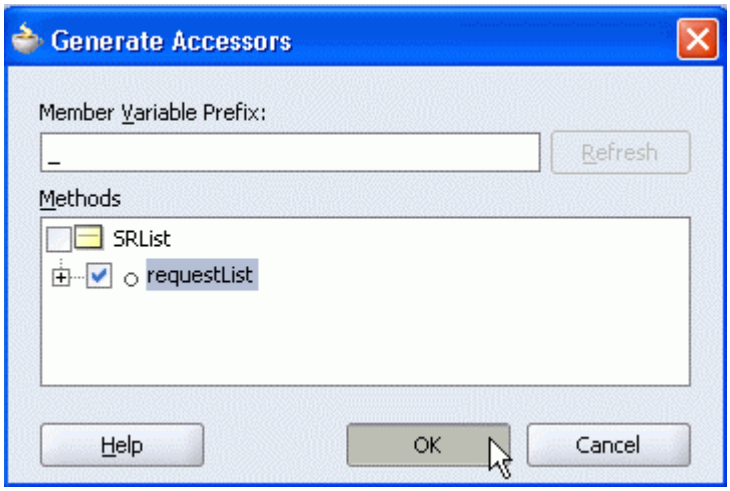
4. Press **[Alt]+[Enter]** when prompted to import:

```
java.util.List
org.srdemo.persistence.ServiceRequests
java.util.ArrayList
```

5. Right-click within the code editor window and select **Generate Accessors** from the context menu.



6. In the Generate Accessors dialog box, select the check box next to the **requestList** variable that you just added, and then click **OK**.



You can scroll to the bottom of the file to see the accessors that were created. In the `getRequestList()` method, locate the following line:

```
return requestList;
```

Replace this line with the following lines:

```
    // Get the Service Locator's instance
    ServiceLocator serviceLocator = null;
    try {
        serviceLocator = serviceLocator.getInstance();
        ServiceRequestFacadeLocal srLocal =
(ServiceRequestFacadeLocal)serviceLocator.
getFacadeService("java:comp/env/ejb/ServiceRequestFacade");
        FacesContext ctx = FacesContext.getCurrentInstance();
        Application app = ctx.getApplication();
        ValueBinding curUser =
app.createValueBinding("#{UserInfo.currentUser}");
```

```
        Users currentUser = (Users)curUser.getValue(ctx);
        return srLocal.findServiceRequests(currentUser,
          currentUser);
    } catch (NamingException e) {
        // TODO
        e.printStackTrace();
    }
    return null;
```

```
    public List<ServiceRequests> getRequestList() {
        // Get the Service Locator's instance
        ServiceLocator serviceLocator = null;
        try {
            serviceLocator = serviceLocator.getInstance();
            ServiceRequestFacadeLocal srLocal =
        (ServiceRequestFacadeLocal)serviceLocator.
        getFacadeService("java:comp/env/ejb/ServiceRequestFacade");
            FacesContext ctx = FacesContext.getCurrentInstance();
            Application app = ctx.getApplication();
            ValueBinding curUser =
        app.createValueBinding("#{UserInfo.currentUser}");
            Users currentUser = (Users)curUser.getValue(ctx);
            //return srLocal.findSRbyUser(currentUser.getEmail());
            return srLocal.findServiceRequests(currentUser,
              currentUser);
        } catch (NamingException e) {
            // TODO
            e.printStackTrace();
        }
        return null;
    }
}
```

The `// TODO` comment in the code creates a task that is listed in the Tasks window that you can display by selecting View → Tasks Window. You can add to code a line that begins with `// TODO` when you need to complete a task later (such as, in this case, adding an exception handler for the naming exception). Any text that you enter after the comment appears as a description in the Tasks window. In this tutorial, you do not actually go back and add the exception handler. But in a real development situation, this would serve as a reminder to do so.

7. Press **[Alt]+[Enter]** when prompted to add several import statements (for `Application`, choose to import `javax.faces.application.Application`).

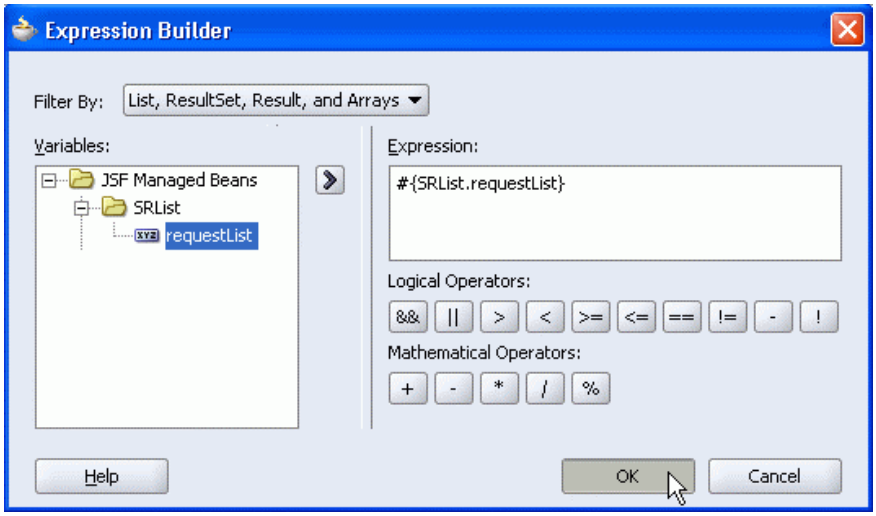
8. Click **Save All** to save your work.

Displaying Data

Now that you have added the named query and the backing bean code, you can add data components to the JSP to utilize this code to retrieve the applicable data. To add components to display this read-only data, perform the following steps:

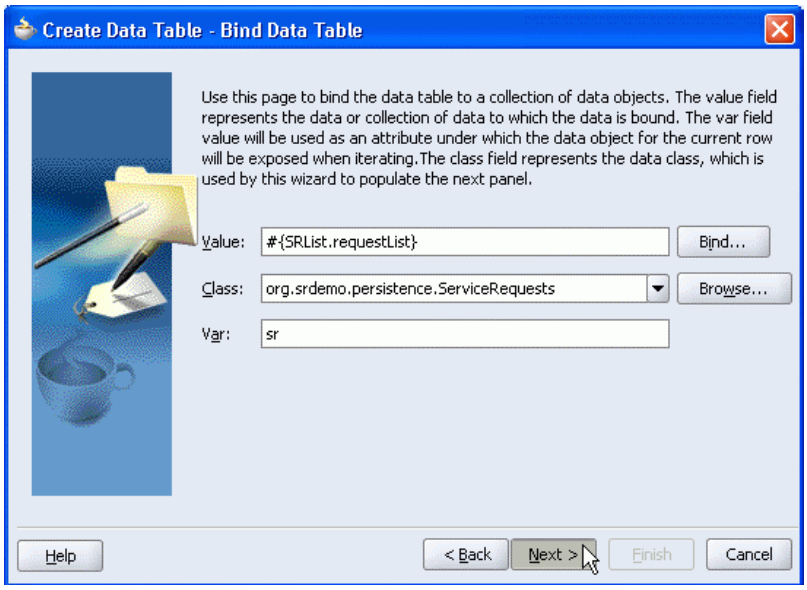
1. In the Applications Navigator, expand **ViewController**, **Web Content**, and **app**.

2. Double-click **SRList.jsp** to open it in the editor and then click the Design tab.
3. From the Component Palette drop-down list, select **JSF HTML** and then drag the **Data Table** component to the line just above the Back to Home link on the page in the visual editor.
4. If the Welcome page of the Create Data Table wizard appears, click **Next**.
5. Select **Bind the Data Table Now** in the Binding window of the wizard, and then click **Next**.
6. In the Bind Data Table page of the wizard, click **Bind**.
7. In the Expression Builder window, expand **SRList**, select **requestList**, and click the **right arrow** to add it to the Expression. Then click **OK**.

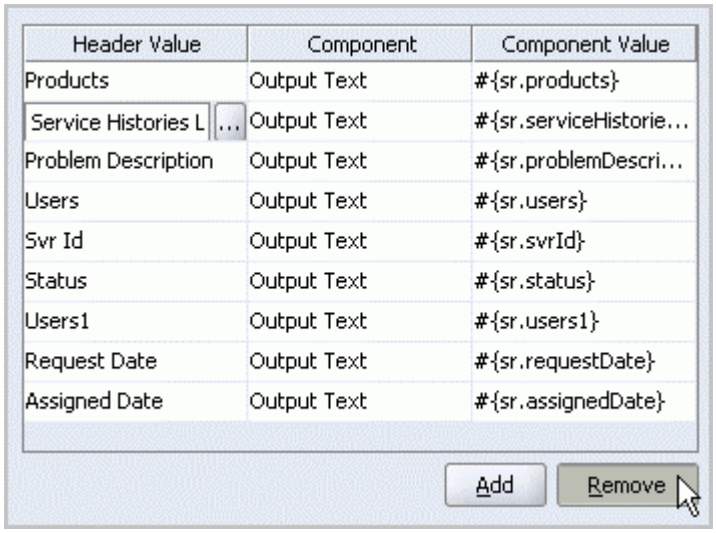


8. Continuing in the Bind Data Table page of the wizard, enter the following values and then click **Next**:

Field	Value
Class	`org.srdemo.persistence.ServiceRequests`
Var	`sr`



9. On the Header and Row Data page of the wizard, select the table column with the header value **Service Histories List** and click **Remove**.

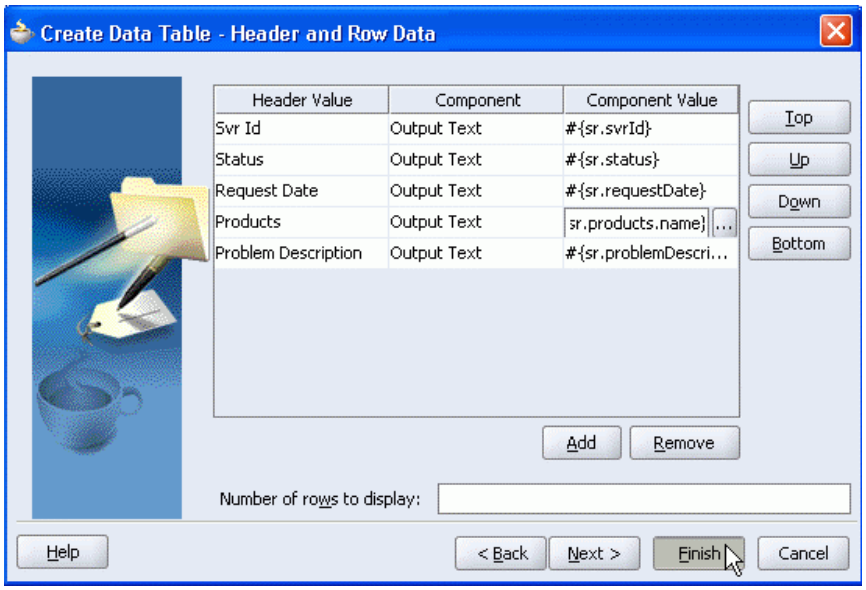


10. In a similar fashion, remove the table columns with the header values **Users**, **Users1**, and **Assigned Date**.

11. By using the buttons at the right of the window, arrange the remaining table columns in the following order:

    `Svr Id`
    `Status`
    `Request Date`
    `Products`
    `Problem Description`

12. Change the Component Value for the Products row to **`#{sr.products.name}`**, and then click **Finish**.



13. Click **SaveAll** to save your work.

Passing Data Between Pages

Now you add a method to SRList.java to forward the selected Service Request object to the SREdit.jsp page. Perform the following steps:

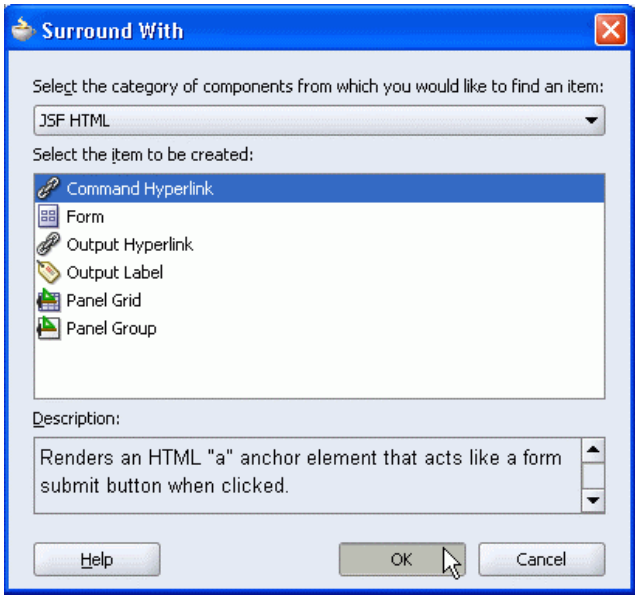
1. In the Applications Navigator, expand **ViewController**, **Application Sources**, **org.srdemo.view**, and **backing**.
2. Double-click **SRList.java** to open it in the editor.
3. Add the following method to forward the selected Service Request object to the SREdit.jsp or SRTEdit.jsp pages:

```
public String editLinkAction()
{
    /* pull out the currently selected service request */
    ServiceRequests editSR =
    (ServiceRequests)this.getDataTable1().getRowData();
    FacesContext ctx = FacesContext.getCurrentInstance();
    Application app = ctx.getApplication();
    ValueBinding curUser =
    app.createValueBinding("#{UserInfo.userRole}");
    Object obj = curUser.getValue(ctx);
    String roleString = obj.toString();

    if (roleString.equals("user")) {
        ValueBinding binding =

ctx.getApplication().createValueBinding("#{SREdit.serviceRequest}");
        binding.setValue(ctx,editSR);
        return "update";
    }
    else {
```

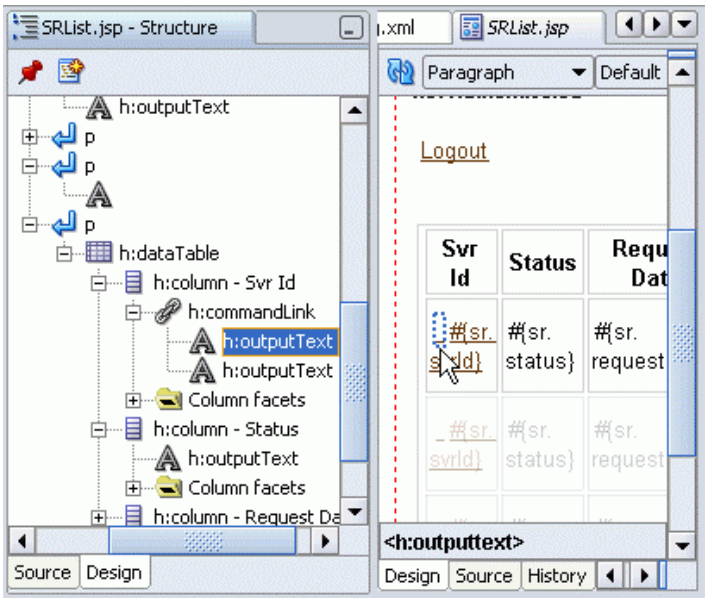
```
                    ValueBinding binding =
ctx.getApplication().createValueBinding("#{SRTEdit.serviceRequest}");
                    binding.setValue(ctx,editSR);
                    return "techupdate";
                }
            }
```

The value bindings that are created by this method are used in the next chapter when you define the SREdit and SRTEdit pages. Also note that the method returns either `update` or `techupdate`, depending on whether or not the user has the `user` role. These strings correspond to navigation cases that you defined in the JSF Navigation Diagram to navigate to the SREdit and SRTEdit pages, respectively.

4. Return to the **SRList.jsp** by clicking the tab for the page (or opening it as previously described). Click the **Design** tab for the page.
5. Change the `svrId` field to a link that passes the selected row to another page for editing. To accomplish this, either in the visual editor or the Structure window, right-click the output text component **`#{sr.svrId}`** and select **Surround With...** from the context menu.
6. In the Surround With dialog box, select **Command Hyperlink** and click **OK**.



7. In the Insert Command Hyperlink dialog box, leave the Value blank and click **OK**.
8. In the Structure window, remove the extra **outputText** component that was created (in front of the #{sr.svrId} output text) by selecting it and pressing **[Delete]**.



9. In the Structure window, select the **h:commandLink** that you just created. In the Property Inspector select the Action attribute and then select **editLinkAction()** from the drop-down list.

10. Click **SaveAll** to save your work.

Specifying the Location of the Session Facade

The SRList page is the first page that you have developed that uses the session facade. Before a page can use the session facade on the Web, you must specify the local home for the session facade in the Web deployment descriptor (`web.xml`) for the application. The declaration consists of:

- An optional description
- The EJB reference name used in the code of the Web application that references the enterprise bean
- The expected type of the referenced enterprise bean
- The expected local home and local interfaces of the referenced enterprise bean
- Optional ejb-link information (used to specify the referenced enterprise bean)

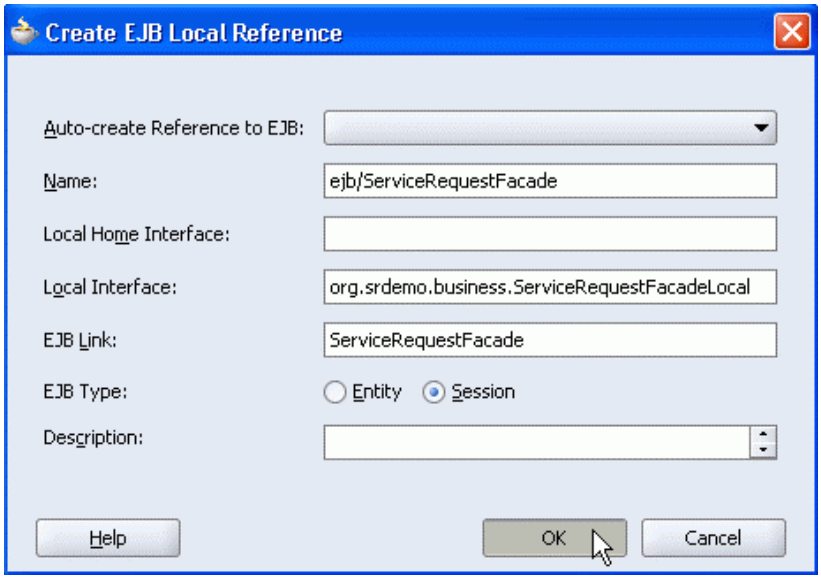
To configure the Web deployment descriptor to locate the session facade, perform the following steps:

1. In the Applications Navigator, expand the **ViewController** project and then expand **Web Content** and **WEB-INF**.
2. Right-click **web.xml** and select **Properties** from the context menu.
3. In the tree at the left of the Web Application Deployment Descriptor window, select **EJB Local References**, and then click **Add** in the right section of the window.

4. In the Create EJB Local Reference dialog box, enter the following values:

Field	Value
Name	`ejb/ServiceRequestFacade`
Local Interface	`org.srdemo.business.ServiceRequestFacadeLocal`
EJB Link	`ServiceRequestFacade`
EJB Type	Select the Session option

Click **OK** and then click **OK** again to dismiss the Web Application Deployment Descriptor window.



5. Double-click the **web.xml** file to open it in the editor.
6. Scroll to the end of the file and delete the following line:

   ```
   <local-home></local-home>
   ```

7. Click **SaveAll** to save your work.

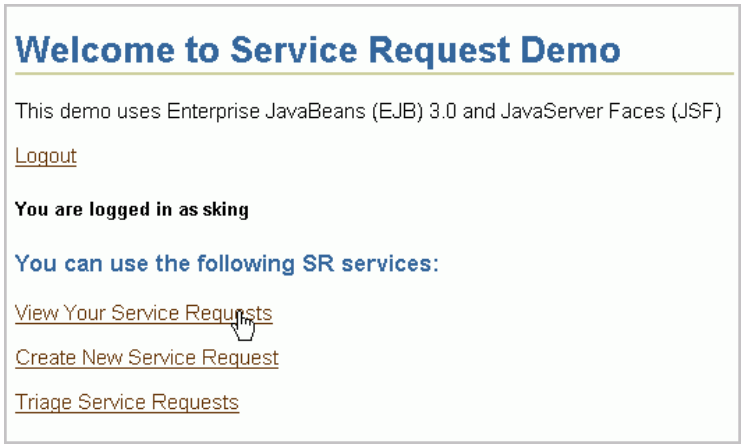
Testing the Update Logic and the UI Components

Now that you have defined list logic and created the necessary UI components to list service requests, you can test the functionality by performing the following steps:

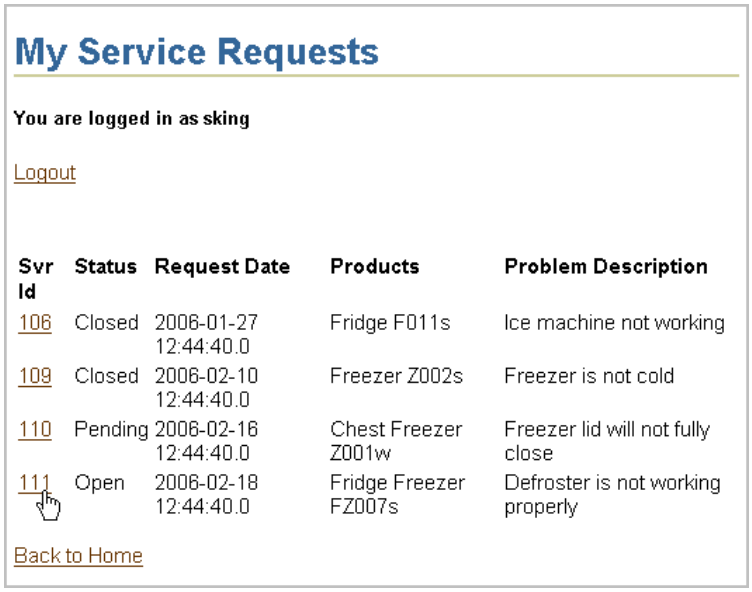
1. In the Applications Navigator, expand the **ViewController** project and the **Web Content** node.
2. Right-click **index.jsp** and select **Run** from the context menu.
3. A browser opens to display the Logon page. Enter the following logon information and click **Sign On**:

Username	Password
sking	welcome

4. On the SRWelcome page, click **View Your Service Requests**.



5. The SRList page displays all service requests that were created by or assigned to the user.



Note that “Svr Id” is a link. This passes the selected object to either the SREdit.jsp page or the SRTEdit.jsp page to be defined in the next chapter. At this point, you’ve defined the method to pass the Service Request object. If you click the link now, you receive an error indicating that the page does not yet exist. You can click the Back to Home link to return to the Welcome page.

Summary

In this chapter, you created the page that enables users to view their service requests. To accomplish this, you performed the following key tasks:

- Created the list page that contains components that:

  - Display the name of the logged-in user
  - Log out of the application
  - Navigate to the home page of the application
- Added a named query to retrieve the service requests that were created by or assigned to the current user
- Exposed the named query to the session facade
- Updated the backing bean to call the session facade method that retrieves the data
- Added a data table to the page that displays data retrieved by the method in the backing bean
- Added logic to the backing bean to package a Service Requests object and forward it for editing
- Added a link to each row of the data table on the page to call the code from the backing bean, passing the selected Service Requests object for editing
- Modified the Web deployment descriptor so that the session facade can be located at run time
- Tested the update logic and the UI components

9

Developing Pages to Edit Records

You have created the page that lists a user's service requests. Now you create pages that enable the user to edit service requests. There are two such pages in this application: SREdit for customers and SRTEdit for technicians and managers.

Users access the edit pages from the list page by clicking a link that passes the data pertaining to the selected service request to the edit page. When you created the SRList page, you defined the logic for passing that data, along with the logic to determine which edit page to display based on roles.

If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh08.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

- Edit Page Overview
- Creating the Edit Page for Customers
- Adding Components to Display Data
- Adding Data Entry Components
- Persisting or Canceling an Update
- Displaying Detail Records
- Conditionally Hiding Records
- Developing the Edit Page for Staff
- Testing the Update Logic and the UI Components
- Summary

Edit Page Overview

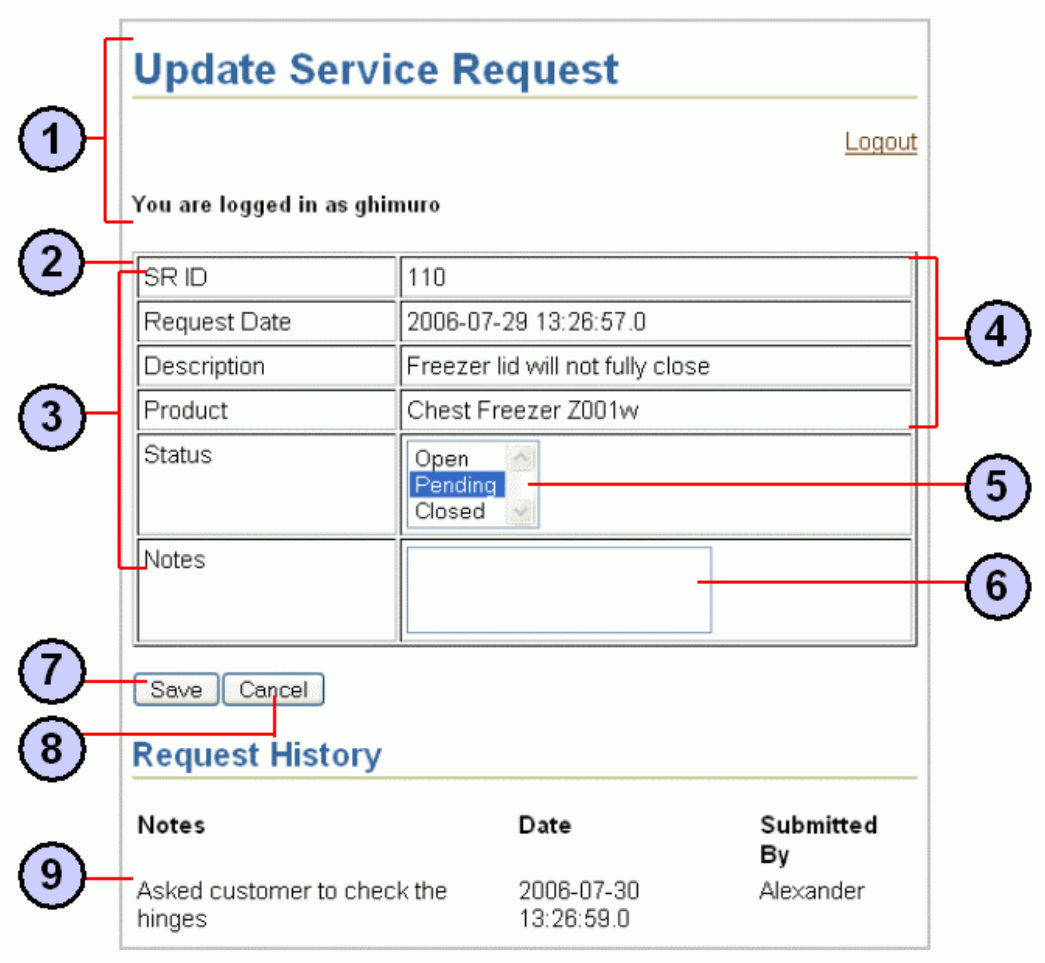
The SREdit page enables customers to edit existing service requests. This page is accessed from the SRList page, where users can view their service requests and select one of them to update.

SREdit enables customers to change the status of their service requests by selecting a new status from a list. They can also add a note to the service request and save the note as a service history record. The page is a master-detail page that also displays the associated history records for the service request.

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

If the user chooses to commit the data, the backing bean code updates the service request record. The code also inserts a related service history record into the database. Navigation to the SRList page then occurs.

The following screenshot depicts the features of the Edit page:



1. Each application page contains a title, information about the logged in user, and a Logout link.
2. An HTML table is used to lay out the JSF components.
3. Labels are plain text that is entered into the left column of the HTML table.
4. Output text components display four of the Service Request fields. You bind these fields to the data by adding a `ServiceRequests` object to the page's backing bean and then accessing the

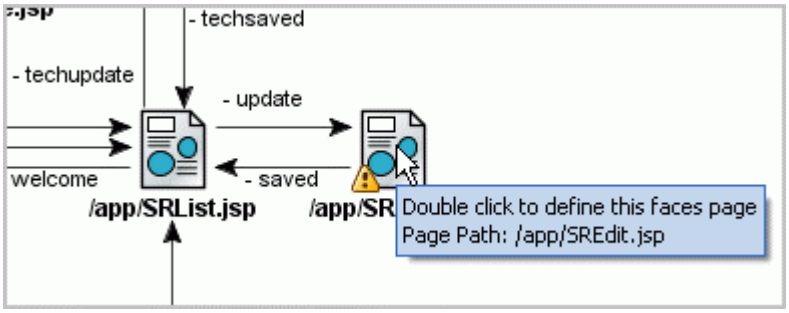
data by using Expression Language. Users cannot edit data that is displayed in output text components.

5. A JSF Listbox component displays hard-coded values for the status of the service request. You use Expression Language to bind this component to the `status` attribute of the backing bean's `ServiceRequests` object. The user can select a new status to update the request.
6. A JSF Input Textarea component enables the user to enter multiple lines of text. You define backing bean code to take the value in the input text area and assign it to the `notes` attribute of an associated `ServiceHistories` entity.
7. A Save button calls code that you add to the backing bean to persist the data to the underlying database tables. The bean is set to session scope so that the data is available for all pages in the session. This code in the bean returns a string value that corresponds to one of the From outcomes in a navigation rule that you defined earlier.
8. A Cancel button has an action set to one of the From outcomes in a navigation rule that you defined earlier. This enables navigation to occur without executing the code to persist the data.
9. Associated service history records are displayed in a JSF Data Table that is bound to the list of service histories associated with the backing bean's `ServiceRequests` object.

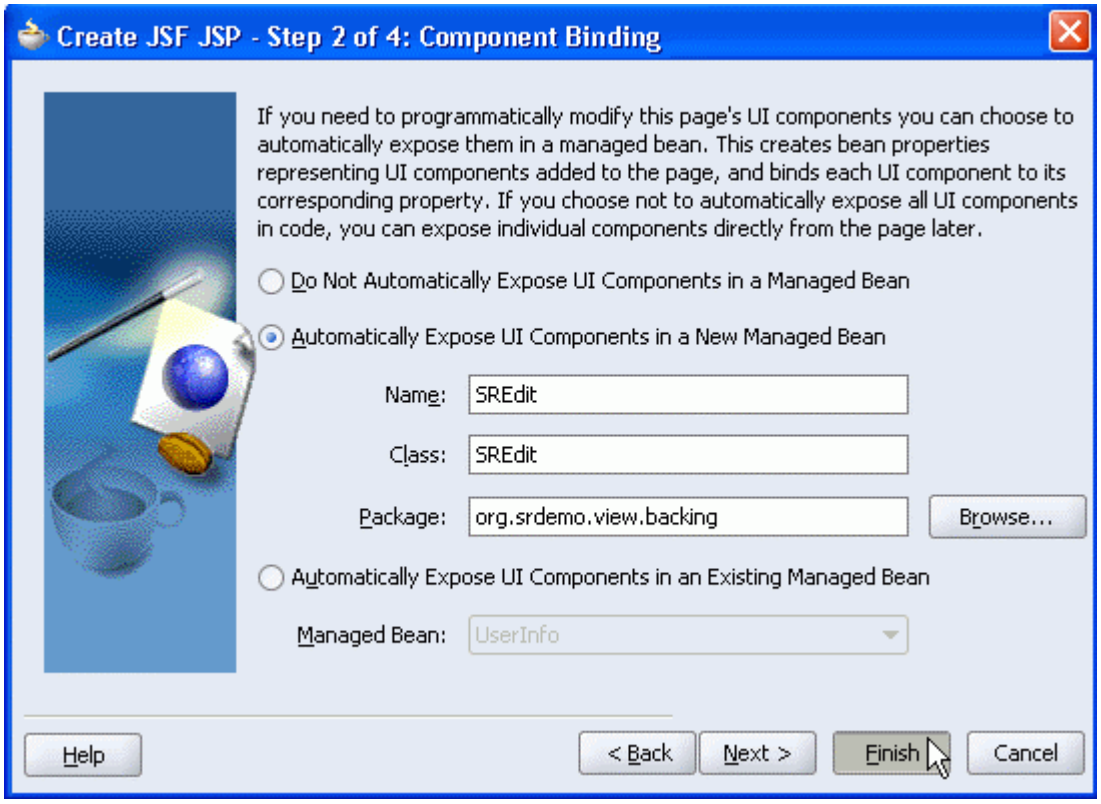
Creating the Edit Page for Customers

To define the SREdit page, perform the following steps:

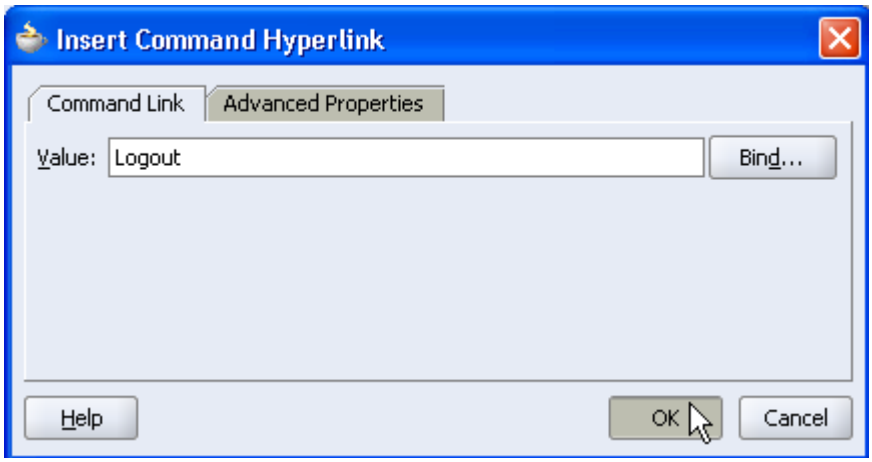
1. Open **faces-config.xml** in the editor if it is not already open, or click its tab if it is open. (To open the file, right-click the **ViewController** project and select **Open JSF Navigation** from the context menu.) Click the **Diagram** tab.
2. Double-click the **/app/SREdit.jsp** icon to invoke the Create JSF JSP wizard.



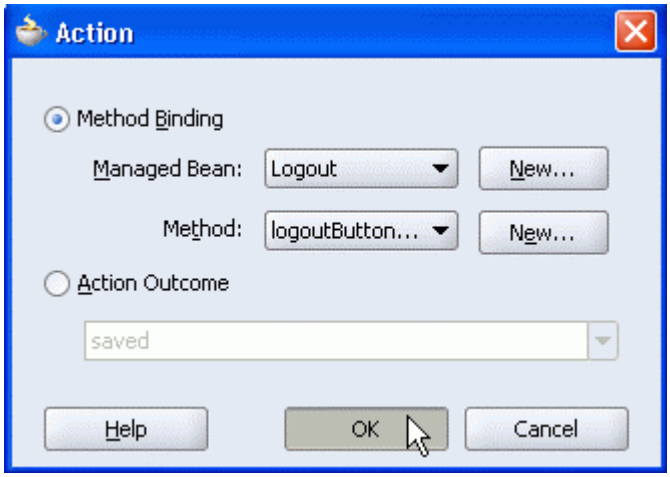
3. If the Welcome page of the wizard appears, click **Next**.
4. On the JSP File page of the wizard, click **Next**.
5. On the Component Binding page of the wizard, select the option **Automatically Expose UI Components in a New Managed Bean**. Change the Name of the managed bean to **`SREdit`** and the Package to **`org.srdemo.view.backing`**. Click **Finish**.



6. The new page opens in the editor. Enter the text **`Update Service Request`** and then select **Heading 1** from the Block Format drop-down list.
7. From the Component Palette, select **JSF HTML** from the drop-down list and drag the **Messages** component to the next line on the page in the visual editor.
8. Place the cursor after the Messages component and press **[Enter]**, and then click the **Command Hyperlink** component in the Component Palette.
9. In the Insert Command Hyperlink dialog box, enter the Value **`Logout`** and click **OK**:

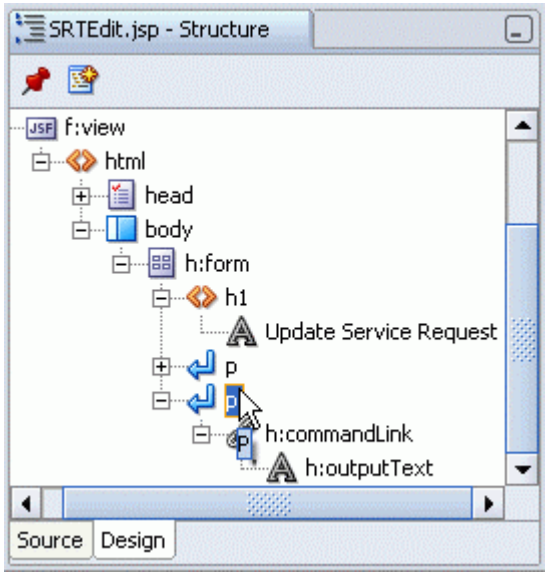


10. In the Property Inspector for the command hyperlink, enter `#{}` in the Action property and press **[Enter]**. Then click back into the Action property. Click the **ellipsis** ... at the end of the field.
11. In the Action dialog box, ensure that the **Method Binding** option is selected. Select **Logout** from the Managed bean drop-down list, and select **logoutButton_action** from the Method drop-down list. Click **OK**.

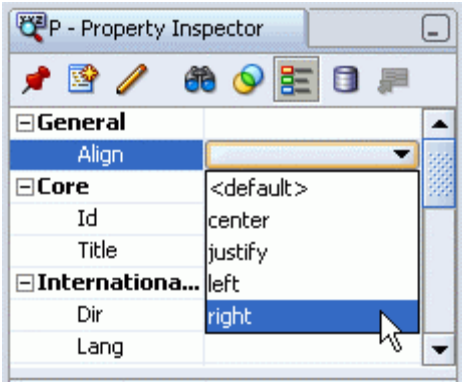


This ensures that the Logout link executes the code in the Logout managed bean that you created in a previous chapter.

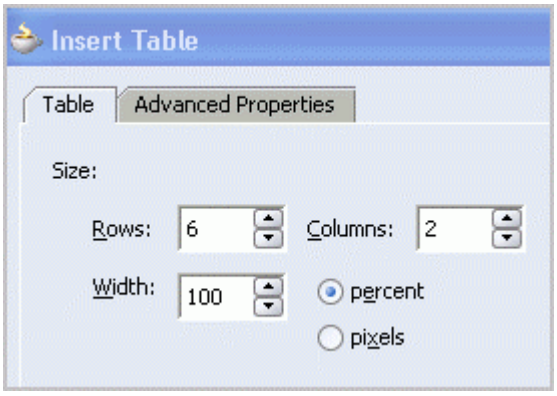
12. In the Structure window, select the **paragraph node** p that contains the commandLink that you just defined.



13. In the Property Inspector, set the Align property of the paragraph to `right` by selecting that value from the drop-down list.



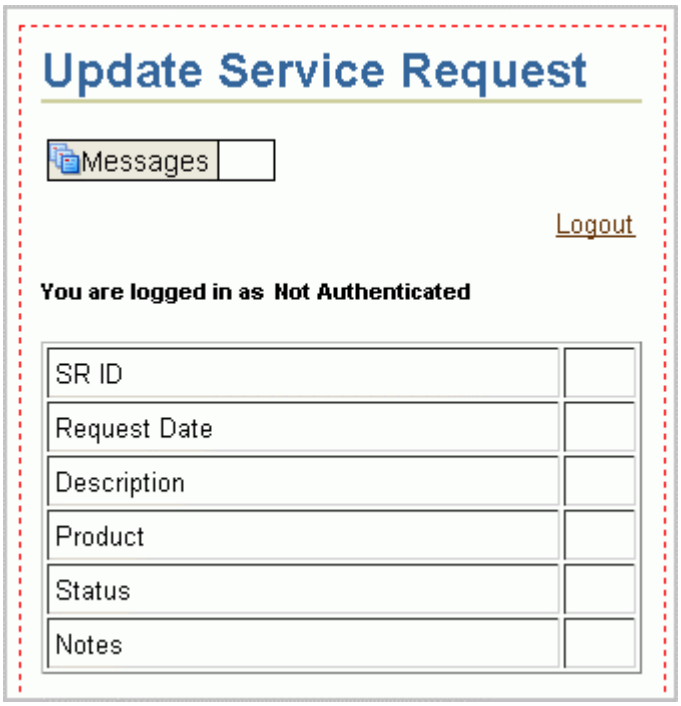
14. In the editor, on the next line enter the text `You are logged in as`, and then press **[Space]**. From the Block Format drop-down list, select **Heading 5**.
15. In the Component Palette, click the **Output Text** component.
16. An output text field is created in the editor. In the Property Inspector for the output text, select the **Value** property and click **Bind to Data** .
17. In the Value dialog box, expand **JSF Managed Beans** and **UserInfo**. Select **userName** and click the right arrow to add it to the expression on the right. Click **OK**. This causes the application to display the name of the authenticated user at run time.
18. Select **HTML Common** from the Component Palette drop-down list and then drag the **Table** component to the next line in the editor to create a table on the page.
19. In the Insert Table dialog box, set the number of rows to `6`, the number of columns to `2`, and the width to `100` with the **percent** option selected. Then click **OK**.



20. In the left column, enter the following phrases in each of the rows:

```
SR ID
Request Date
Description
Product
Status
Notes
```

21. Select **CSS** from the Component Palette drop-down list and then select **JDeveloper**. Your page should now look similar to the following screenshot:



22. Click **Save All** to save your work.

Displaying Data

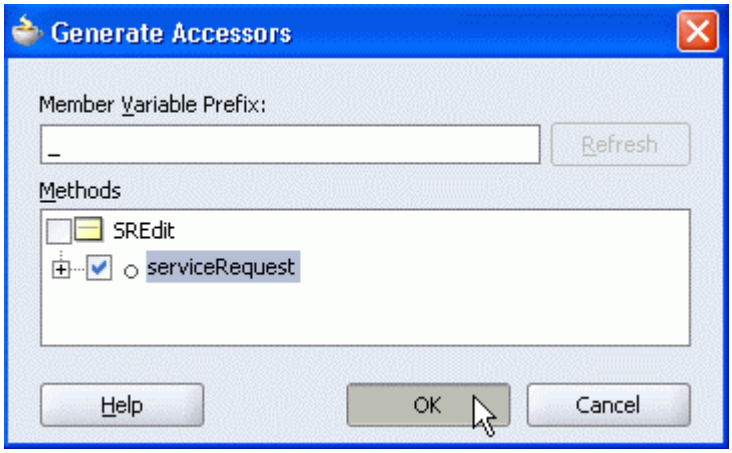
The SREdit page has some fields that are not editable; they only display data. To add output text components to display this read-only data, perform the following steps:

1. Add a new attribute to the backing bean for the service request. In the Applications Navigator, expand **ViewController**, **Application Sources**, **org.srdemo.view**, and **backing**.
2. Double-click **SREdit.java** to open it in the editor.
3. Add the following attribute:

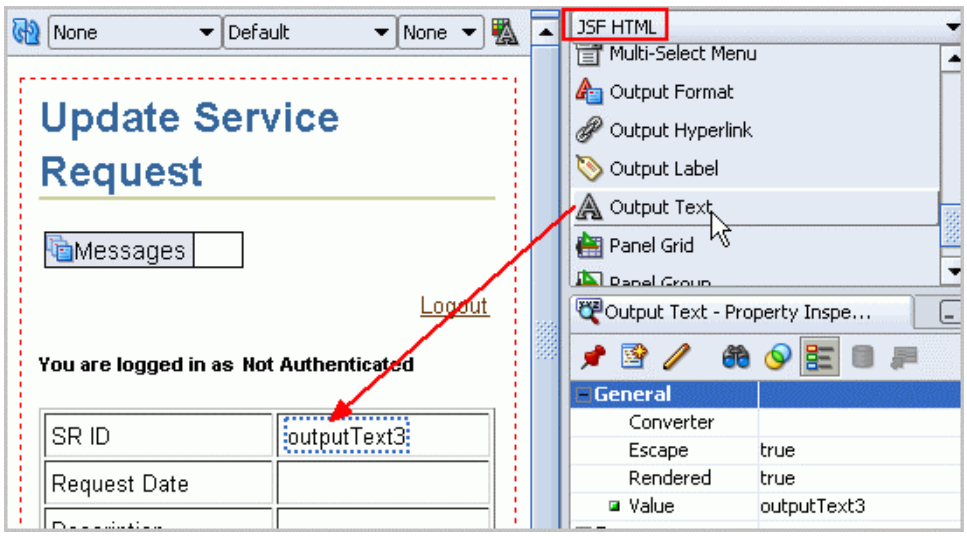
```
ServiceRequests serviceRequest;
```

```
import javax.faces.component.html.HtmlCommandLink;
import javax.faces.component.html.HtmlForm;
import javax.faces.component.html.HtmlOutputText;

public class SREdit {
    private HtmlForm form1;
    private HtmlCommandLink commandLink1;
    private HtmlOutputText outputText1;
    private HtmlOutputText outputText2;
import org.srdemo.persistence.ServiceRequests (Alt-Enter)
    ServiceRequests serviceRequest;

    public void setForm1(HtmlForm form1) {
        this.form1 = form1;
```

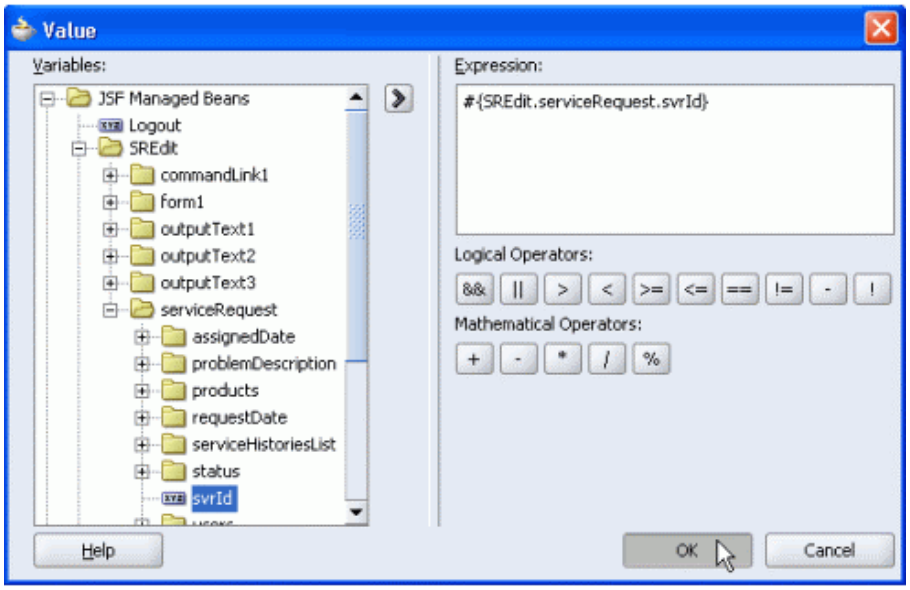
4. Press **[Alt]+[Enter]** when prompted to import `org.srdemo.persistence.ServiceRequests`.
5. Right-click in the editor window and select **Generate Accessors** from the context menu.
6. In the Generate Accessors dialog box, select the check box next to **serviceRequest** and click **OK**.



7. Now that you have added code to the backing bean to access service request data, you can add databound components to the SREdit page. Switch to **SREdit.jsp** by clicking its tab in the editor. Select **JSF HTML** from the Component Palette drop-down list and drag the **Output Text** component to the top-right cell of the table in the visual editor.



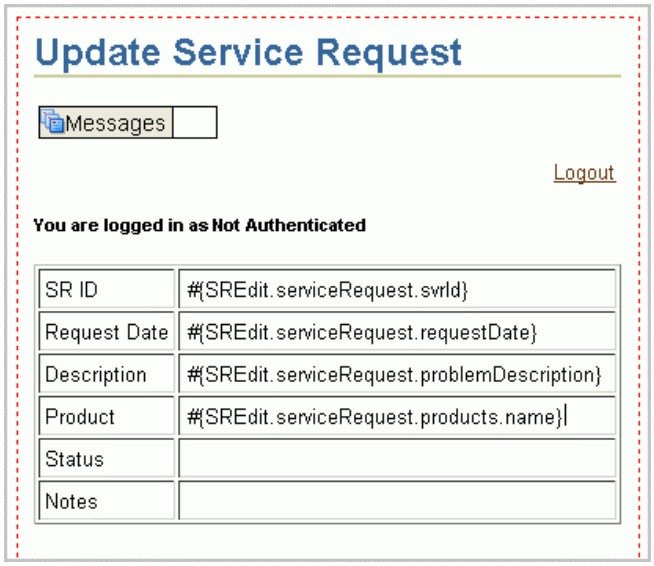
8. In the Property Palette, select the **Value** property and then click **Bind to Data** .
9. In the Value dialog box, expand **JSF Managed Beans**, **SREdit**, and **serviceRequest**. Select **svrId** and click the **right arrow** to add it to the Expression. Then click **OK**.



10. In a similar fashion, add the following to the next three cells beneath the svrId cell:

Row Label	Component	Value
Request Date	Output Text	`#{SREdit.serviceRequest.requestDate}`
Description	Output Text	`#{SREdit.serviceRequest.problemDescription}`
Product	Output Text	`#{SREdit.serviceRequest.products.name}` (Expand `products` and shuttle `name` to the Expression.)

11. Click **Save All** to save your work. Your page should now look like the following screenshot:



Adding Data Entry Components

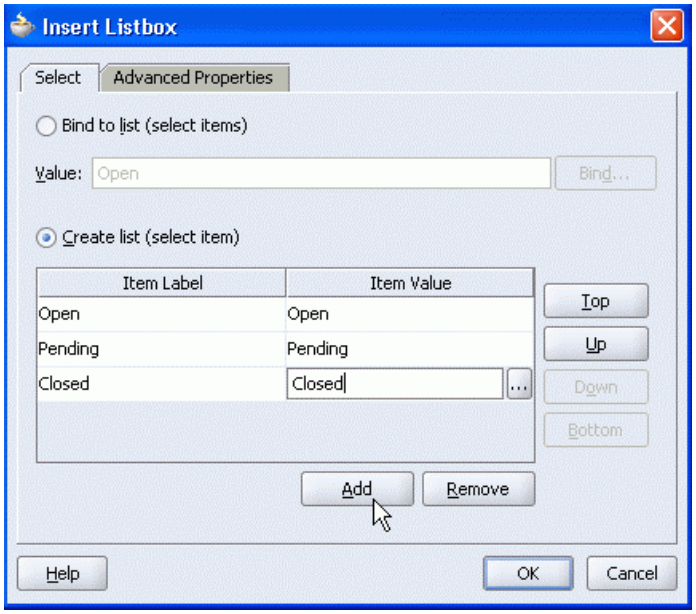
Now you add UI components that ensure data accuracy and facilitate data entry. In this section you create components and add the code that enables users to:

- Select a status
- Enter a note

Enabling Users to Select a Status

To ensure that users enter a correct status, you implement a select list. Because there are only three possible values for the status, you hard-code the values into three separate select items. To add a select list to the page, perform the following steps:

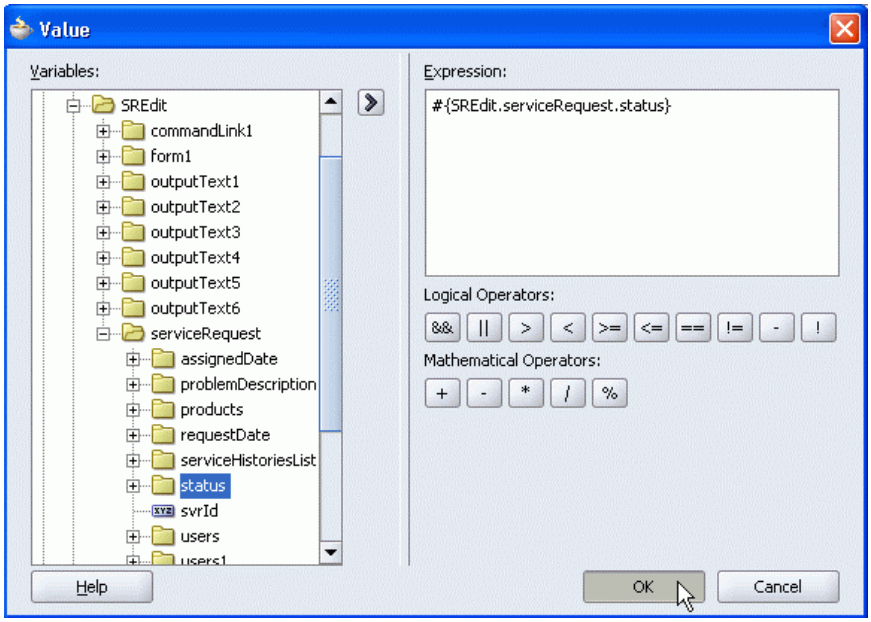
1. In the Component Palette, select **JSF HTML** from the drop-down list and drag the **Listbox** component to the right cell of the row that you previously labeled `Status`.
2. Select the **Create list (select item)** option and click **Add**, and then enter the Item Label **`Open`** and the Item Value **`Open`**. This creates one item in the select list with the label and value both set to "Open."
3. Click **Add** twice more to add two additional list items, one with the label and value **`Pending`** and another with the label and value **`Closed`**.



4. Click the **Advanced Properties** tab.

5. Click the **Value** property and then click **Bind to Data** .

6. In the Value dialog box, expand **JSF Managed Beans**, **SREdit**, and **serviceRequest**. Select **status** and click the right arrow to shuttle it to the Expression, and then click **OK**.



This associates this list item's value with the `status` of the `serviceRequest` variable in the `SREdit` managed bean.

7. Click **OK** to dismiss the Insert Listbox dialog box.

8. Click **Save All** to save your work.

Enabling Users to Enter a Note

Next you add the component to enable users to input a note that will be part of a new service histories record pertaining to this update. To enable text input, perform the following steps:

1. From the Component Palette, drag the **Input Textarea** component to the right cell of the row that you previously labeled `Notes` in the visual editor. This creates an area where users can enter multiple lines of text.

   You do not need to bind the value to data, because the backing bean code that you define in the next section takes the value in the input text area and assigns it to the `notes` attribute of the `ServiceHistories` entity.

2. Click **Save All** to save your work.

Persisting or Canceling the Update

Now you add code to the managed bean for the page that assigns the input values to the entity objects and then commits the data to the database. You also add a button to call that code and another button to cancel the update. Finally, you change the scope of the managed bean to `session` so that updated information can be displayed throughout the session.

Perform the following steps:

1. Open or switch to **SREdit.java** (**ViewController** > **Application Sources** > **org.srdemo.view** > **backing**).
2. Add the following method that assigns the input fields to properties in the entity beans and then persists (commits) the data to the database:

```
public String updateSR(){
    FacesContext ctx = FacesContext.getCurrentInstance();
    Application app = ctx.getApplication();
    ValueBinding curUser =
        app.createValueBinding("#{UserInfo.currentUser}");
    Users currentUser = (Users)curUser.getValue(ctx);
    ServiceHistories historynotes = new ServiceHistories();
    historynotes.setServiceRequests(serviceRequest);
    historynotes.setNotes(getInputTextarea1().getValue().
        toString());
    historynotes.setLineNo(null);
    historynotes.setSvhType("Customer");
    historynotes.setSvhDate(null);
    historynotes.setUsers(currentUser);
    serviceRequest.addServiceHistories(historynotes);

    ServiceLocator serviceLocator = null;
    try {
        serviceLocator = serviceLocator.getInstance();

        ServiceRequestFacadeLocal srLocal =
            (ServiceRequestFacadeLocal)
            serviceLocator.getFacadeService
            ("java:comp/env/ejb/ServiceRequestFacade");
        srLocal.mergeEntity(serviceRequest);
        return "saved";
    } catch (NamingException e) {
        // TODO
```

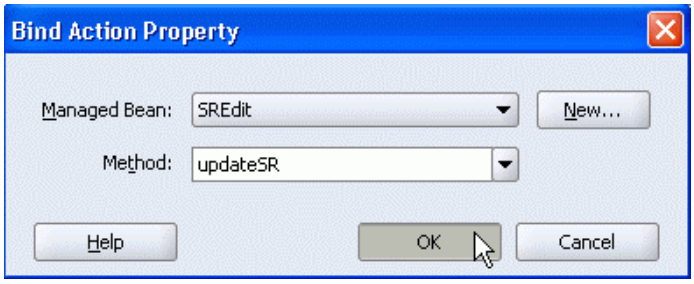
```
            e.printStackTrace();
        }
        return "saved";
    }
```

Note that this method returns the string `saved`, which is the name of the navigation case (pointing from this page to the SRList page) that you defined earlier in the JSF Navigation Diagram.

3. Press [Alt]+[Enter] when prompted to import:

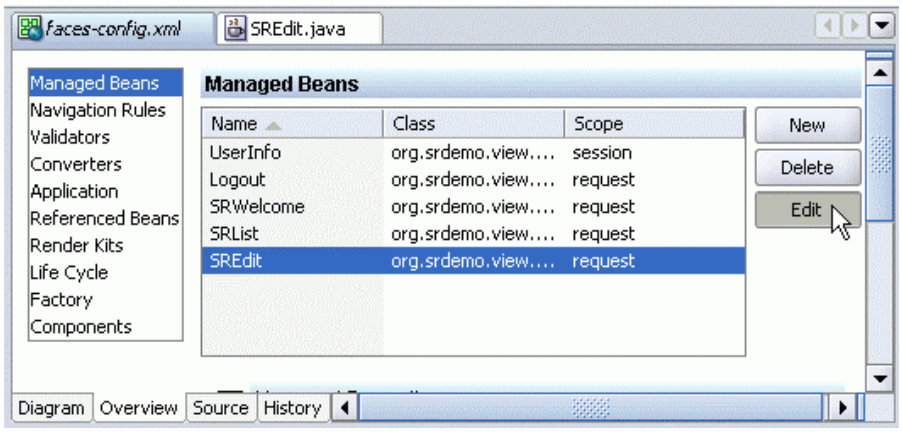
```
javax.faces.context.FacesContext
javax.faces.application.Application
javax.faces.el.ValueBinding
org.srdemo.persistence.Users
org.srdemo.persistence.ServiceHistories
org.srdemo.view.util.ServiceLocator
org.srdemo.business.ServiceRequestFacadeLocal
javax.naming.NamingException
```

4. Switch to SREdit.jsp in the editor. Place the cursor after the table and press [Enter] to create a blank line.
5. In the Component Palette, select JSF HTML from the drop-down list and click Command Button.
6. In the Property Palette, set the Value property to Save.
7. Double-click the Save button in the editor, and in the Bind Action Property dialog box, select updateSR from the Method drop-down list. Click OK.

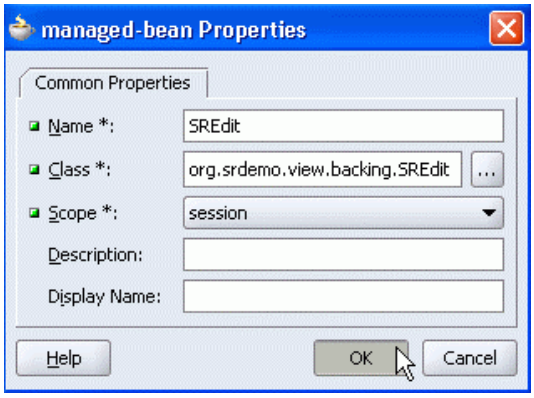


Note: The `updateSR()` method in the SREdit managed bean returns the string `saved`, which is the name of the navigation case that you defined in the `faces-config.xml` file to navigate to the SRList page.

8. In the editor for SREdit.jsp, drag the Command Button component from the Component Palette to the right of the Save button on the page.
9. In the Property Palette, set the Value property to **`Cancel`** and set the Action property to **saved** so that the user navigates to the SRList page, but without executing the update code in the backing bean.
10. Now you change the scope of the managed bean to `session` so that changes can be seen on other pages of the application. Open (right-click **ViewController** and select **Open JSF Navigation**) or switch to the **faces-config.xml** file in the editor.
11. Click the **Overview** tab, select **SREdit**, and click **Edit**.



12. In the managed-bean Properties dialog box, select **session** from the Scope drop-down list and click **OK**.



13. Classes for bean with `session` scope must implement the `Serializable` interface if they are part of a distributable Web application. Open or switch to **SREdit.java** and locate the following line:

    ```
    public class SREdit {
    ```

    Change it to:

    ```
    public class SREdit implements Serializable{
    ```

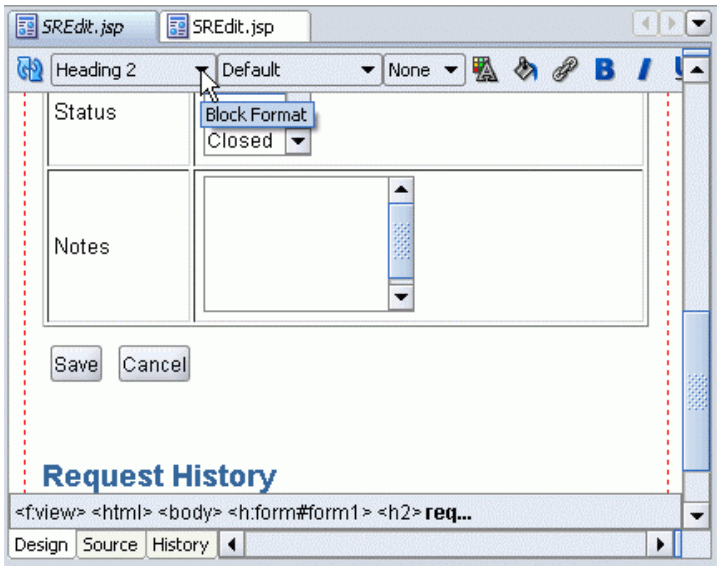
    Press **[Alt]+[Enter]** when prompted to import **`java.io.Serializable`**.

14. Click **Save All** to save your work.

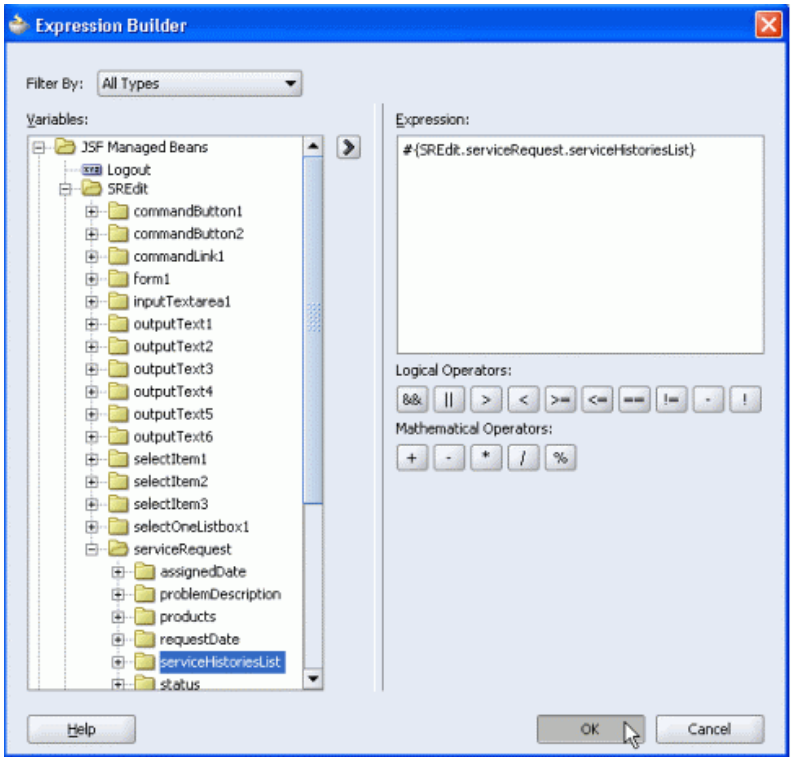
Displaying Detail Records

On the same page in which customers edit service requests, you want to display any associated service history records so that customers can see the notes that have been added previously. To display the detail records, perform the following steps:

1. With **SREdit.jsp** open in the editor, skip a line after the command buttons and enter the text **`Request History`**. Then select **Heading 2** from the Block Format drop-down list.



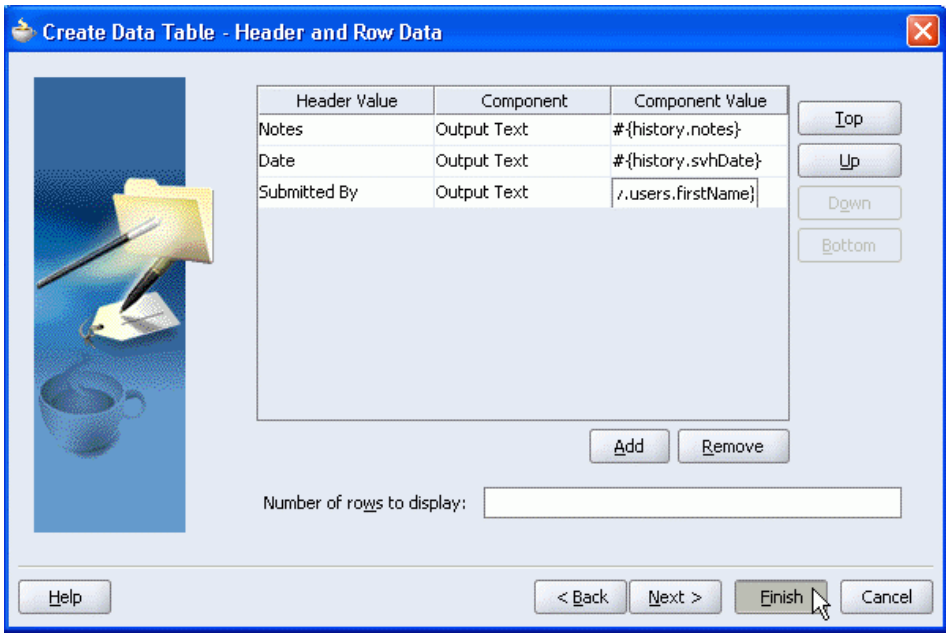
2. Place the cursor at the end of the horizontal line under the Request History heading and press **[Enter]** to create a new line.
3. In the Component Palette, select **JSF HTML** from the drop-down list, and then drag **Data Table** to the blank line that you just created to place a data table on the page.
4. If the Welcome page of the Create Data Table Wizard appears, click **Next**.
5. On the Binding page of the wizard, select the option **Bind the Data Table Now** and click **Next**.
6. On the Bind Data Table page of the wizard, click **Bind** next to the Value field.
7. In the Expression Builder window, select **All Types** from the Filter By drop-down list.
8. In the tree at the left, expand **JSF Managed Beans**, **SREdit**, and **serviceRequest**.
9. Select **serviceHistoriesList** and click the **right arrow** to shuttle it to the Expression field, and then click **OK**.



10. On the Bind Data Table page of the wizard, change the Class to **`org.srdemo.persistence.ServiceHistories`**. In the Var field, enter **`history`**, and then click **Next**.

11. On the Header and Row Data page of the wizard, you specify the columns of the data table. You need only three columns in the data table: **Users**, **Notes**, and **Svh Date**. Remove all of the other columns by selecting each in turn and clicking **Remove**. Change the Header Value `Users` to **`Submitted By`**, and change `Svh Date` to **`Date`**. Change the Component Value `Submitted By` to **`#{history.users.firstName}`**. By using the buttons at the right of the window, rearrange the data in the following order, and then click **Finish**:

Header Value	Component	Component Value
`Notes`	Output Text	`#{history.notes}`
`Date`	Output Text	`#{history.svhDate}`
`Submitted By`	Output Text	`#{history.users.firstName}`



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

Conditionally Hiding Records

Some of the Service History records are marked as hidden by staff. These records contain notes of a confidential nature that should not be visible to customers. To conditionally hide these records from view in the SREdit page that is used by customers, perform the following steps:

1. With **SREdit.jsp** open in the editor, select the **`#{history.notes}`** output text, select the **Rendered** property in the Property Inspector, and click **Bind to Data** .
2. In the Rendered property dialog box, enter **`#{history.svhType != "Hidden"}`** in the Expression field and click **OK**.
3. In a similar fashion, set the **Rendered** property to **`#{history.svhType != "Hidden"}`** for the other two output text items in the detail section of the page.
4. Click **Save All** to save your work.

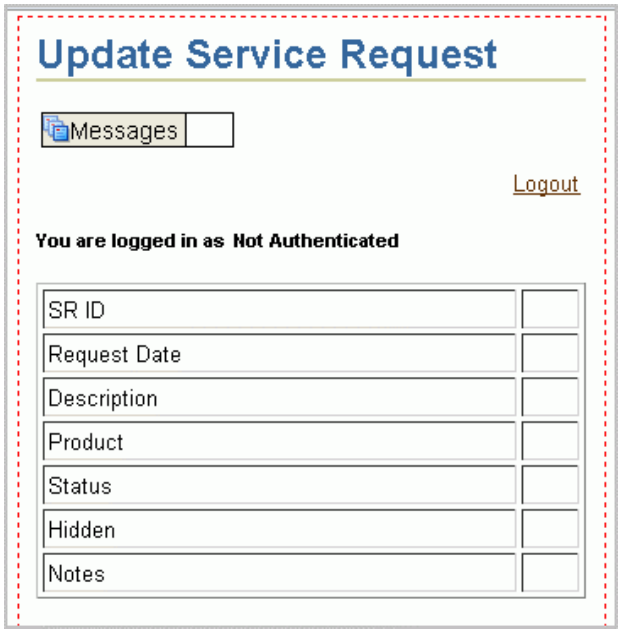
Developing the Edit Page for Staff

The page that the staff uses to edit service requests is almost identical to that used by customers. The main difference is that the staff edit page contains a field indicating that a note should be hidden from customer view, and because staff can see the notes that are hidden from customers, no conditional rendering is needed.

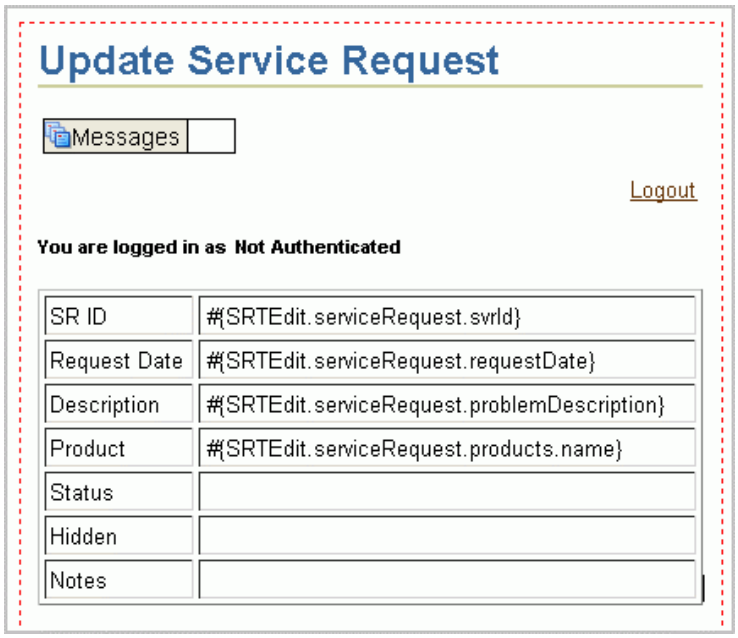
Most of the steps that you need to create the edit page for staff are similar to those you followed in creating the edit page for customers.

1. Perform all steps as in “Creating the Edit Page for Customers,” except that you double-click **/app/staff/SRTEdit** (rather than /app/SREdit) in the JSF Navigation Diagram and you create seven rows in the table (rather than six). The last row should still be labeled **`Notes`**, but label the

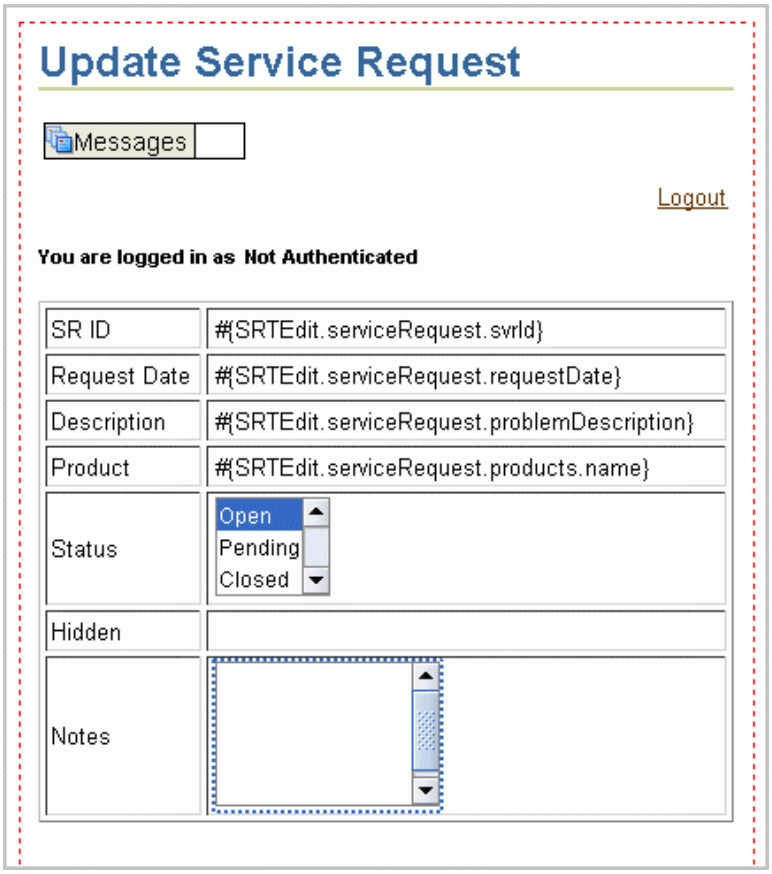
sixth row **`Hidden`**. The page should look similar to the following screenshot:



2. Follow all steps as in "Displaying Data," except that you edit **SRTEdit.java** and **SRTEdit.jsp** (rather than SREdit.java and SREdit.jsp), and all references are to **SRTEdit** (rather than SREdit). When you are finished with this section, the page should look similar to the following screenshot:

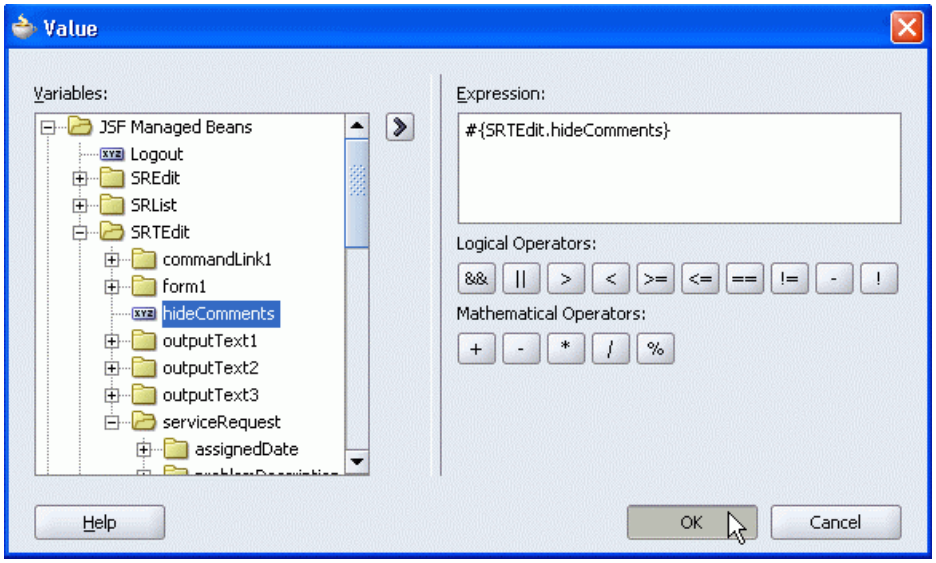


3. Perform all steps in "Adding Data Entry Components," except that you edit **SRTEdit.jsp (**rather than SREdit.jsp), and all references are to **SRTEdit** (rather than SREdit). When you are finished with this section, the page should look similar to the following screenshot:



4. Add the check box to indicate that comments should be hidden from customers. Drag the **Checkbox** component from the Component Palette to the right cell of the row labeled Hidden.
5. Now add code to the backing bean for the page to hold the value of the check box. Open or switch to **SRTEdit.java** in the code editor (**ViewController** > **Application Sources** > **org.srdemo.view** > **backing**).
6. Add the following private variable: `private boolean hideComments = false;`
7. Right-click in the editor and select **Generate Accessors** from the context menu.
8. In the Generate Accessors dialog box, select the check box next to **hideComments** and then click **OK**.
9. Switch to **SRTEdit.jsp** in the editor and select the check box component that you just created. In the Property Palette, click the **Value** field and then click **Bind to Data** .

10. In the Value dialog box, expand **JSF Managed Beans** and **SRTEdit** in the Variables tree. Select **hideComments**, click the **right arrow** to add it to the Expression, and then click **OK**.



11. Follow all steps in "Persisting or Canceling the Update," except that you edit **SRTEdit.java** and **SRTEdit.**jsp (instead of SREdit.java and SREdit.jsp), and you add the following code to update the service request (instead of the `updateSR()` method):

```
public String updateTSR(){
        FacesContext ctx = FacesContext.getCurrentInstance();
        Application app = ctx.getApplication();
        ValueBinding curUser = app.createValueBinding
          ("#{UserInfo.currentUser}");
        Users currentUser = (Users)curUser.getValue(ctx);
        ServiceHistories historynotes = new ServiceHistories();
        historynotes.setServiceRequests(serviceRequest);
        historynotes.setNotes(getInputTextarea1().
          getValue().toString());
        historynotes.setLineNo(null);
        if (hideComments) {
            historynotes.setSvhType("Hidden");
        }
        else {
            historynotes.setSvhType("Technician");
        }
        historynotes.setSvhDate(null);
        historynotes.setUsers(currentUser);
        serviceRequest.addServiceHistories(historynotes);

        ServiceLocator serviceLocator = null;
        try {
            serviceLocator = serviceLocator.getInstance();
            ServiceRequestFacadeLocal srLocal =
              (ServiceRequestFacadeLocal)serviceLocator.
              getFacadeService
              ("java:comp/env/ejb/ServiceRequestFacade");
            srLocal.mergeEntity(serviceRequest);
```

```
            return "techsaved";
        } catch (NamingException e) {
            // TODO
            e.printStackTrace();
        }
        return "techsaved";
    }
```

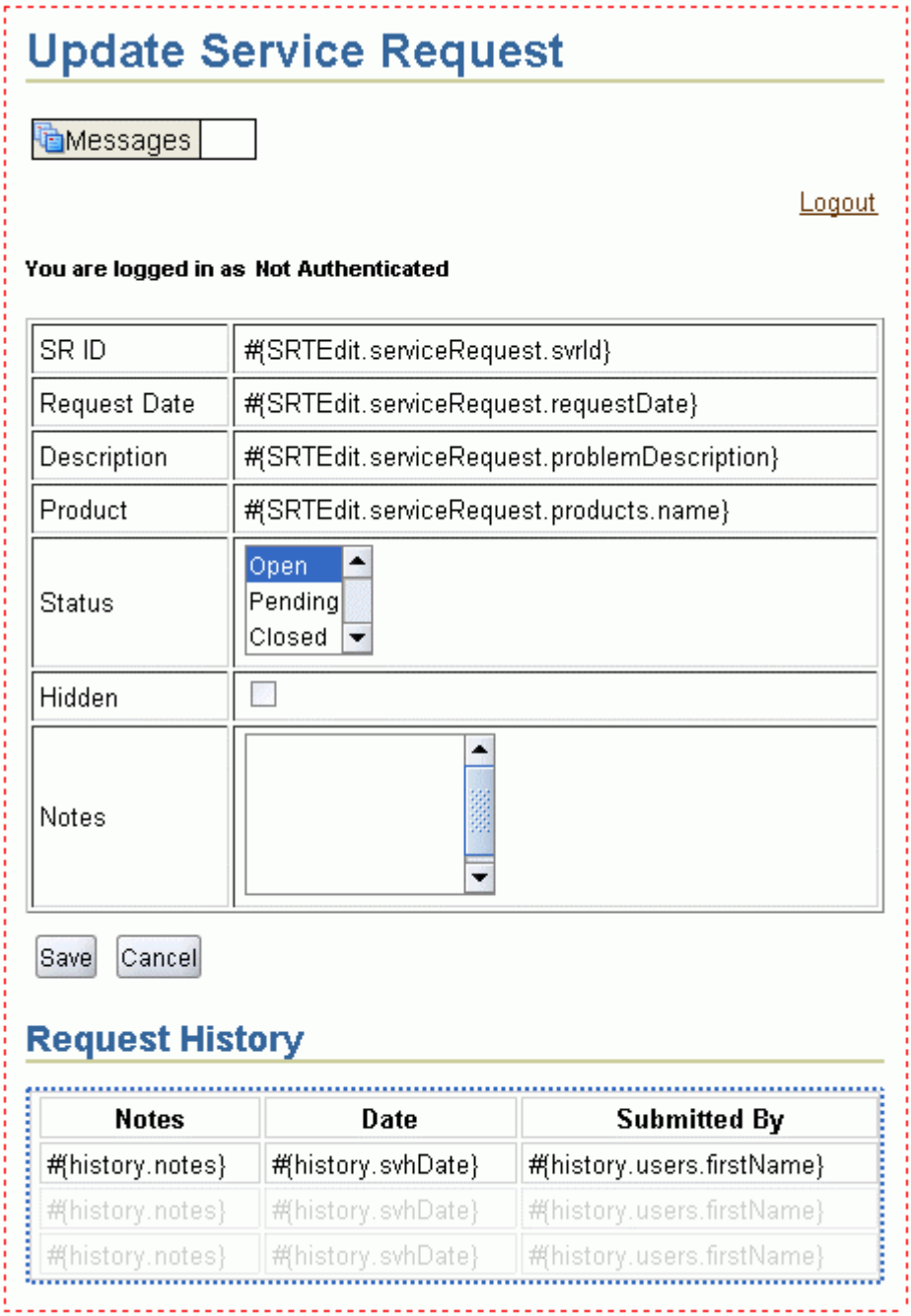
Note that this method returns the string `techsaved`, which is the name of the navigation case (pointing from this page to the SRList page) that you defined earlier in the JSF Navigation Diagram.

Press **[Alt]+[Enter]** when prompted to add several import statements. When you are given a choice of Application packages to import, select **`javax.faces.application.Application`**.

Continue with creating the command buttons as in "Persisting or Canceling the Update," except that you define the Action for the Cancel button as **`techsaved`** (rather than `saved`). You also need to change the scope of the managed bean to `session` and implement the `Serializable` interface for the `SRTEdit` backing bean class, as you did for the `SREdit` backing bean.

12. Follow all steps in "Displaying Detail Records," except that you edit **SRTEdit.java** and **SRTEdit.**jsp (instead of SREdit.java and SREdit.jsp).
13. Do *not* conditionally hide detail records as you did for the customers' edit page. All of the notes should be visible to staff, even those marked as hidden.

14. Click **Save All** to save your work. The page should look like the following screenshot:



Testing the Update Logic and the UI Components

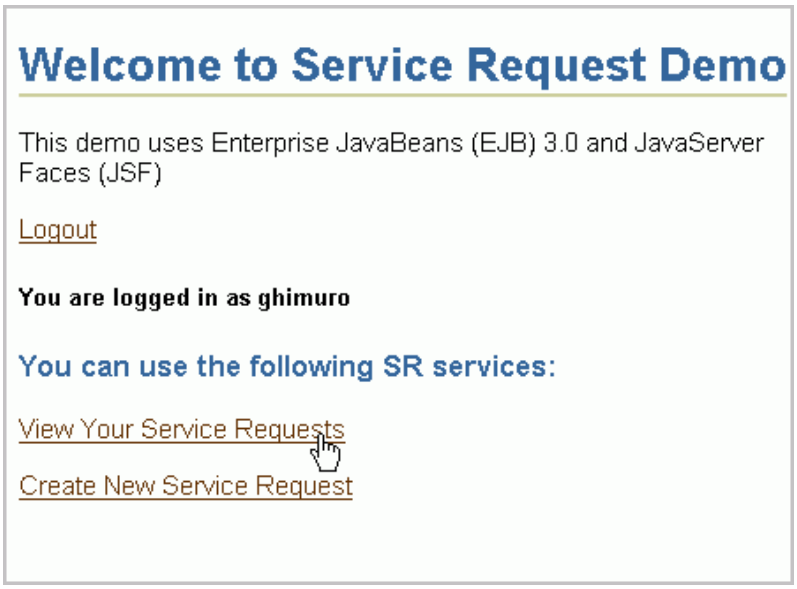
Now that you have defined update logic and created the necessary UI components to edit service requests, you can test the functionality by performing the following steps:

1. In the Applications Navigator, expand the **ViewController** project and the **Web Content** node.
2. Right-click **index.jsp** and select **Run** from the context menu.

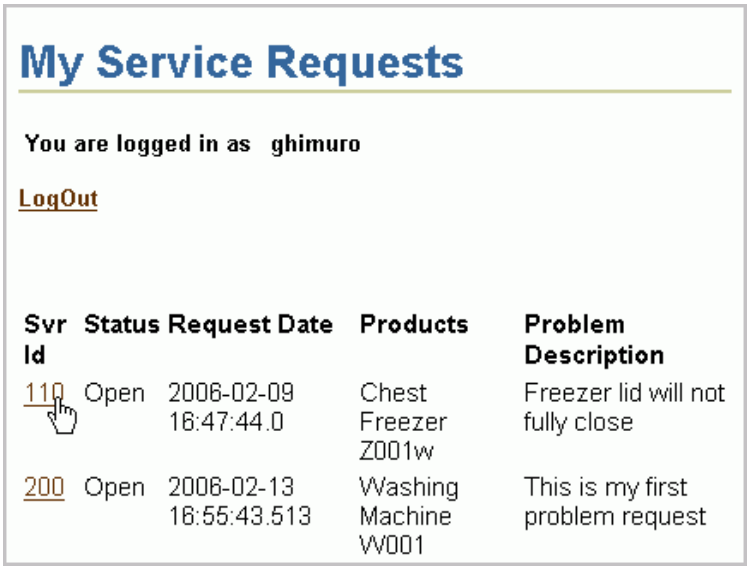
3. A browser opens to display the Logon page. Enter the following logon information and click **Sign On**:

Username	Password
`ghimuro`	`welcome`

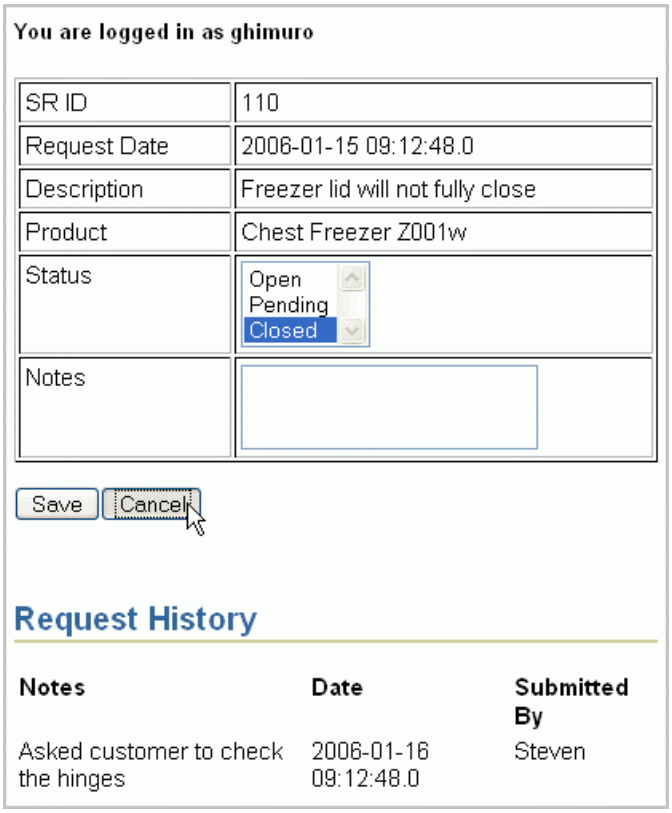
4. On the SRWelcome page, click **View Your Service Requests**.



5. On the List page, click the link for one of the service requests.

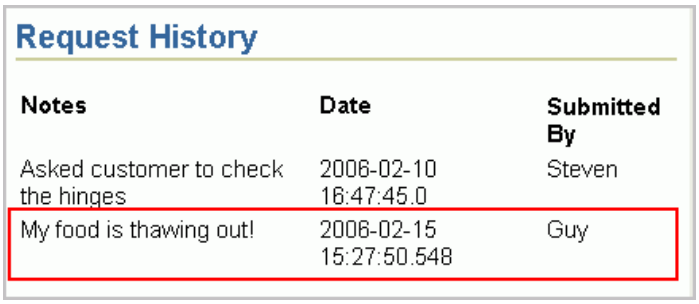


6. On the Edit page, click **Cancel**. You should return to the List page.



7. Click a link for a different service request. On the Edit page, change the status and enter a note, and then click **Save**.

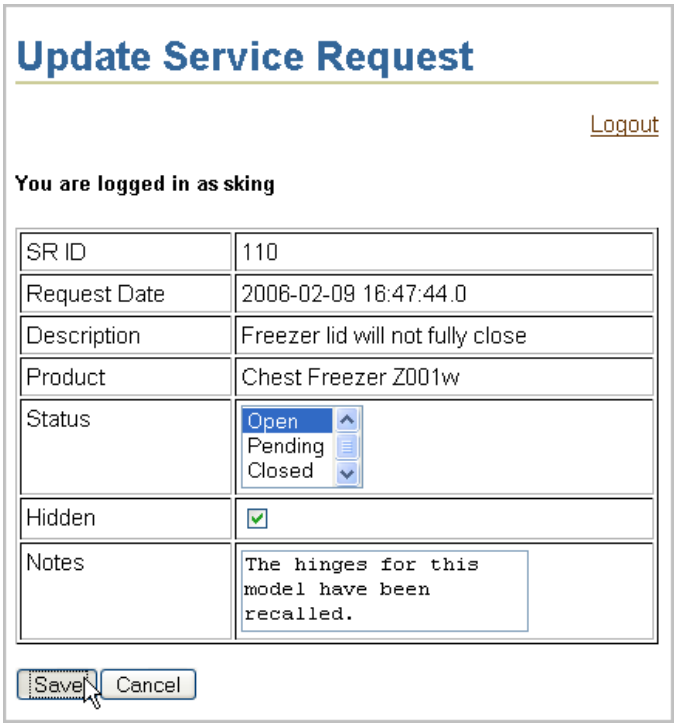
8. You should return to the List page. Click the same link again to see that the detail section on the edit page shows the new note entry.



9. Click **Logout** to redisplay the Logon page. Enter the following logon information, and then click **Sign On**:

Username	Password
`sking`	`welcome`

10. On the SRWelcome page, click **View Your Service Requests**.
11. On the List page, click the link for SR ID **110**. On the Edit page, change the status, mark the note as hidden, add a note, and then click **Save**:



12. You are returned to the list page. Notice the changed status. Click the link to SR ID **110** again. In the details section, you should see the note that you just entered.

Request History

Notes	Date	Submitted By
Asked customer to check the hinges	2006-02-10 16:47:45.0	Steven
My food is thawing out!	2006-02-15 15:27:50.548	Guy
The hinges for this model have been recalled.	2006-02-15 15:44:08.995	Steven

13. Click **Logout**, and then log back in with the following:

Username	Password
`ghimuro`	`welcome`

Note: If you get an error at this point, restart the application and log in as **`ghimuro`**, then continue with the remaining steps.

14. Click **View Your Service Requests**. On the List page, click the SR ID **110** link. The detail section shows none of the hidden notes.

Request History

Notes	Date	Submitted By
Asked customer to check the hinges	2006-02-10 16:47:45.0	Steven
My food is thawing out!	2006-02-15 15:27:50.548	Guy

Summary

In this chapter, you created the page that enables users to edit service requests. To accomplish this, you performed the following key tasks:

- Created the edit pages for customers and staff that contain:
  - Display components for read-only data
  - Components to facilitate data entry
  - Buttons to persist or cancel the update
  - Display of detail service history records
- Added logic to the customer edit page to conditionally hide certain detail records
- Added logic to the staff edit page to mark certain detail records as hidden from customer view
- Tested the update logic and the UI components

10

Creating the Triage Page

This chapter describes how to create the SRTriage page that enables managers to triage (assign) service requests to technicians.

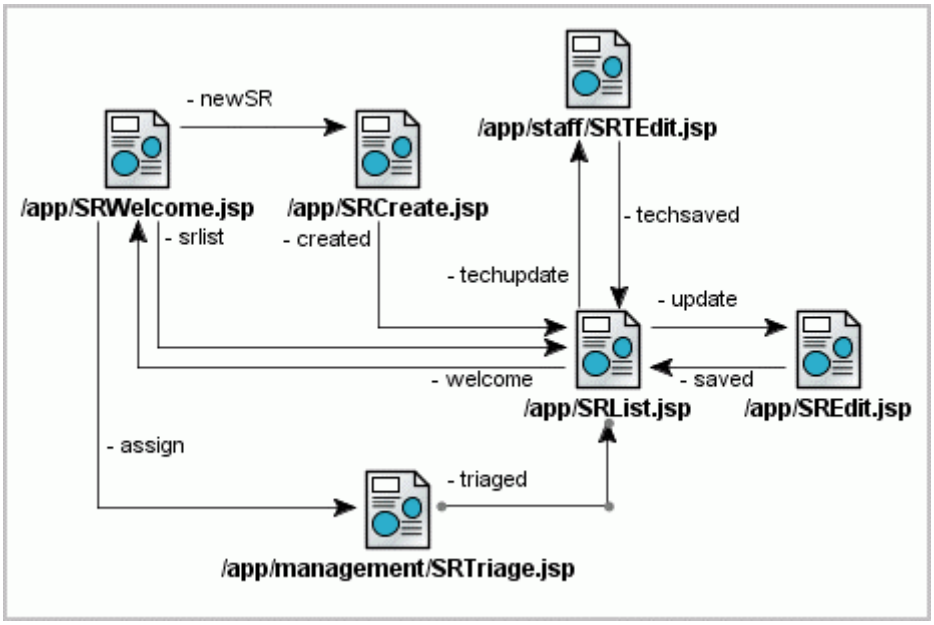
If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh09.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

- Triage Page Overview
- Creating the Page Outline
- Adding Named Queries to the Data Model
- Exposing the New Named Queries in the Session Facade
- Updating the Backing Bean to Retrieve the Data
- Adding the Data Table to the Page
- Persisting or Canceling the Update
- Testing the Functionality of the Page
- Summary

Triage Page Overview

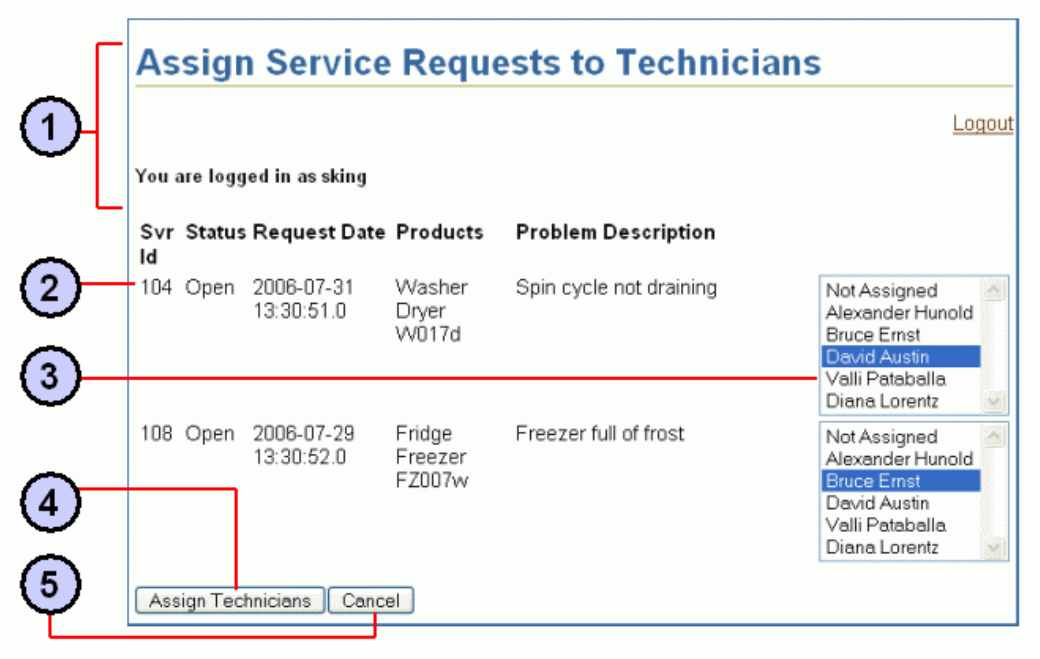
Managers use the SRTriage page to assign new requests to technicians and also to reassign existing requests to different technicians. The JSF Navigation Diagram shows how the SRTriage page relates to the other pages:



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

- This page is available only to managers.
- The manager accesses the SRTriage page from the welcome page by clicking a link that is visible to only those users with the role of `manager`.
- When the manager clicks the Triage Service Requests link, the SRTriage page shows the current list of unassigned and open service requests.
- The manager can select a technician from the list of technicians for each request, and then click Assign Technicians to update the service requests.

The following screenshot depicts the features of the SRTriage page:

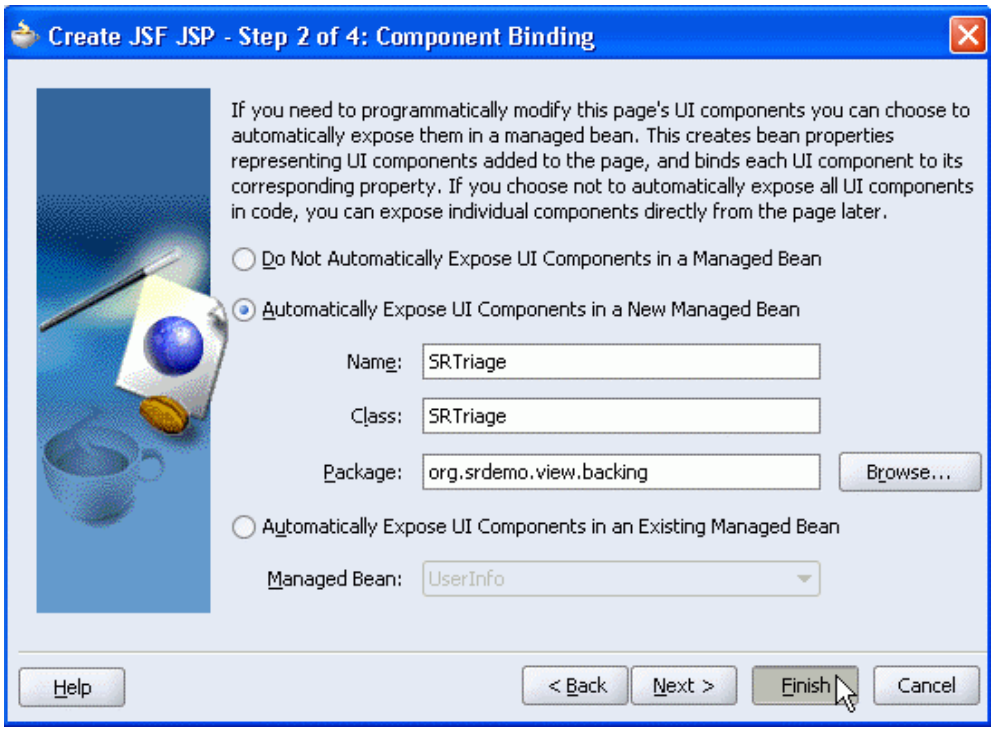


1. Each application page contains a title, information about the logged in user, and a Logout link.
2. A JSF Data Table component displays a list of open and unassigned service requests. The data comes from a named query that you add to the `ServiceRequests` entity. You add code to the backing bean for the page to retrieve this list of service requests, and you use EL to bind the data table to this code.
3. A JSF ListBox component displays all users with the `technician` role. The data comes from a named query that you add to the `Users` entity. You add code to the backing bean for the page to retrieve this list of technicians, and you use EL to bind the list box to this code. If the service request has already been assigned to a technician, you also use EL to set the value of the list box to the technican who is already assigned to that service request.
4. An Assign Technicians button calls code that you add to the backing bean to persist the data to the underlying database tables. The bean is set to session scope so that the data is available for all pages in the session. This code in the bean returns a string value that corresponds to one of the From outcomes in a navigation rule that you defined earlier.
5. A Cancel button has an action set to one of the From outcomes in a navigation rule that you defined earlier. This enables navigation to occur without executing the code to persist the data.

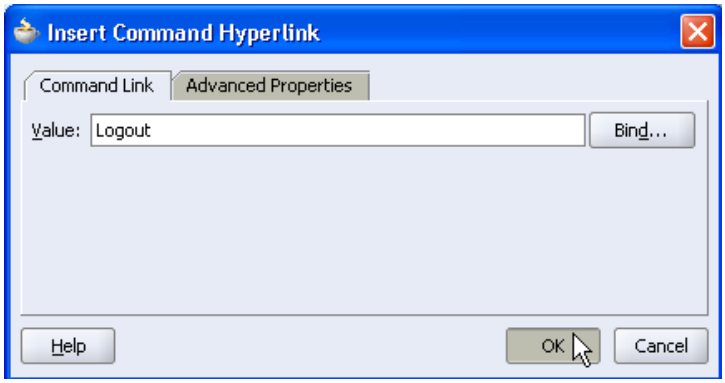
Creating the Page Outline

You first create the basic page by performing the following steps:

1. Open or switch to **`faces-config.xml`** in the editor. (To open, right-click the **ViewController** project and select **Open JSF Navigation** from the context menu.)
2. On the **Diagram** tab, double-click the **SRTriage** page to invoke the Create JSF JSP wizard.
3. If the Welcome page of the wizard appears, click **Next**.
4. On the JSP File page of the wizard, click **Next**.
5. On the Component Binding page of the wizard, select the option **Automatically Expose UI Components in a New Managed Bean**. Change the Name to **`SRTriage`** and the package to **`org.srdemo.view.backing`**, and then click **Finish**.

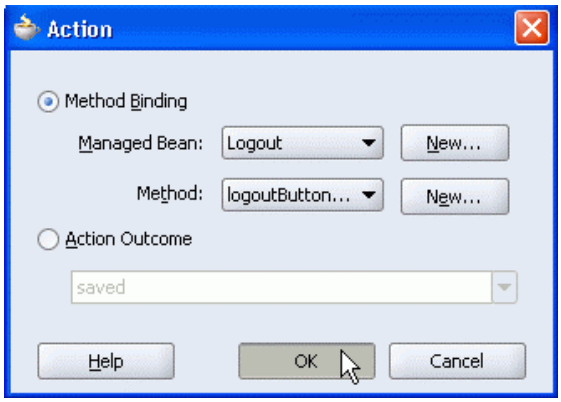


6. The SRTriage page is created and opens in the editor. Enter the text **`Assign Service Requests to Technicians`** and then select **Heading 1** from the Block Format drop-down list.
7. From the Component Palette, select **JSF HTML** from the drop-down list and drag the **Messages** component to the next line in the visual editor.
8. Place the cursor after the Messages component and press **[Enter]**, and then click the **Command Hyperlink** component in the Component Palette.
9. In the Insert Command Hyperlink dialog box, enter the Value **`Logout`** and click **OK**.



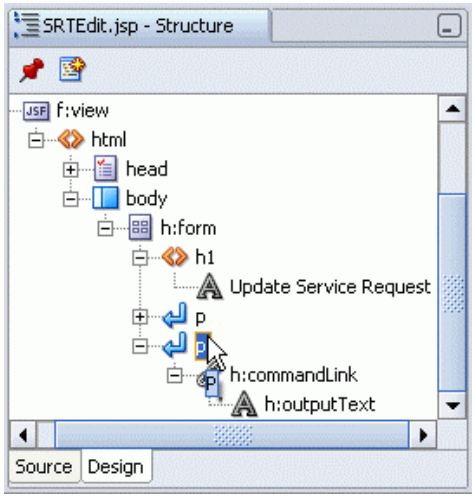
10. In the Property Inspector for the command hyperlink, enter an Action of **`#{}`** and press **[Enter]**, and then click back into the Action property. Click the **ellipsis** ... at the end of the field.
11. In the Action dialog box, ensure that the **Method Binding** option is selected.

Select **Logout** from the Managed bean drop-down list, and select **logoutButton_action** from the Method drop-down list. Click **OK**.

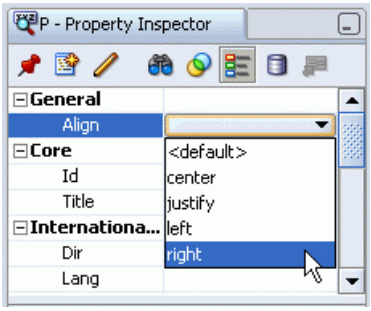


This ensures that the Logout link executes the code in the Logout managed bean that you created in a previous chapter.

12. In the Structure window, select the **paragraph node** that contains the commandLink that you just defined.



13. In the Property Inspector, set the **Align** property of the paragraph to **`right`** by selecting that value from the drop-down list.



14. In the editor, on the next line enter the text **`You are logged in as`**, and press **[Space]**. From the Block Format drop-down list, select **Heading 5**.
15. In the Component Palette click the **Output Text** component.
16. An output text field is created in the editor. In the Property Inspector for the output text, select the **Value** property and click **Bind to Data** .
17. In the Value dialog box, expand **JSF Managed Beans** and **UserInfo**. Select **userName** and click the right arrow to add it to the expression on the right. Then click **OK**. This causes the application to display the name of the authenticated user at run time.
18. In the Component Palette, select **CSS** from the drop-down list and then select **JDeveloper**.
19. Click **Save All** to save your work.

Adding Named Queries to the Data Model

The next addition to the page is a data table to display a subset of service requests and a list of technicians to which each request can be assigned. To accomplish this, you first need to add two more named queries to the data model. The first named query that you add retrieves the service requests that are open or unassigned. The second returns a list of technicians.

To add the named queries that are needed for the data table, perform the following steps:

1. In the Applications Navigator, expand the **Model** project node, and then expand **Application Sources** and **oracle.srdemo.persistence**.
2. Double-click **`ServiceRequests.java`** to open it in the editor.
3. Add the following named query to the Named Queries section, being sure to use a comma to separate it from the other named queries:

   ```
   @NamedQuery(name="findTriageRequests",query="select object(sr)" +
   " from ServiceRequests sr where " +
   "sr.status = :status  or sr.users1 is null " )
   ```

   This query retrieves the service requests that have a specified status or are not yet assigned to a technician. You pass a status of `Open` when you call this named query in the data table, so you use it to retrieve all open service requests as well as any service request that is not assigned.
4. In the Applications Navigator, double-click **`Users.java`** to open it in the editor.
5. Add the following named query to the Named Queries section, being sure to use a comma to separate it from the other named queries:

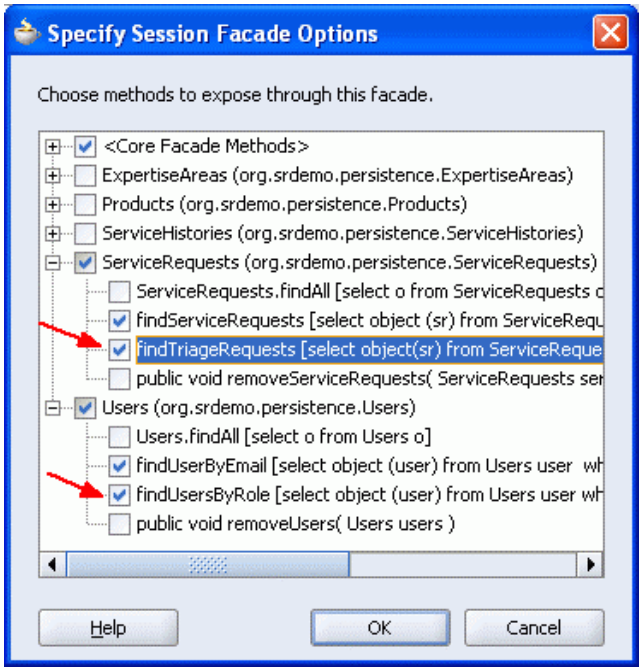
   ```
   @NamedQuery(name="findUsersByRole", query="select object (user)"+
   " from Users user where user.userRole = :userRole")
   ```

   This query retrieves all the users of a specified role. You pass a role name of `technician` when you call this named query in the data table.
6. Click **Save All** to save your work.

Exposing the New Named Queries in the Session Facade

The new named queries cannot be used in the application until you modify the session facade to include methods to access them. To expose the new named queries, perform the following steps:

1. The classes containing the named queries must be compiled in order for them to be visible to the facade editor, so click **Rebuild** to compile all.
2. In the Applications Navigator, expand **Model** > **Application Sources** > **org.srdemo.business**.
3. Right-click **ServiceRequestFacadeBean.java** and select **Edit Session Facade**.
4. In the Session Facade Options dialog box, expand the **ServiceRequests** and **Users** nodes and select the check boxes next to the queries that you just added: **findTriageRequests** and **findUsersByRole**. Click **OK**.



5. The ServiceRequestFacadeBean.java file opens in the editor. Scroll to the end of the file to see the new methods that were added to expose the new named queries.

```
/** <code>select object(sr) from ServiceRequests sr where sr.status
public List<ServiceRequests> findTriageRequests(Object status) {
    return em.createNamedQuery("findTriageRequests").setParameter("
}

/** <code>select object (user) from Users user where user.userRole
public List<Users> findUsersByRole Object userRole) {
    return em.createNamedQuery("findUsersByRole").setParameter("use
}
```

6. Click **Save All** to save your work.

Update the Backing Bean to Retrieve the Data

The SRTriage page displays open and unassigned service requests with a list of technicians who can be assigned to each request. The data is retrieved from the session facade and exposed via a data table in the SRTriage.jsp page. To modify the backing bean for the page to make this data available, perform the

following steps:

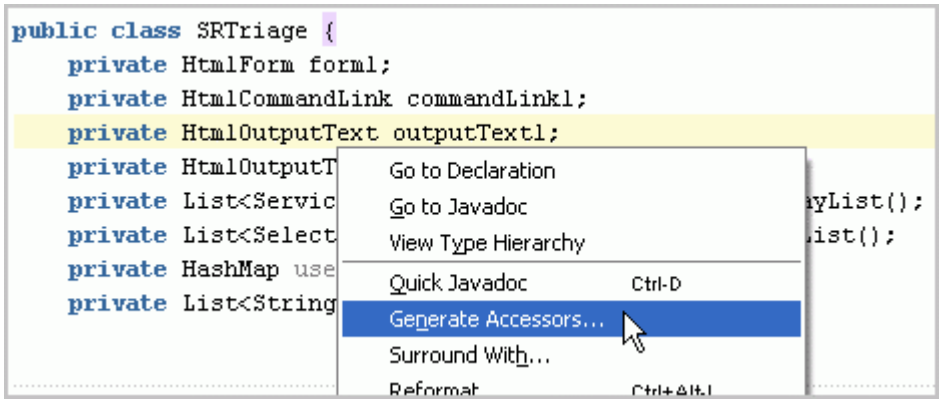
1. In the Applications Navigator, expand the **ViewController** project and the **Application Sources**, **org.srdemo.view**, and **backing** nodes.
2. Double-click **SRTriage.java** to open it in the editor.
3. Add the following private variables:

```
private List<ServiceRequests> srTriageList = new ArrayList();
private List<SelectItem> techniciansList = new ArrayList();
private HashMap usersMap = new HashMap();
private List<String> assignedUsers;
```

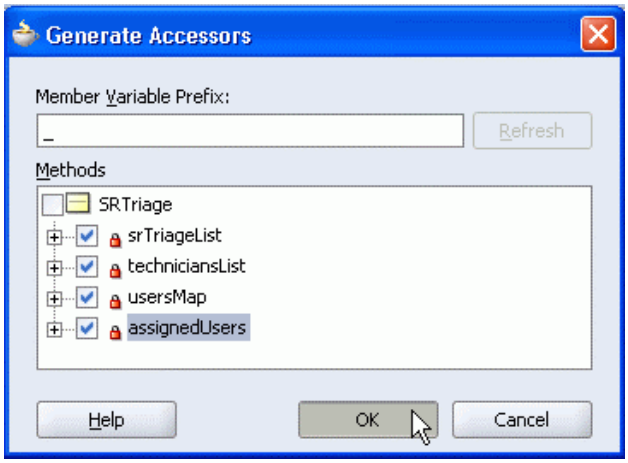
```
public class SRTriage {
    private HtmlForm form1;
    private HtmlCommandLink commandLink1;
    private HtmlOutputText outputText1;
    private HtmlOutputText outputText2;
    private List<ServiceRequests> srTriageList = new ArrayList();
    private List<SelectItem> techniciansList = new ArrayList();
    private HashMap usersMap = new HashMap();
    private List<String> assignedUsers;
```

The data for these variables is retrieved from the session façade, and their accessors can be used by components in the page.

4. Press **[Alt]+[Enter]** when prompted to add import statements, being sure that the `List` class that you import is **`java.util.List`**.
5. Right-click within the code editor window and select **Generate Accessors** from the context menu.



6. In the Generate Accessors dialog box, select the check boxes next to all of the variables that you just added, and then click **OK**.



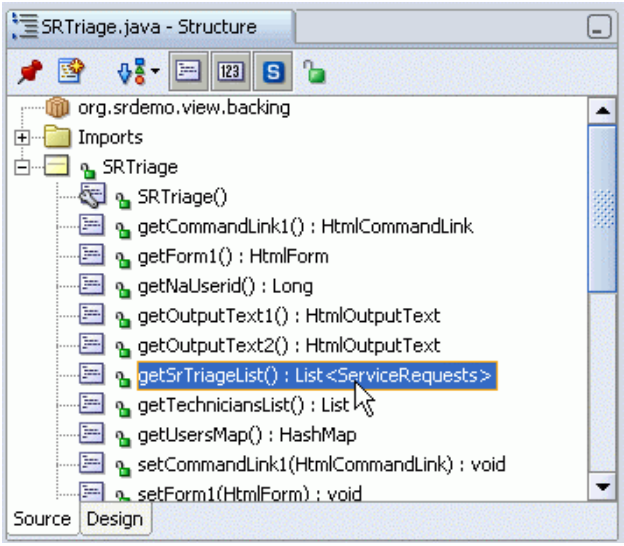
7. You can scroll to the bottom of the file to see the accessors that were created.
8. In the `SRTriage.java` backing bean class, add a constructor that populates an array with all the technicians. You use an `ArrayList`, which has `javax.jsf.SelectItem` objects so that it can be displayed in a list box.

   The constructor gets an instance of the `ServiceLocator` class that caches the session facade instances and calls the `findUsersByRole` method in the session facade. This method returns a `List` of `Users`. The retrieved list is then populated into `techniciansList`, which has `SelectItem` objects to store `userid` and `username`. Each of the retrieved `Users` is also stored in a `hashMap` in order to match it with the user selected in the list box of the page.

   Add the following method to SRTriage.java:

```
    public SRTriage(){
        // Get the Service Locator's instance
            ServiceLocator serviceLocator = null;
        try {
            serviceLocator = serviceLocator.getInstance();
            ServiceRequestFacadeLocal srLocal =
(ServiceRequestFacadeLocal)serviceLocator.getFacadeService("java:
comp/env/ejb/ServiceRequestFacade");
            List usersList =
               srLocal.findUsersByRole("technician");
            techniciansList.add(new SelectItem("", "Not
Assigned"));
            Iterator it  = usersList.iterator();
                 while(it.hasNext()){
                     Users user  = (Users)it.next();
                     String userID = user.getUserId().toString();
                     usersMap.put(userID, user);
                     String userName = user.getFirstName() +
                        " " + user.getLastName();
                     techniciansList.add(new SelectItem(userID,
userName));
                     }

        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
```

9. Press **[Alt]+[Enter]** to accept each of the suggested import statements.
10. In the Structure window, double-click the `getSrTriageList()` method to locate it in the editor.



11. Update the `getSrTriageList()` method to retrieve the service requests that need to be triaged. The method gets an instance of the `ServiceLocator` class, which caches the session facade instances and calls the `findTriageRequests()` method in the session facade. This method returns a list of service requests that need to be triaged (those that have a status of "Open" or are not assigned to any technician).

```
    public List<ServiceRequests> getSrTriageList() {
       // Get the Service Locator's instance
       ServiceLocator serviceLocator = null;
       try {
         serviceLocator = serviceLocator.getInstance();
         ServiceRequestFacadeLocal srLocal =
(ServiceRequestFacadeLocal)serviceLocator.getFacadeService("java:
comp/env/ejb/ServiceRequestFacade");
         List<ServiceRequests> srList =
srLocal.findTriageRequests("Open");
         if (assignedUsers == null) {
                assignedUsers = new ArrayList();
            }
            else {
                assignedUsers.clear();
            }
            for (ServiceRequests sr : srList) {
                String userId = (sr.getUsers1() == null) ? "" :
sr.getUsers1().getUserId().toString();
                assignedUsers.add(userId);
            }
            return srList;
       } catch (NamingException e) {
            // TODO
            e.printStackTrace();
       }
```

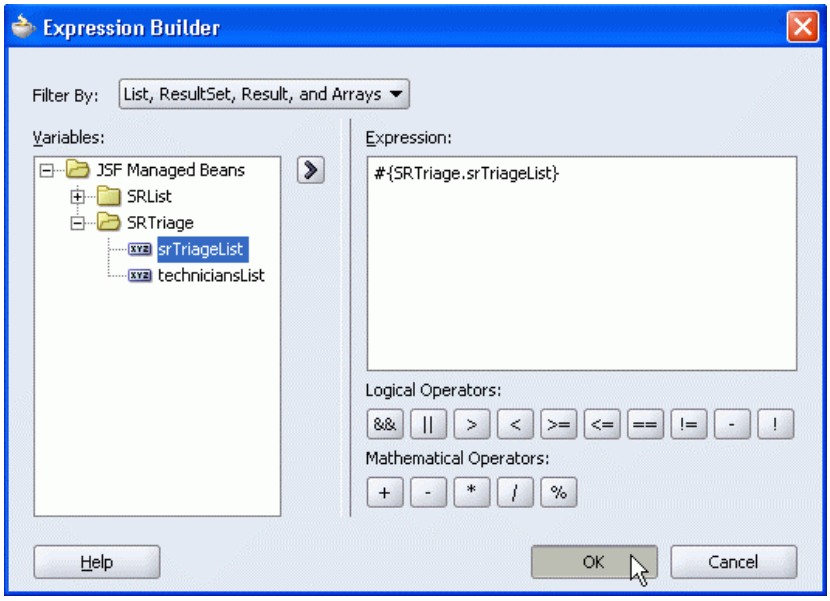
```
        return srTriageList;
    }
```

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

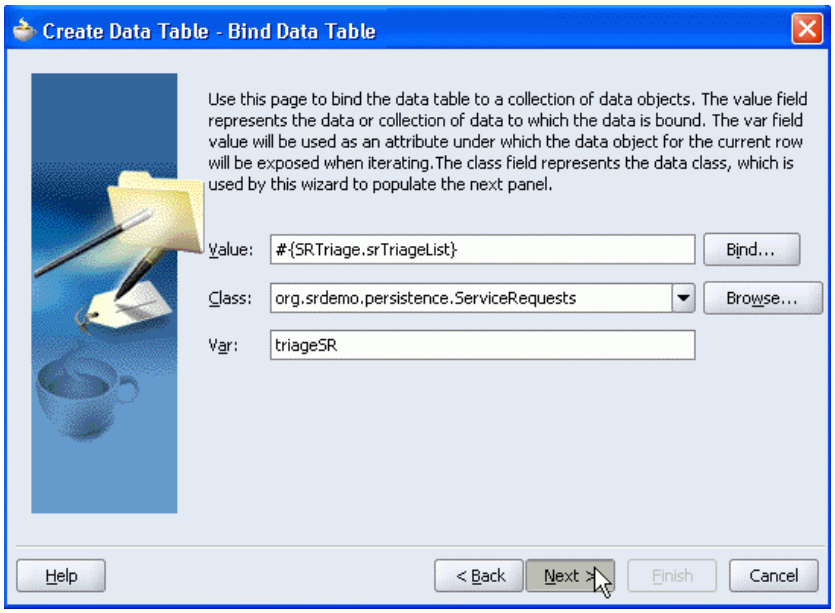
Adding the Data Table to the Page

Now that you have modified the backing bean, you can use its code in the data table. To create the data table on the page, perform the following steps:

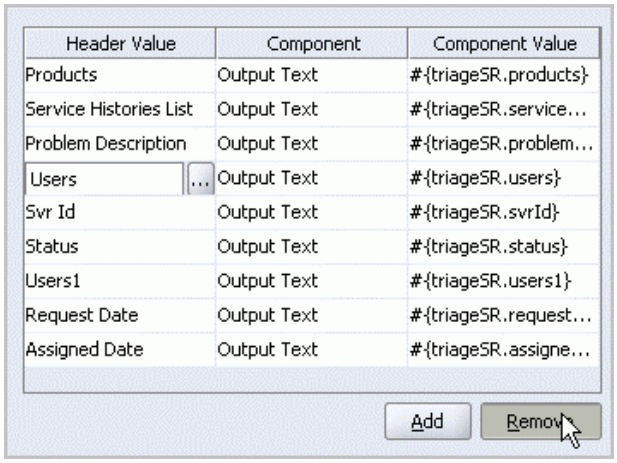
1. With SRTriage.jsp open in the editor, in the Component Palette select **JSF HTML** from the drop-down list and drag the **Data Table** component to a blank line at the end of the page in the visual editor.
2. If the Welcome page of the Create Data Table wizard appears, click **Next**.
3. On the Binding page of the wizard, select the option **Bind the Data Table Now** and click **Next**.
4. On the Bind Data Table page of the wizard, click **Bind**.
5. In the Expression Builder dialog box, expand **SRTriage**, select **srTriageList**, and click the **right arrow** to add it to the expression. Then click **OK**.



6. In the wizard, change the Class to `org.srdemo.persistence.ServiceRequests`. Enter `triageSR` in the Var field, and then click **Next**.



7. On the Header and Row Data page of the wizard, select **Users** and click **Remove**.



Header Value	Component	Component Value
Products	Output Text	#{triageSR.products}
Service Histories List	Output Text	#{triageSR.service...
Problem Description	Output Text	#{triageSR.problem...
Users	Output Text	#{triageSR.users}
Svr Id	Output Text	#{triageSR.svrId}
Status	Output Text	#{triageSR.status}
Users1	Output Text	#{triageSR.users1}
Request Date	Output Text	#{triageSR.request...
Assigned Date	Output Text	#{triageSR.assigne...

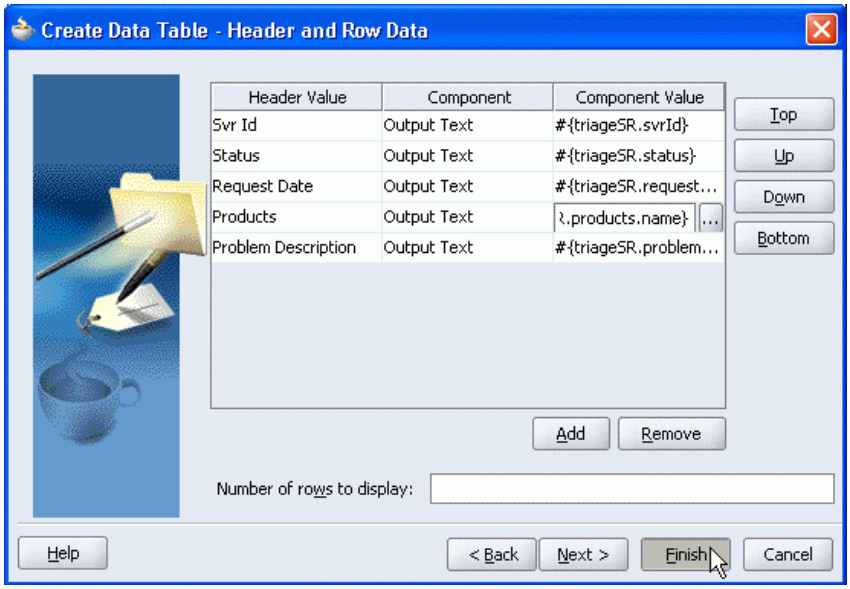
Add
Remov

In a similar fashion, remove **AssignedDate**, **Service Histories List**, and **Users1**.

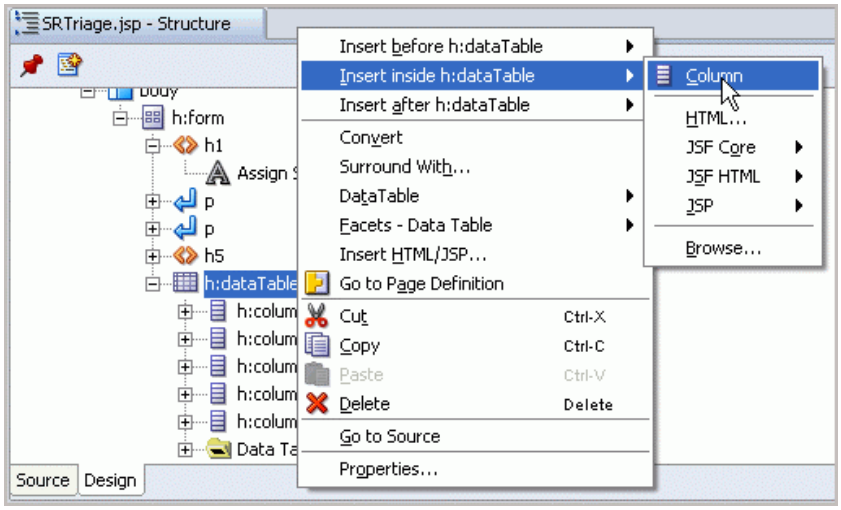
8. By using the buttons at the right of the window, rearrange the data in the following order:

   `Svr ID`
   `Status`
   `Request Date`
   `Products`
   `Problem Description`

9. Change the Component Value of `Products` to **`#{triageSR.products.name}`** and click **Finish**.



10. Now that you have created the data table, you need to add another column to show the list of technicians that the manager can choose. Select **SRTriage.jsp** in the Applications Navigator or in the editor. In the Structure window, right-click the **h:dataTable** component and select **Insert inside h:dataTable** and **Column** from the context menu.



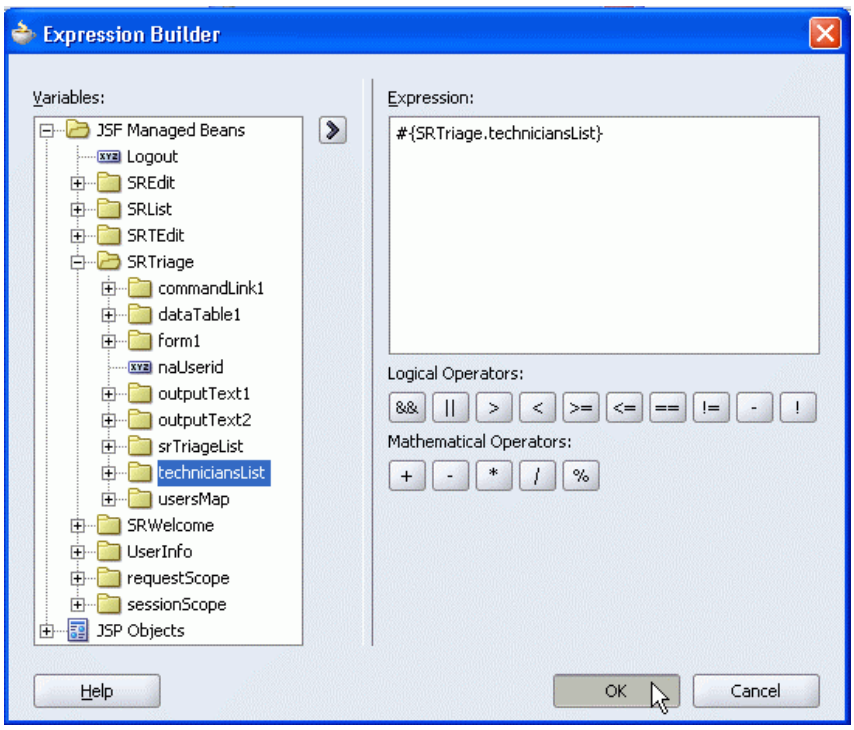
This inserts a column at the right side of the data table.

11. You need to add a list box to this column. The select items for this list box use the `techniciansList` that you created in the SRTriage backing bean. You then bind the value of the list box to the `userId` of the `users1` attribute of the service request. This is the attribute that is mapped to `ASSIGNED_TO` in the database.

    In the Component Palette, select **JSF HTML**. Select the **Listbox** component and drag it to the column that you just inserted in the previous step. You can drag it to the column either in the editor or in the Structure window.

12. In the Insert Listbox dialog box, click **Bind**.

13. In the Expression dialog box, expand the **SRTriage** node, select **techniciansList**, and click the **right arrow** to add the expression `#{SRTriage.techniciansList}`. Click **OK**.



14. In the Insert Listbox dialog box, click the **Advanced Properties** tab.
15. Select the **Value** property and enter the following value:

    `#{SRTriage.assignedUsers[SRTriage.dataTable1.rowIndex]}`

    Click **OK** to dismiss the Insert List box dialog box.
16. Click **Save All** to save your work.

Persisting or Canceling the Update

Now you add code to the managed bean so that the page can process the service request updates. You also add a button to call that code and another button to cancel the record update. Then you change the scope of the managed bean to `session` so that updated information can be displayed throughout the session.

Perform the following steps:

1. Open or switch to **SRTriage.java** (**ViewController** > **Application Sources** > **org.srdemo.view** > **backing**).
2. Add the following method:

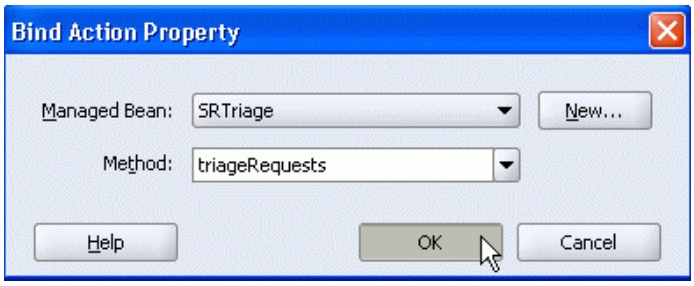
```
    public String triageRequests(){
       for (int i = 0; i < getDataTable1().getRowCount(); i++) {
         getDataTable1().setRowIndex(i);
         ServiceRequests assignTechToSR =
(ServiceRequests)getDataTable1().getRowData();
          String assignedUserId = assignedUsers.get(i);
```

```
            if (
usersMap.containsKey(assignedUserId))
              {
                Users techUser =
(Users)usersMap.get(assignedUserId);
                assignTechToSR.setUsers1(techUser);
                assignTechToSR.setAssignedDate(null);
                ServiceLocator serviceLocator = null;
                try {
                  serviceLocator =
serviceLocator.getInstance();
                  ServiceRequestFacadeLocal srLocal =
(ServiceRequestFacadeLocal)serviceLocator.getFacadeService("java:
comp/env/ejb/ServiceRequestFacade");
                  srLocal.mergeEntity(assignTechToSR);

                } catch (NamingException e) {
                    // TODO
                    e.printStackTrace();
                }
            }
        }
        return "triaged";
    }
```

This method processes the service request updates after the manager has selected technicians for each service request that should be assigned. The method iterates over each row of the data table to see if the manager has selected a user from the list. If so, it gets an instance of `ServiceLocator`, finds the service request facade bean, and calls the `mergeEntity` method, passing the updated `ServiceRequests` object to be committed to the database.

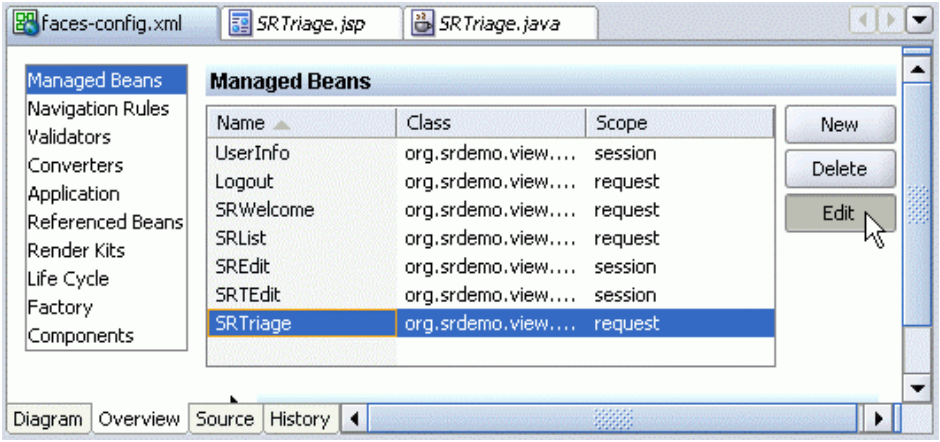
Note: This method returns the string `triaged`, which is the name of the navigation case (pointing from this page to the SRList page) that you defined earlier in the JSF Navigation Diagram.

3. Switch to **SRTriage.jsp** in the editor. Place the cursor after the data table within the form (red dotted line) and press **[Enter]** to create a blank line.
4. In the Component Palette, select **JSF HTML** from the drop-down list and click **Command Button**.
5. In the Property Palette, set the Value property to **`Assign Technicians`**.
6. Double-click the **Assign Technicians** button in the editor, and in the Bind Action Property dialog box, select **triageRequests** from the Method drop-down list. Click **OK**.

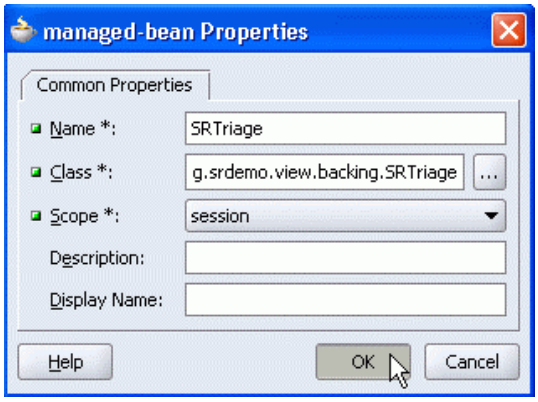


Note: The `triageRequests()` method in the SRTriage managed bean persists the updated records and returns the string `triaged`, which is the name of the navigation case that you defined in the `faces-config.xml` file to navigate to the SRList page.

7. In the editor for **SRTriage.jsp**, place the cursor to the right of the Assign Technicians button and click **Command Button** in the Component Palette.
8. In the Property Palette, set the Value property to `Cancel` and set the Action property to **triaged** so that the user navigates to the SRList page without executing the update code in the backing bean.
9. Change the scope of the managed bean to `session` so that changes can be seen in other pages of the application. Open (right-click **ViewController** and select **Open JSF Navigation**) or switch to the **faces-config.xml** file in the editor.
10. Click the **Overview** tab, select **SRTriage**, and click **Edit**.



11. In the managed-bean Properties dialog box, select **session** from the Scope drop-down list and click **OK**.



12. Classes for beans with `session` scope must implement the `Serializable` interface if they are part of a distributable Web application. Because you intend to deploy this application to a cluster (a group of OC4J instances), you must perform this step.

    Open or switch to **SRTriage.java** and locate the following line:

    ```
    public class SRTriage {
    ```

    Change it to:

    ```
    public class SRTriage implements Serializable{
    ```

    Press **[Alt]+[Enter]** when prompted to import **`java.io.Serializable`**.

13. Click **Save All** to save your work.

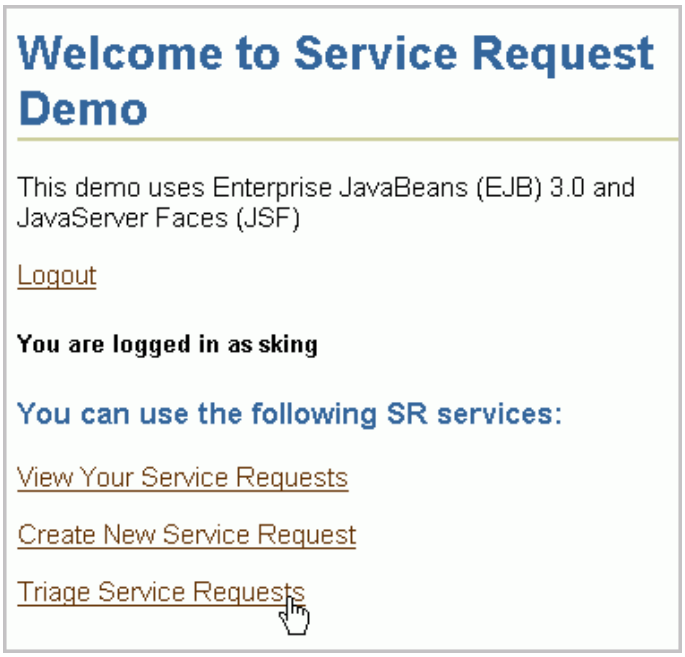
Testing the Functionality of the Page

Now that you have defined update logic and created the necessary UI components to triage service requests, you can test the functionality by performing the following steps:

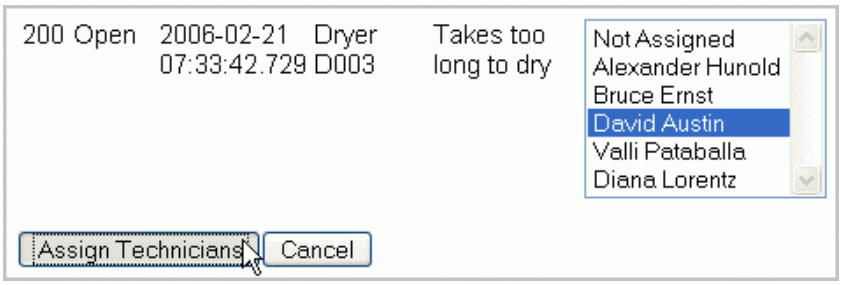
1. In the Applications Navigator, expand the **ViewController** project and the **Web Content** node.
2. Right-click **index.jsp** and select **Run** from the context menu. A browser opens to display the Logon page. Enter the following logon information and click **Sign On**:

Username	Password
`sking`	`welcome`

3. On the SRWelcome page, click **Triage Service Requests**.



4. On the Triage page, click **Cancel**. You should then navigate to the List page, where you can click **Back to Home** to return to the Welcome page.
5. On the Welcome page, again click **Triage Service Requests**.
6. On the Triage page, select a technician for one or more of the unassigned service requests and click **Assign Technicians**.



7. In JDeveloper, click the Connections tab. In the Connections Navigator, expand the **Database**, **SRDemo**, **SRDEMO**, and **Tables** nodes. Double-click the **SERVICE_REQUESTS** table to open it in the editor.
8. Click the **Data** tab. You should see that the update was committed to the database.

   Note: If the table is open in the editor, you need to select **View** > **Refresh** to see the update.

Summary

In this chapter, you created a service requests triaging page that uses JavaServer Faces components. Those components display data coming from the EJB 3.0 data model. You also added an action that processes the service requests displayed in the data table and assigns a technician based on the manager selection.

In short, here are the key tasks that you performed in this chapter:

- Created the page outline with the heading, a Messages component, a Logout link, and a message displaying the name of the logged-in user
- Added named queries to display the service requests and a list of technicians
- Exposed the new named queries by editing the session facade
- Updated the backing bean for the page to get the data from the session facade
- Added a data table component to the page to display a list of open or unassigned service requests, along with a list of technicians to whom they can be assigned
- Added command button components to persist or cancel the update
- Tested the functionality of the page

11

Developing a Page to Insert Records

All of the pages to perform DML that you have created so far have enabled users to update existing data. Now you create an insert page that enables users to insert new service request records into the database. The page contains UI components that ensure data accuracy and facilitate data entry.

If you did not successfully complete the previous section of the tutorial, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh10.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

- Creating the Insert Page
- Adding Data Entry Components
- Persisting or Canceling the Insert
- Testing the Insert Logic and the UI Components
- Summary

Creating the Insert Page

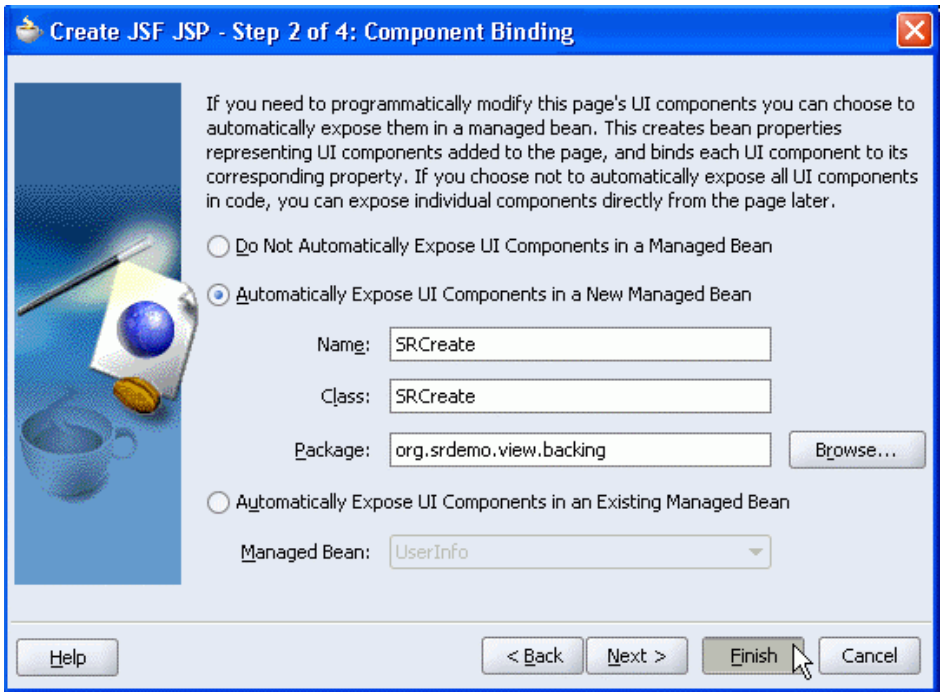
The SRCreate page, which is accessed from the SRWelcome page, enables users to create a new service request. SRCreate enables users to select from a list of all products and then enter a problem description for the service request record and some notes for the service history record.

An 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.

If the user chooses to commit the data, the backing bean code inserts a service request record and a related service history record into the database. Navigation to the SRList page then occurs.

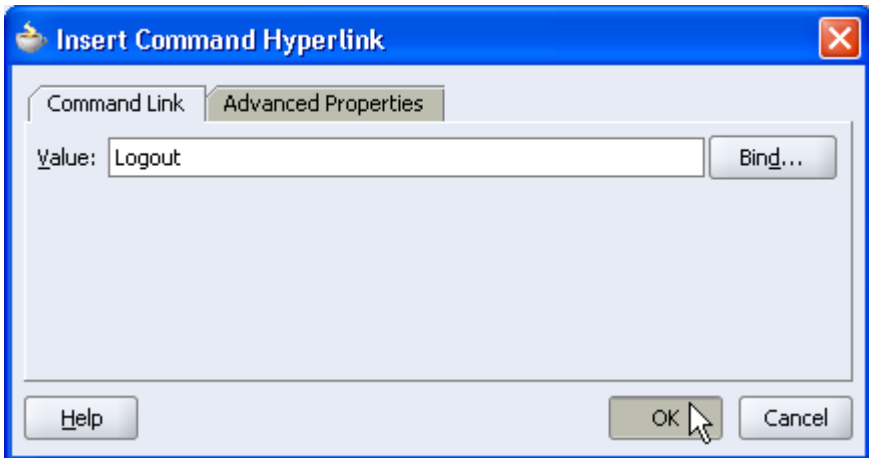
To define the SRCreate page, perform the following steps:

1. Open **faces-config.xml** in the editor if it is not already open, or click its tab if it is open. (To open the file, right-click the **ViewController** project and select **Open JSF Navigation** from the context menu.) Click the **Diagram** tab.
2. Double-click the **/app/SRCreate.jsp** icon to invoke the Create JSF JSP wizard.
3. If the Welcome page of the wizard appears, click **Next**.
4. On the JSP File page of the wizard, click **Next**.
5. On the Component Binding page of the wizard, select the **Automatically Expose UI Components in a New Managed Bean** option. Change the Name of the managed bean to `SRCreate` and the Package to `org.srdemo.view.backing`. Click **Finish**.

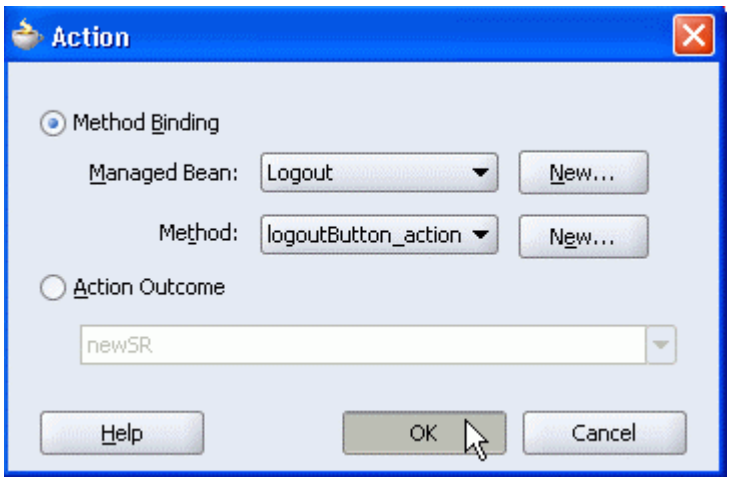


6. The new page opens in the editor. Enter the text `Create a new service request` and then select **Heading 1** from the Block Format drop-down list.
7. In the Component Palette, select **JSF HTML** from the drop-down list and drag the **Messages** component to the next line on the page in the visual editor.

8. Place the cursor after the Messages component and press **[Enter]**. Then click the **Command Hyperlink** component.
9. In the Insert Command Hyperlink dialog box, enter the Value `Logout` and click **OK**.

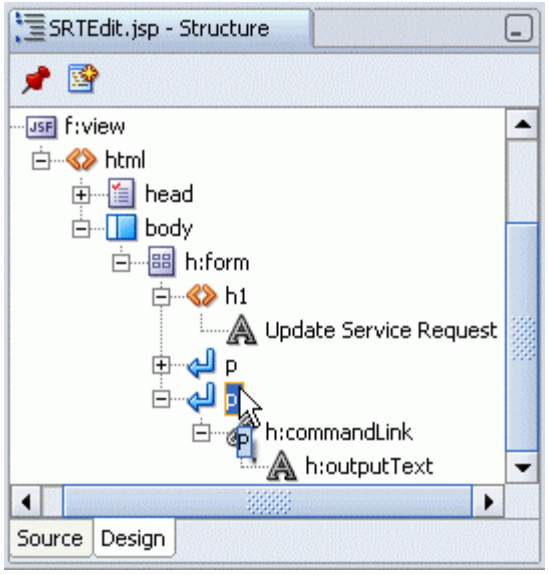


10. In the Property Inspector for the command hyperlink, enter the Action `#{}` and press **[Enter]**, and then click back into the Action property. Click the ellipsis ... at the end of the field.
11. In the Action dialog box, ensure that the **Method Binding** option is selected. Select **Logout** from the Managed bean drop-down list, and select **logoutButton_action** from the Method drop-down list. Click **OK**.

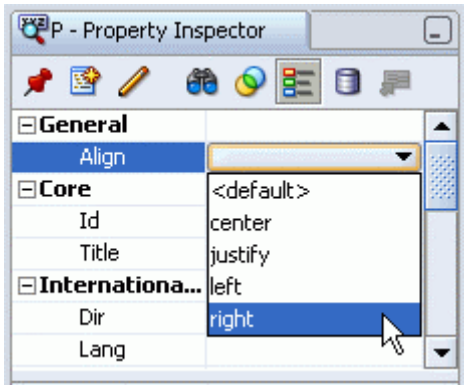


This ensures that the Logout link executes the code in the Logout managed bean that you created in a previous chapter.

12. In the Structure window, select the **paragraph node** p that contains the commandLink that you just defined.



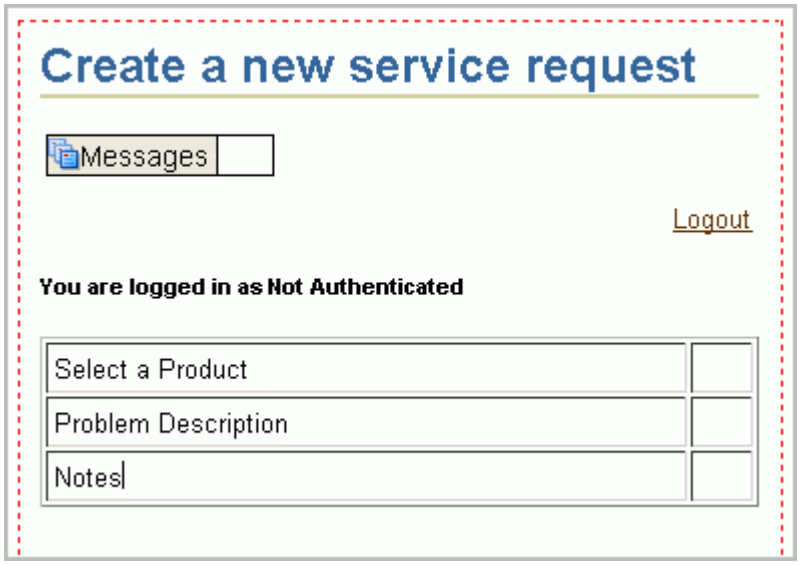
13. In the Property Inspector, set the **Align** property of the paragraph to **`right`** by selecting that value from the drop-down list.



14. On the next line, enter the text **`You are logged in as`**, and then press **[Space]**. From the Block Format drop-down list, select **Heading 5**.
15. In the Component Palette, click the **Output Text** component.
16. An output text field is created in the editor. In the Property Inspector for the output text, select the **Value** property and click **Bind to Data** .
17. In the Value dialog box, expand **JSF Managed Beans** and **UserInfo**. Select **userName** and click the right arrow to add it to the expression on the right. Click **OK**. This causes the application to display the name of the authenticated user at run time.
18. Select **HTML Common** from the Component Palette drop-down list, and then drag **Table** to the next line on the page in the visual editor to create a table on the page.
19. In the Insert Table dialog box, set the number of rows to **`3`** and click **OK**.
20. In the left column, enter the following three phrases in each of the three rows:

```
Select a Product
Problem Description
Notes
```

21. Select **CSS** from the Component Palette drop-down list, and then select **JDeveloper**. Your page should now look similar to the following screenshot:



22. Click **Save All** to save your work.

Adding Data Entry Components

Now you add UI components that ensure data accuracy and facilitate data entry. In this section, you create several components and add the code that enables users to:

- Select a product
- Enter text
- Commit or cancel

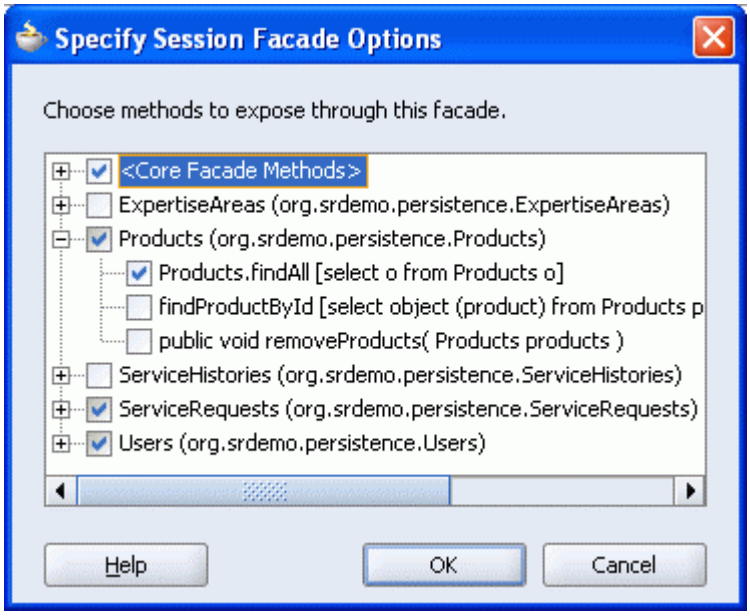
Enabling Users to Select a Product

To ensure that users enter a correct product ID, you implement a select list.

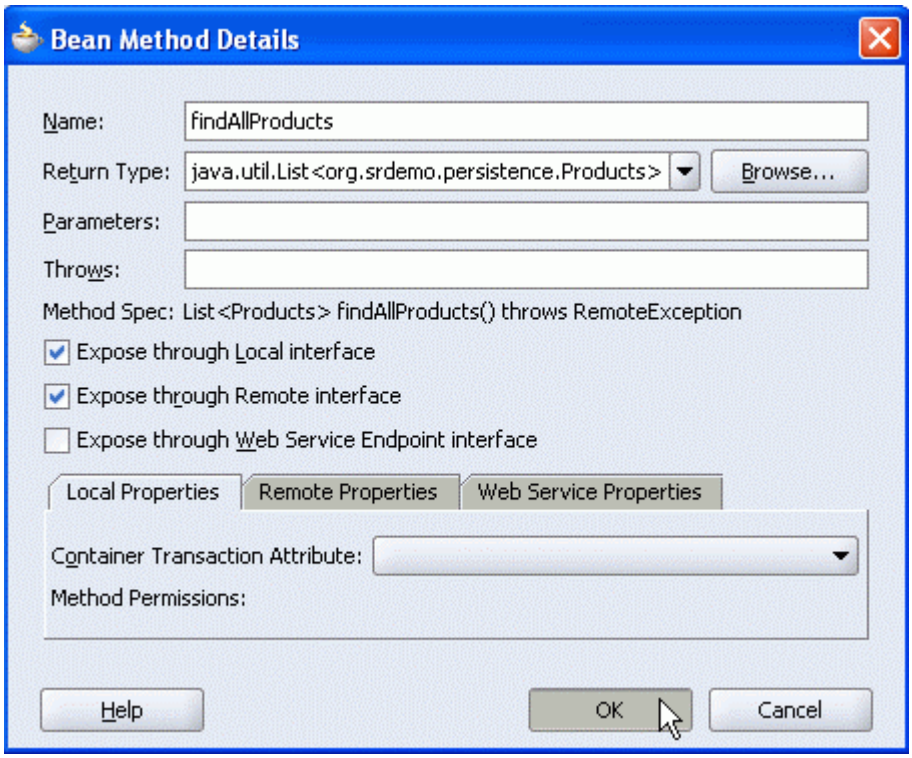
First you code the logic to retrieve a list of products in the SRCreate managed bean, and then you add a component to the page that calls the code in the managed bean. Implementing a product list also requires that a new method be added to the session facade.

Perform the following steps:

1. Add a new method to the session facade interfaces and bean. In the Applications Navigator, expand **Model**, **Application Sources**, and **org.srdemo.business**.
2. Right-click **ServiceRequestFacadeBean.java** and select **Edit Session Facade** from the context menu.
3. In the Session Facade Options dialog box, expand **Products** and select the check box next to **Products.findAll**. Click **OK**.



4. The **ServiceRequestFacadeBean.java** opens in the editor. In the Structure window, right-click the method that was just added, **queryProductsFindAll()**, and select Properties from the context menu.
5. In the Bean Methods Details dialog box, change the name of the method to **`findAllProducts`**, and then click **OK**.



When you change the method properties in this manner, the local and remote interface code remains synchronized with that of the session bean.

6. You can scroll to the end of the ServiceRequestFacadeBean.java file in the editor to see the method that was added.

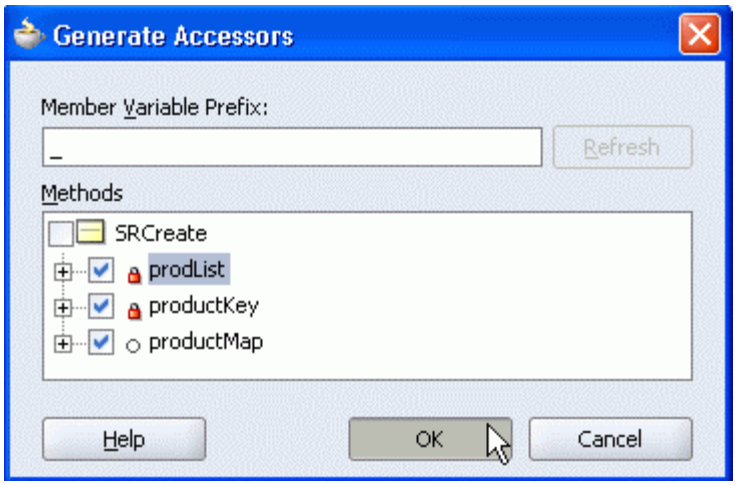
```
    /** <code>select object(o) from Products o</code> */
    public List<Products> findAllProducts() {
        return em.createNamedQuery("findAllProducts").getResultList();
    }
}
```

7. Click **Save All** to save your work.
8. Now that you have added code to the session facade, you can modify the backing bean for the SRCreate page to retrieve the product list so that it is available for a selection list component. In the Applications Navigator, expand **ViewController**, **Application Sources**, **org.srdemo.view**, and **backing**. Then double-click **SRCreate.java** to open it in the editor.
9. Add the following variable declarations:

```
private List prodList = new ArrayList();
private String productKey;
HashMap productMap = new HashMap();
```

```
package org.srdemo.view.backing;

import ...;

public class SRCreate {
    private HtmlForm form1;
    private HtmlCommandLink commandLink1;
    private HtmlOutputText outputText1;
import antlr.collections.List; ... (Alt-Enter)
    private HtmlOutputText outputText2;
    private List prodList = new ArrayList();
    private String productKey;
    HashMap productMap = new HashMap();
```

10. Press **[Alt]+[Enter]** when prompted to add import statements, choosing **`java.util.List`** for the `List` class.
11. Right-click in the editor window and select **Generate Accessors** from the context menu.
12. In the Generate Accessors dialog box, select the check boxes next to the **`prodList`**, **`productKey`**, and **`productMap`** variables and then click **OK**.



13. Toward the end of the file, modify the accessor methods for `prodlist`.

    First, modify the signature for the setter method. Locate the following line:

    ```
    public void setProdList(List prodList) {
    ```

    Change it to:

    ```
    public void setProdList(List<Products> prodList) {
    ```

    Then press **[Alt]+[Enter]** to accept the suggested import statement.

14. Change the signature for the getter method. Locate the following line:

    ```
    public List getProdList() {
    ```

    Change it to:

    ```
    public List<Products> getProdList() {
    ```

15. Change the body of the getter method. Locate the following line:

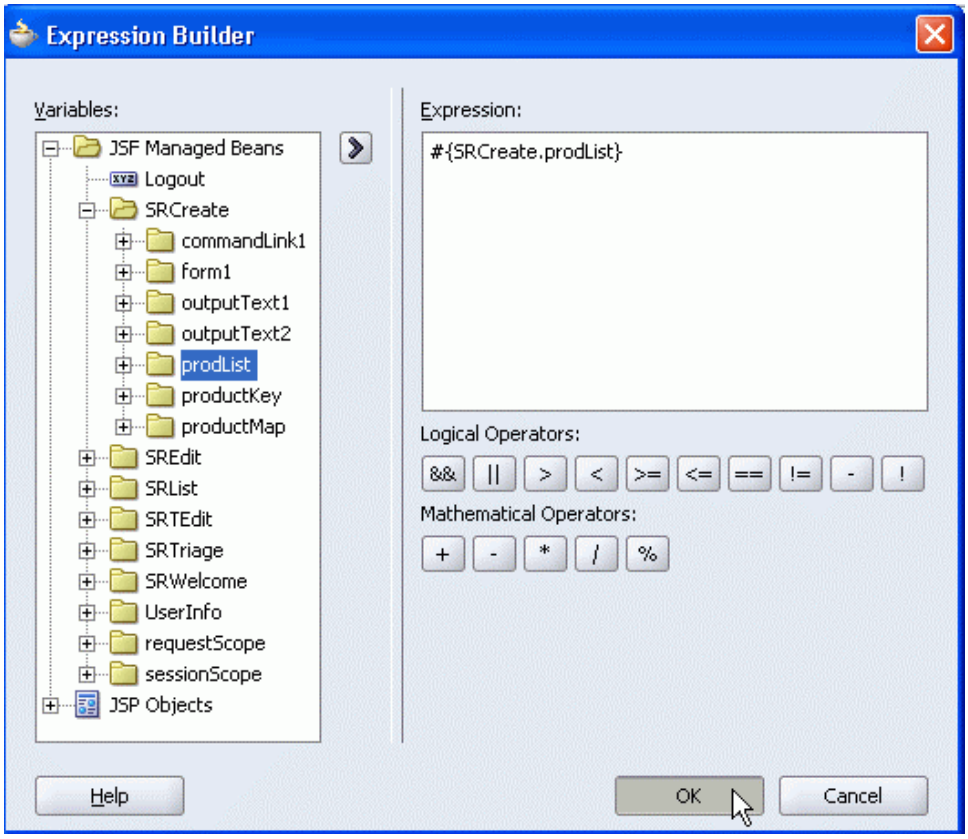
    ```
    return prodList;
    ```

    Replace it with the following lines:

    ```
    ServiceLocator serviceLocator = null;
    try {
        serviceLocator = serviceLocator.getInstance();
        ServiceRequestFacadeLocal srLocal =
            (ServiceRequestFacadeLocal)
            serviceLocator.getFacadeService
            ("java:comp/env/ejb/ServiceRequestFacade");
        List productsList =  srLocal.findAllProducts();
        Iterator it  = productsList.iterator();
             while(it.hasNext()){
                 Products product  = (Products)it.next();
                 SelectItem sItem = new SelectItem();
                 String pId =
                    product.getProdId().toString();
                 productMap.put(pId, product);
                 String prodName = product.getName();
                 sItem.setLabel(prodName);
                 sItem.setValue(pId);
                 prodList.add(sItem);
    ```

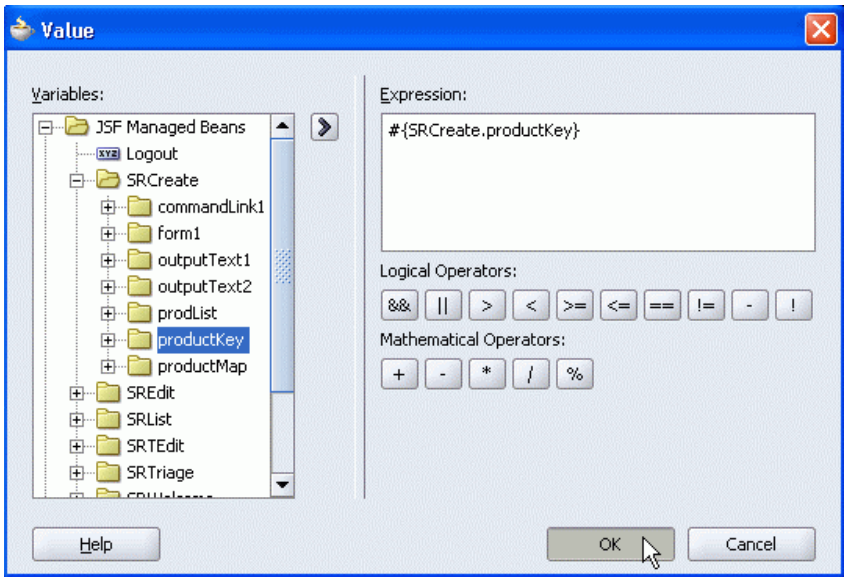
```
        }
    } catch (NamingException e) {
        e.printStackTrace();
    }
    return prodList;
```

16. Press **[Alt]+[Enter]** after each prompt to accept all of the suggested import statements.
17. Click **Save All** to save your work.
18. Now you can add the select list component to the page. Open or switch to **SRCreate.jsp** in the editor. In the Component Palette, select **JSF HTML** from the drop-down list and drag the **Listbox** component to the top-right cell of the table that you previously created on the page.
19. In the Insert Listbox dialog box, ensure that the **Bind to list (select items)** option is selected and then click **Bind**.
20. In the Expression Builder, expand **JSF Managed Beans** and **SRCreate**. Select **prodList** and click the right arrow to shuttle it to the Expression, and then click **OK**.



This appears in the list item of the list of products from the SRCreate managed bean.

21. Click the **Advanced Properties** tab.
22. Click into the **Value** property and click **Bind to Data** .
23. In the Value dialog box, expand **JSF Managed Beans** and **SRCreate**. Select **productKey** and click the right arrow to shuttle it to the Expression, and then click **OK**.



This sets the value of the list item to the product ID of the product that the user selects from the list.

24. Click **OK** to dismiss the Insert Listbox dialog box.

25. Click **Save All** to save your work.

Enabling Text Input

Next you add components to enable users to input a short description of the problem and to record notes. To enable text input, perform the following steps:

1. From the Component Palette, drag the **Input Text** component to the page in the visual editor to the cell below the list box. This creates an area where users can enter the problem description on a single line.

2. From the Component Palette, drag the **Input Textarea** component to the page in the visual editor to the cell below the Input Text. This creates an area where users can enter notes as multiple lines of text.

3. Click **Save All** to save your work.

Persisting or Canceling the Insert

Now you add code to the managed bean for the page that assigns the input values to the entity objects and then persists the data to the database. You add a button to call that code and another button to cancel the record insertion.

Perform the following steps:

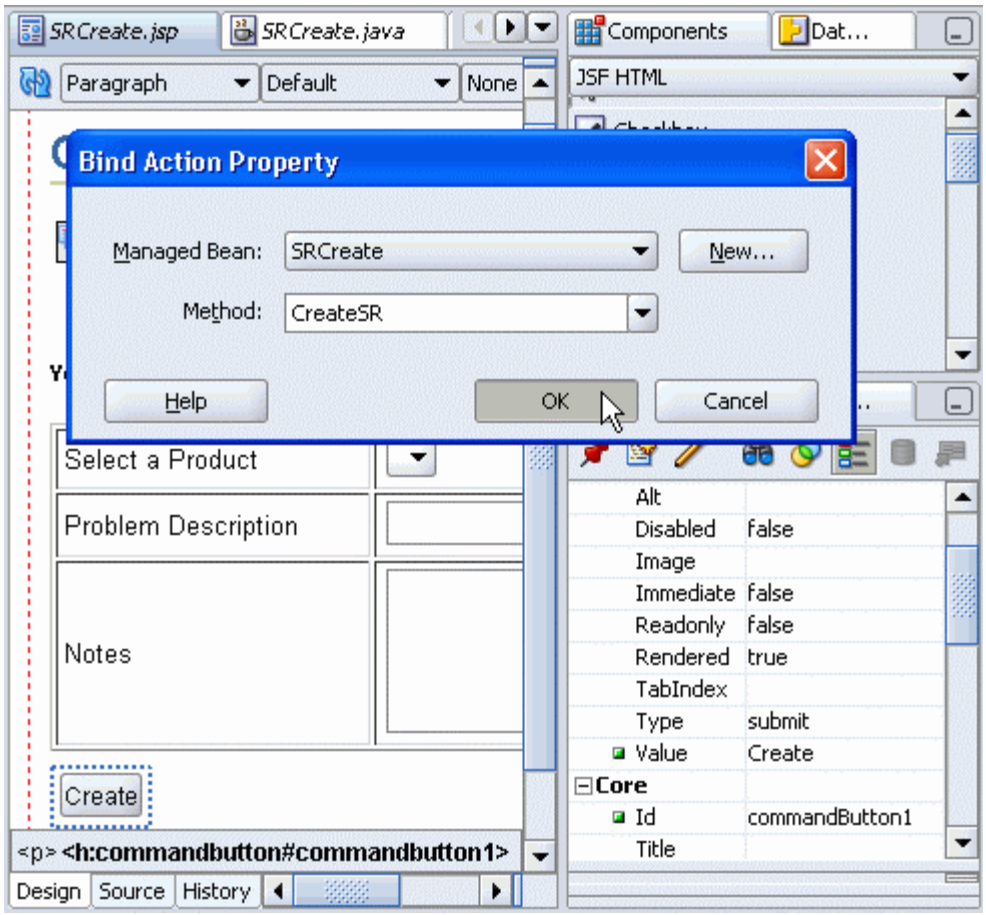
1. Open or switch to **SRCreate.java** (**ViewController** > **Application Sources** > **org.srdemo.view** > **backing**).

2. Add the following method that assigns the input fields to properties in the entities and then persists (commits) the data to the database:

```
public String CreateSR(){
    ServiceRequests sr = new ServiceRequests();
    sr.setStatus("Open");
```

```
        Products product = (Products)productMap.get(productKey);

        //sr.setProducts(getProduct());
        sr.setProducts(product);
        sr.setProblemDescription
            (getInputText1().getValue().toString());
        FacesContext ctx = FacesContext.getCurrentInstance();
        Application app = ctx.getApplication();
        ValueBinding curUser =
            app.createValueBinding("#{UserInfo.currentUser}");
        Users currentUser = (Users)curUser.getValue(ctx);
        sr.setUsers(currentUser);
        sr.setRequestDate(null);

        ServiceHistories svh = new ServiceHistories();
        svh.setServiceRequests(sr);
        svh.setNotes(getInputTextarea1().getValue().toString());
        svh.setLineNo(null);
        svh.setSvhDate(null);
        svh.setSvhType("Customer");
        svh.setUsers(currentUser);

//      sr.addToServiceHistoriesList(svh);
        List<ServiceHistories> svhList = new
            ArrayList<ServiceHistories>();
        svhList.add(svh);
        sr.setServiceHistoriesList(svhList);
        ServiceLocator serviceLocator = null;
        try {
            serviceLocator = serviceLocator.getInstance();
            ServiceRequestFacadeLocal srLocal =
                 (ServiceRequestFacadeLocal)
                 serviceLocator.getFacadeService
                 ("java:comp/env/ejb/ServiceRequestFacade");
            srLocal.persistEntity(sr);

            return "created";
        } catch (NamingException e) {
            // TODO
            e.printStackTrace();
        }
        return "created";
    }
```

3. Press **[Alt]+[Enter]** after each prompt to accept import statements, choosing to import `javax.faces.application.Application` for the Application class.
4. Switch to **SRCreate.jsp** in the editor. Place the cursor after the table within the red dotted line of the form and press **[Enter]** to create a blank line.
5. In the Component Palette, select **JSF HTML** from the drop-down list and click **Command Button**.
6. In the Property Inspector for the command button, set the Value property to `Create`.
7. Double-click the **Create** button in the editor, and in the Bind Action Property dialog box, select **CreateSR** from the Method drop-down list. Click **OK**.



Note: The `CreateSR()` method in the SRCreate managed bean persists the data and returns the string `created`, which is the name of the navigation case that you defined in the `faces-config.xml` file to navigate to the SRList page.

8. From the Component Palette, drag a **Command Button** to the right of the Create button in the editor for SRCreate.jsp.
9. In the Property Palette, set the Value property to **`Cancel`**. Select **created** from the Action property drop-down list so that the user navigates to the SRList page without executing the code in the managed bean that creates and saves the record.
10. Click **Save All** to save your work.

Testing the Insert Logic and the UI Components

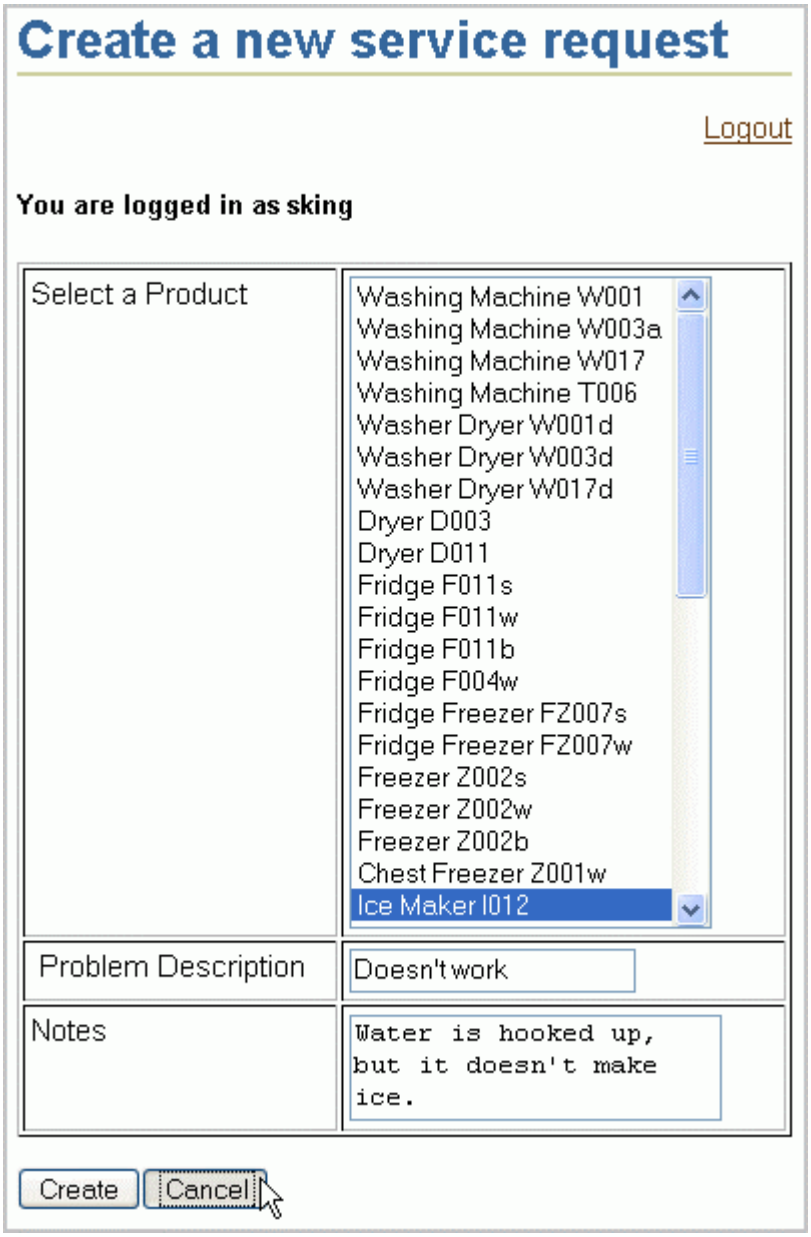
Now that you have defined insert logic and created the necessary UI components to insert new service requests, you can test the functionality by performing the following steps:

1. In the Applications Navigator, expand the **ViewController** project and the **Web Content** node.
2. Right-click **index.jsp** and select **Run** from the context menu.
3. A browser opens to display the Logon page. Enter the following logon information and click **Sign On**:

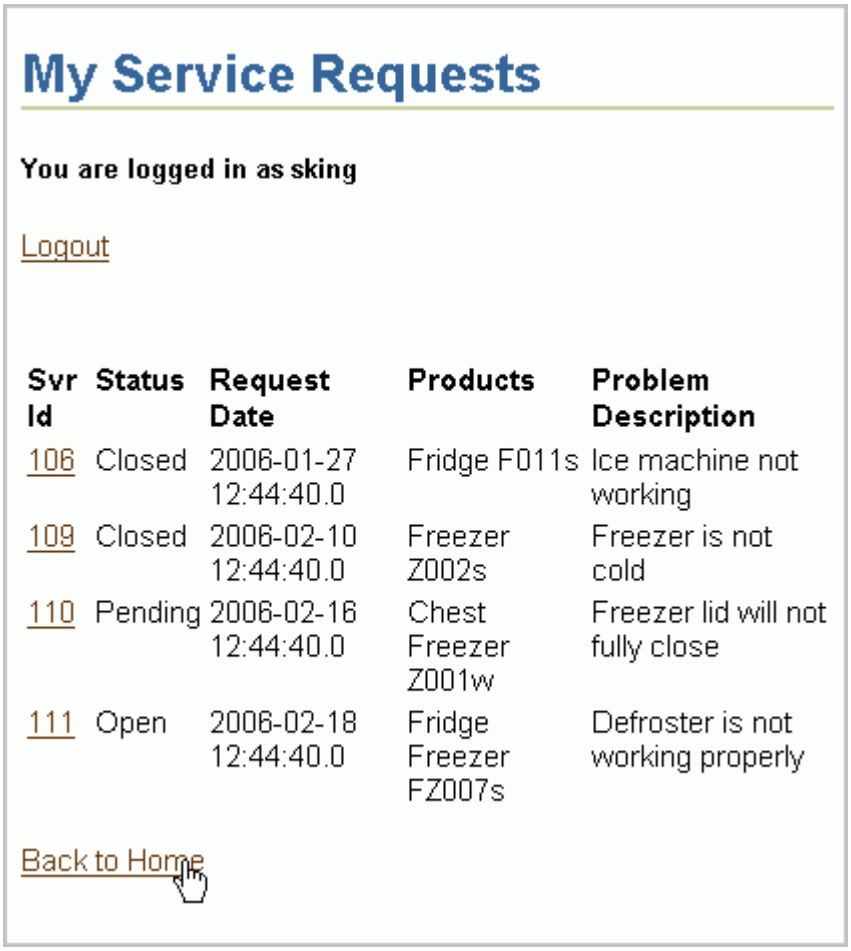
Username	Password

Username	Password
sking	welcome

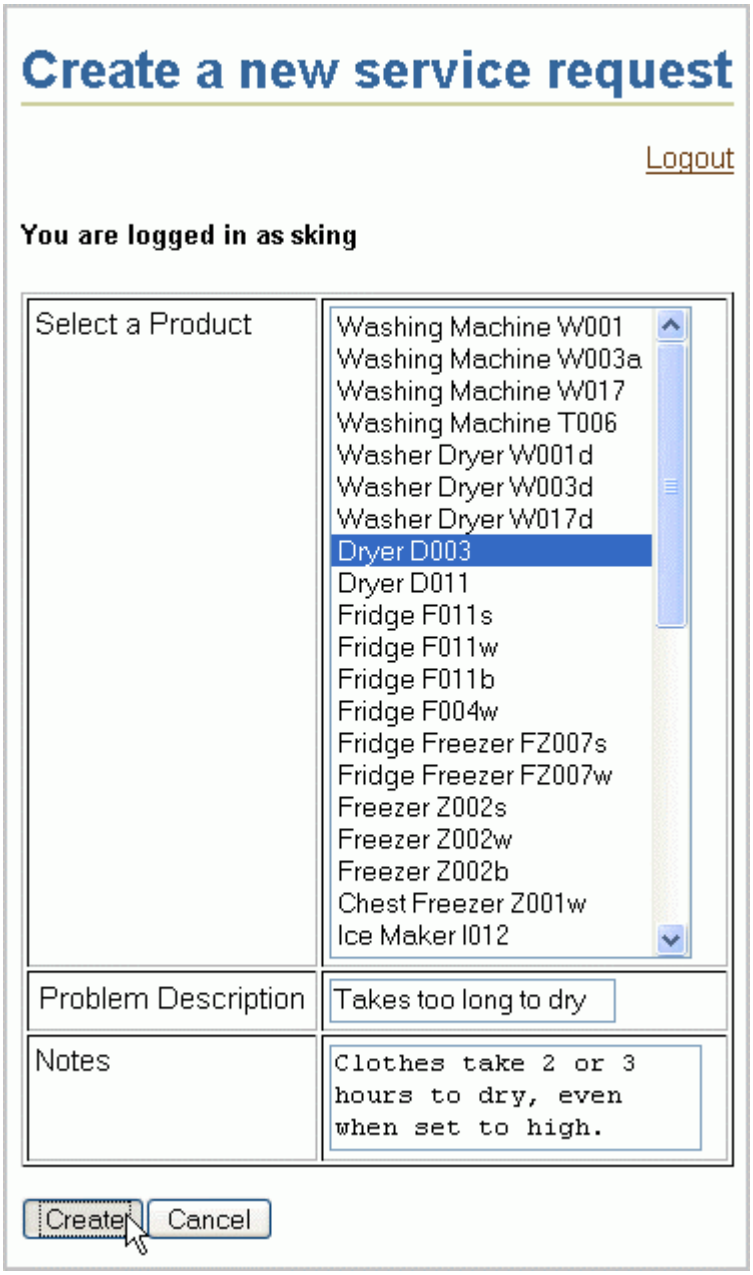
4. On the SRWelcome page, click **Create a New Service Request**.
5. On the Create page, select a product and enter a problem description and notes, and then click **Cancel**.



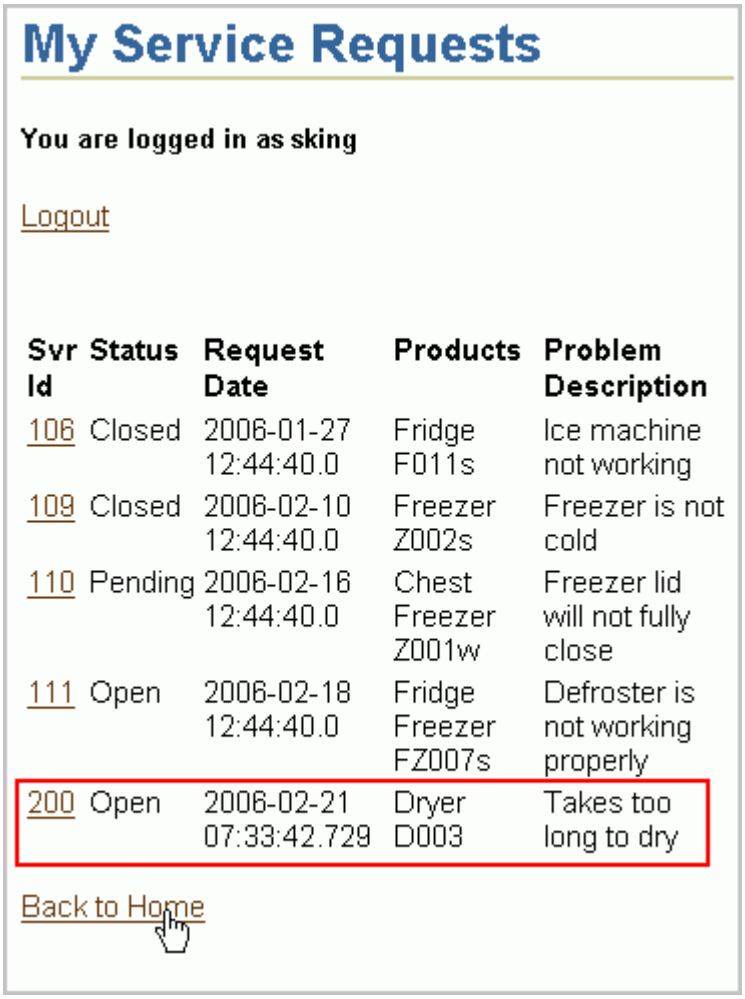
6. The SRList page should display without showing the data that you entered because you did not commit it. Click the **Back to Home** link.



7. On the SRWelcome page, again click **Create a New Service Request**.
8. Select a product and enter a problem description and some notes, and then click **Create**.



9. The SRList page should appear and show the service request that you just created.



10. To confirm that the data has been committed to the database, in JDeveloper click the **Connections Navigator** tab and then expand the **SRDemo**, **SRDEMO**, and **Tables** nodes.
11. Double-click the **SERVICE_HISTORIES** table to open it, and then click the **Data** tab. You should see the service request that you just created. (If you already had the table open, you may need to select **View** > **Refresh** to see the new data.)

SERVICE_REQUESTS

	SVR_ID	STATUS	REQUEST_D...	PROBLEM_D...	PROD_ID	CREATED_E
6	105	Closed	2006-2-10.1...	Air in dryer ...	107	312
7	106	Closed	2006-1-27.1...	Ice machine ...	109	321
8	107	Pending	2006-2-15.1...	Fridge is lea...	111	310
9	108	Open	2006-2-16.1...	Freezer full ...	114	304
10	109	Closed	2006-2-10.1...	Freezer is n...	115	320
11	110	Pending	2006-2-16.1...	Freezer lid w...	118	318
12	111	Open	2006-2-18.1...	Defroster is ...	113	325
13	112	Open	2006-2-18.1...	My Dryer do...	111	306
14	113	Open	2006-2-19.1...	I'm getting a...	100	303
15	114	Open	2006-2-19.1...	There seems...	100	309
16	200	Open	2006-2-21.7...	Takes too lo...	107	300

Structure Data

12. Now double-click the **SERVICE_HISTORIES** table in the Connections Navigator and click the **Data** tab. You should see the newly created record.

SERVICE_REQUESTS SERVICE_HISTORIES

	SVR_ID	LINE_NO	SVH_DATE	NOTES	SVH
13	105	2	2006-2-14...	Gas line was not pr...	Customer
14	106	1	2006-1-28...	Customer did not h...	Technician
15	107	1	2006-2-18...	Customer will call b...	Technician
16	108	1	2006-2-19...	Researching issue	Technician
17	109	1	2006-2-11...	Asked customer to ...	Technician
18	109	2	2006-2-13...	Freezer is plugged i...	Customer
19	109	3	2006-2-13...	Asked customer to ...	Technician
20	109	4	2006-2-14...	Freezer is now cold	Customer
21	110	1	2006-2-17...	Asked customer to ...	Technician
22	111	1	2006-2-19...	Referring to troubl...	Technician
23	200	1	2006-2-21...	Clothes take 2 or 3 ...	Customer

Structure Data

Summary

In this chapter, you created the page that enables users to create a new service request in the database. To accomplish this, you performed the following key tasks:

- Created the SRCreate page with standard logic for displaying the user ID and for logging out
- Added a table to lay out the components
- Added data entry UI components to the page
- Added logic to the managed bean for the page to display the UI and to insert the records in the database
- Bound the UI components to the logic in the managed bean
- Tested the insert logic and the UI components

12

Installing Oracle Application Server 10*g*

The first part of this tutorial showed you how use Oracle JDeveloper to build a Java EE application and deploy it to the embedded OC4J container that is part of JDeveloper. In a production environment, however, you must deploy to an application server. The last two chapters of this tutorial pertain to deploying the SRDEMO application into Oracle Application Server 10*g* (10.1.3.1).

This chapter contains the following topics:

- Introduction
- Downloading Oracle Application Server
- Installing the First Oracle Application Server Instance
- Installing the Second OC4J Instance
- Launching the Application Server Control Console
- Summary

Introduction

Before you begin to install and deploy to Oracle Application Server, it is helpful to understand how Oracle Application Server enables you to deploy your EJB 3.0 application. In this introductory section, you learn answers to the following questions:

- What is Oracle Application Server?
- What are the benefits of using OC4J?
- How are requests routed to the application?

In this chapter, you perform the following key tasks:

- Download Oracle Application Server 10*g*
- Install two instances of Oracle Application Server, ideally on a single host machine
- Cluster the two instances together
- Deploy the application across the cluster
- Configure HTTP session state replication between the application instances running within the cluster

In this tutorial, you install two instances of Oracle Application Server 10*g* so that you can see such features as clustering, state replication, and rolling application upgrades.

Ideally, you should install both Oracle Application Server instances on the same physical host machine.

Oracle Application Server

Oracle Application Server 10*g*, a component of Oracle Fusion Middleware, represents the next generation of enterprise application server. With its service-oriented architecture foundation and grid-enabled deployment infrastructure, Oracle Application Server enables on-demand computing across an enterprise by efficiently pooling and utilizing all of the hardware resources (CPU, memory, storage) prescribed to it from within an enterprise.

In this tutorial, you work with the following Oracle Application Server components:

- **Oracle Application Server Containers for J2EE (OC4J):** OC4J is the core of Oracle Application Server and provides the containers that applications actually run within. As part of this tutorial, you create a small cluster comprised of two OC4J instances.
- **Oracle HTTP Server (OHS):** OHS serves as a front-end Web listener, forwarding Web requests to OC4J instances hosting the requested applications. You install a single OHS instance as part of your cluster.
- **Application Server Control:** This application provides a Web-based user interface for managing the Oracle Application Server component instances—including OC4J and OHS—within a cluster. You install a single instance of this application on one of your OC4J instances

Each Oracle Application Server instance may include one or more OC4J instances. Such OC4J instances are referred to as *managed* instances.

Benefits of OC4J

At the core of Oracle Application Server is Oracle Application Server Containers for J2EE (OC4J). The latest release, OC4J 10*g* Release 3 (10.1.3), provides a fully Java EE-compliant environment for creating and hosting secure, portable, high-performing applications. It provides all the containers, APIs, and services mandated by the Java EE specification, including:

- A Web container providing a run-time environment for JavaServer Pages (JSPs) and servlets

- An Enterprise JavaBeans (EJB) container that supports the EJB 2.1 specification as well as the new EJB 3.0 specification
- A complete design and run-time environment for Web services

In addition, OC4J provides full implementations of the standard Java EE services, including:

- A feature-rich implementation of the Java Message Service (JMS), which enables components running within the container to send and receive messages
- Data access and persistence through the Java Database Connection API (JDBC). You learn how to create data sources for connecting to databases in this tutorial.
- Transaction management through an implementation of the Java Transaction API
- Component lookup via the Java Naming and Directory Interface (JNDI)
- Application security—authorizing and authenticating application users—through implementation of the Java Authentication and Authorization Service (JAAS)
- Connectivity with legacy systems through the Java EE Connector Architecture (JCA)

Routing Requests to the Application

Web communications with OC4J can also be managed through Oracle HTTP Server (OHS), which serves as a front-end listener, and the `mod_oc4j` module, which forwards HTTP requests to OC4J server instances using the Apache JServ Protocol (AJP) 1.3.

The OHS-OC4J request/response flow is as follows:

1. The OHS listener receives an incoming HTTP request.
2. OHS passes the request to a native Web listener configured within the OC4J instance. The connection between OHS and OC4J uses the Apache JServ Protocol (AJP) on a port number negotiated during OC4J startup.

Incoming requests are routed to an OC4J instance serving the requested application using *mount points*, which are stored in OHS. For example, a mount point essentially maps the URL http://<host>:<port>/SRDEMO to the `home` instances hosting the SRDEMO application within your cluster.

Mount points are dynamically created at the time applications are deployed. Similarly, mount points for un-deployed applications are automatically removed.

Requests that come in for specific mount points are routed to the OC4J instance corresponding to that mount point. Dynamic mount-point configuration is a major enhancement provided by OC4J because it ensures that requests are never mistakenly forwarded to a server that does not contain an active instance of the requested application.

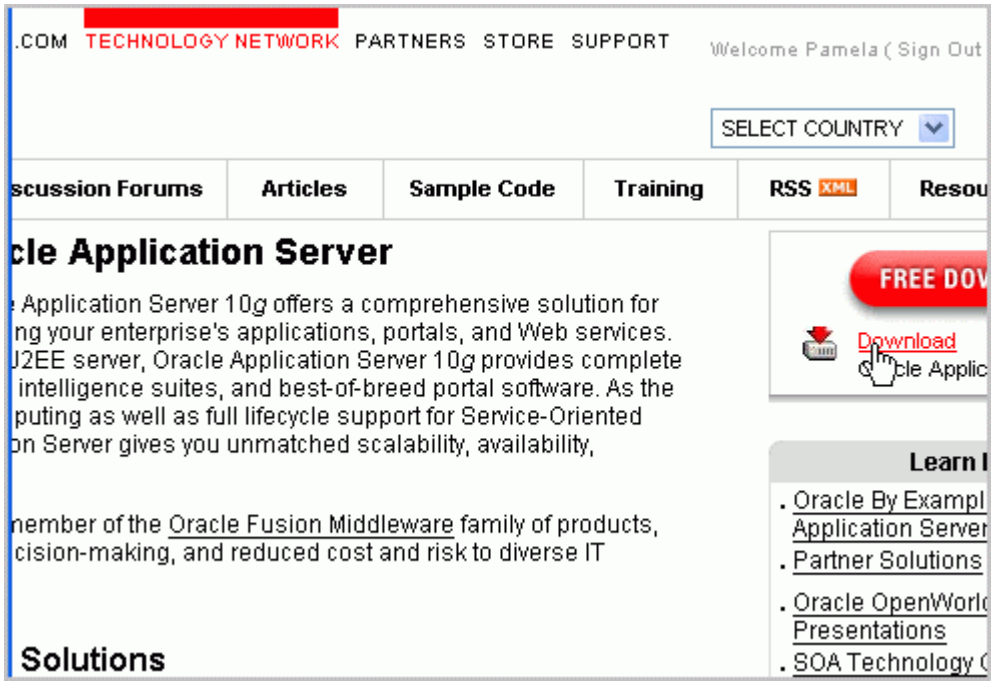
Downloading Oracle Application Server

If you do not have an installation CD, you can download Oracle Application Server 10*g* from Oracle Technology Network (OTN). To download, perform the following steps:

1. Open a browser and enter the following URL:

   http://www.oracle.com/technology/products/ias/index.html

2. Click the **Download** link.



Note: The appearance of the Web page may not be identical to what is shown in the screenshot because content on OTN may change.

3. On the download page, click the link for **Oracle Application Server SOA Suite (10.1.3.1.0)**.
4. After accepting the license terms and export restrictions, click the link that is applicable to your platform (Windows or Linux). The examples in this chapter use the Windows version; instructions for the Linux version may differ. You should consult the installation guide before installing.
5. When the download has finished, extract the files to a directory of your choosing.

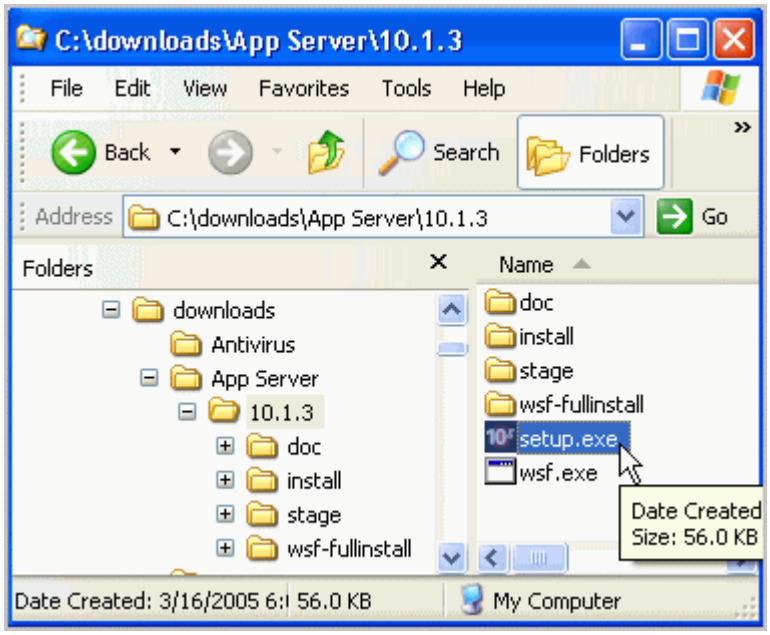
Installing the First Oracle Application Server Instance

The first instance that you install includes each of the following components:

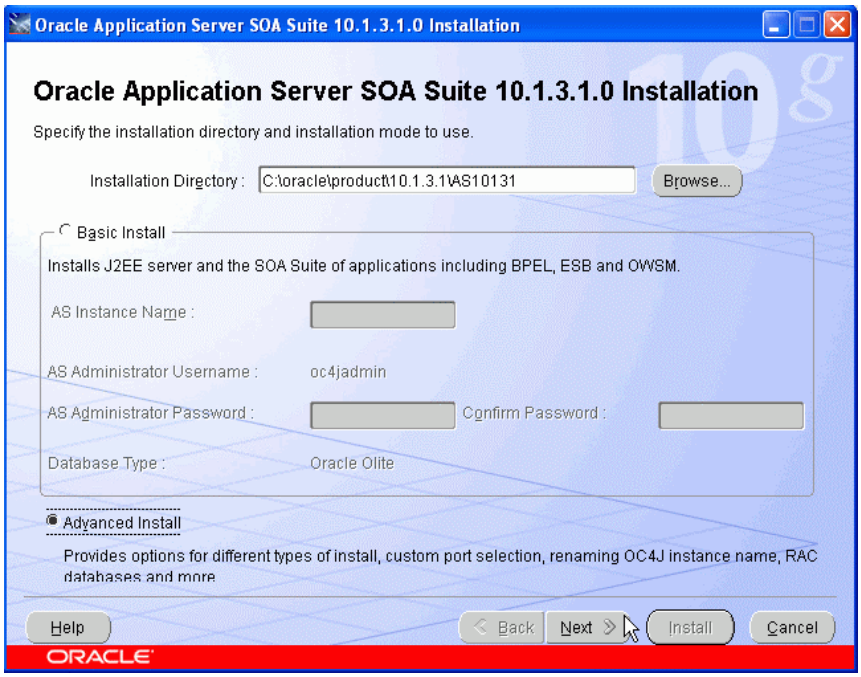
- Oracle Containers for J2EE (OC4J), providing a complete Java EE-compliant environment
- Oracle HTTP Server, which listens for incoming Web requests and routes them to an OC4J instance hosting the requested application
- Oracle Process Manager and Notification Server (OPMN) which manages the various Oracle Application Server components, including OC4J and OHS

To perform the installation, perform the following steps:

1. Launch the Oracle Universal Installer. In Windows you can do this by double-clicking **setup.exe** in the directory where you extracted the installation files.



2. On the Installation screen of the installer, click **Browse** to navigate to the Oracle home directory where the application server should be installed. Be sure to also indicate a new subdirectory for this application server version. This entire path is referred to elsewhere in this tutorial as the application server home directory, or *<AS_HOME>*. Select the **Advanced Install** option and click **Next**.



3. On the Select Installation Type screen of the installer, select the option **J2EE Server and Web Server**. Click **Next**.

4. On the Specify Port Configuration Options screen of the installer, select the **Automatic** option and click **Next**.

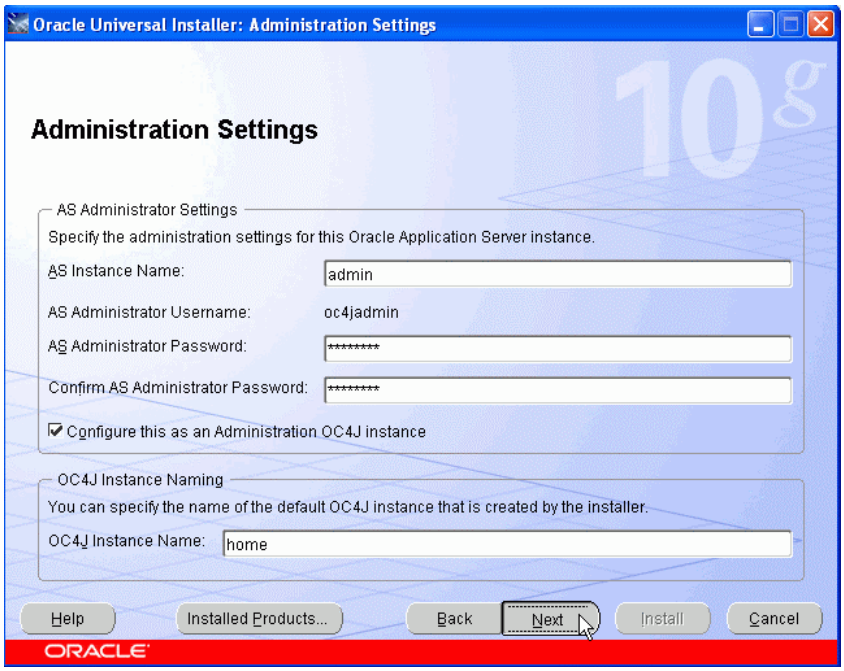
   This enables the OPMN component to dynamically select the ports to use upon installation.

5. On the Administration Settings screen of the installer, specify the Oracle Application Server instance name. Because this is the administration instance, you may want to use **`admin`** (as the example does).

   Specify an Administrator Account Password (at least one numeric character is required) to be associated with the default `oc4jadmin` administrator account. The example uses **`welcome1`** as the password.

   Select the **Configure this as an Administration OC4J Instance** check box.

   For convenience, you can accept the default name **`home`** for the OC4J instance that runs within this Oracle Application Server instance. Click **Next**.

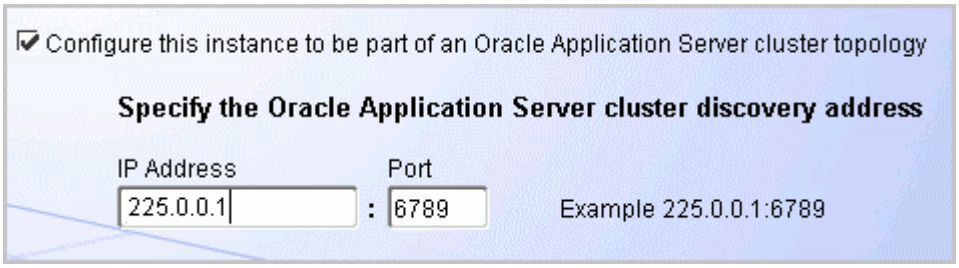


6. On the Cluster Topology Configuration screen of the installer, select the **Configure this instance to be part of an Oracle Application Server cluster topology** check box. Then provide a multicast address and port combination to be used to add the OHS and OC4J instances to a cluster. The multicast address must be within the valid address range, which is 224.0.1.0 to 239.255.255.255.

   In this step, you are enabling the *dynamic discovery* mechanism, which allows Oracle Application Server components to dynamically discover and communicate with one another via multicast messages. The cluster is essentially able to manage itself, as each Oracle Application Server node is updated when other nodes broadcasting on the same address:port combination are added or removed from the cluster via the multicast broadcasts.

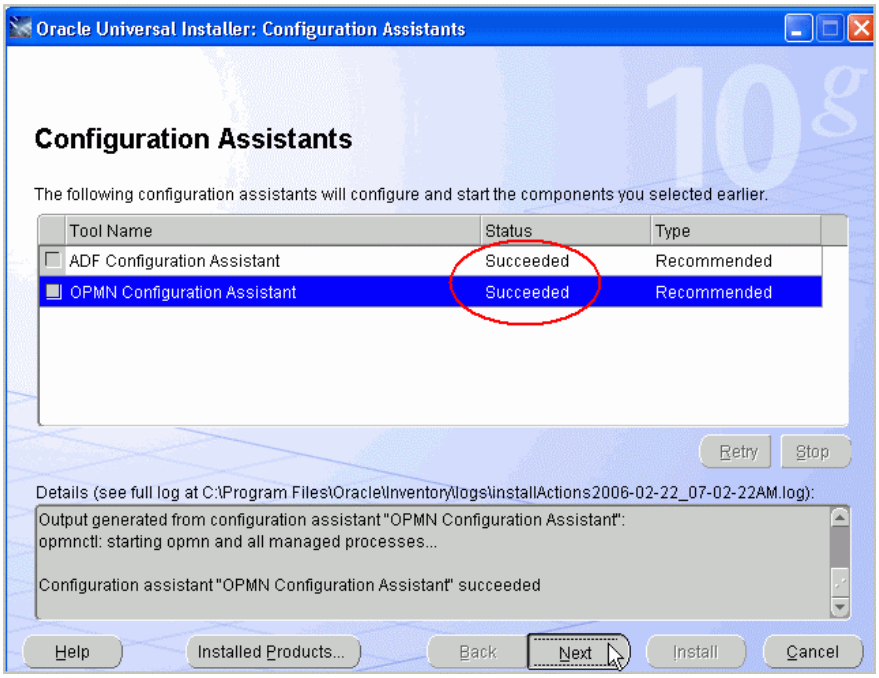
   Note that clustering does not have to be required at installation time, nor is multicast-based dynamic discovery the only clustering option supported. However, because this is the simplest clustering scheme to implement, we are using it here. See the *Oracle Containers for J2EE Configuration and Administration Guide* for complete details on cluster configuration: http://download-west.oracle.com/docs/cd/B25221_02/web.1013/b14432.pdf.

   The example uses an IP address of **`225.0.0.1`** and a port of **`6789`**.



Click **Next**.

7. On the Summary screen of the installer, click **Install** to install the instance.
8. When installation is complete, you should see a Succeeded status on the Configuration Assistants screen of the installer for the two configuration assistants. Click **Next**.



9. The End of Installation screen lists a variety of valuable information, including:
   - The host and port information required to access the Oracle HTTP Server instance
   - The URL used to access the Application Server Control Console management interface

   This information is also contained in the `<AS_HOME>\install\readme.txt` file.

   Click **Exit** when finished to exit the installer. A "Welcome" page launches in your Web browser, providing links to online documentation and additional resources. You can also launch Application Server Control from this page, although you will do that later in the tutorial.

   If you want to see details of the actions performed by the installer, after closing the installer you can open the `C:\Program Files\Oracle\Inventory\logs\installActions<datetime>.log` file.
10. As part of the installation process, all of the Oracle Application Server components that you just installed (OC4J, OHS, and the OPMN management component) are started by default. To verify that these components are running, open a command prompt and type the following command:

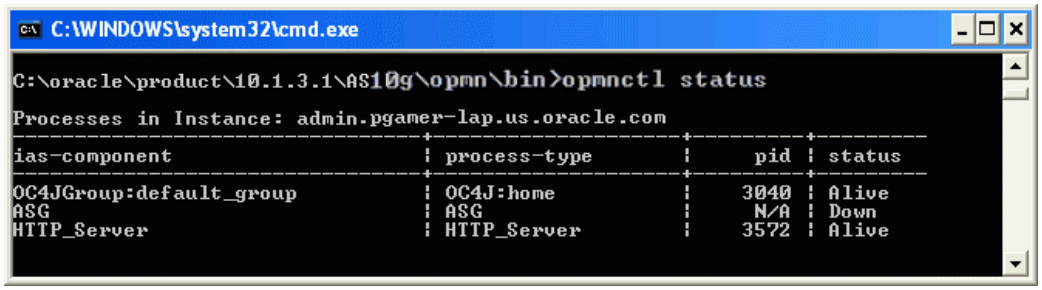
```
<AS_HOME>/opmn/bin/opmnctl status
```

Here is an example:

```
C:\oracle\product\10.1.3.1\AS10g\opmn\bin\opmnctl status
```

If the components are running, you should see output similar to the following:

```
Processes in Instance: admin.<host.domain>
-------------------+--------------------+---------+---------
ias-component      | process-type       |     pid | status
-------------------+--------------------+---------+---------
OC4J               | home               |    7108 | Alive
HTTP_Server        | HTTP_Server        |     596 | Alive
```



Type **`exit`** to exit the command prompt window.

Note: You must be connected directly to the network, rather than through a virtual private network (VPN), in order to use the multicast type of clustering that is described in this chapter. If you are connected via VPN, you can use peer-to-peer clustering as described in the *OC4J Configuration and Administration Guide* (http://download-west.oracle.com/docs/cd/B25221_02/web.1013/b14432.pdf) in the section "Configuring Static Discovery Servers." This tutorial uses multicast (not peer-to-peer) clustering.

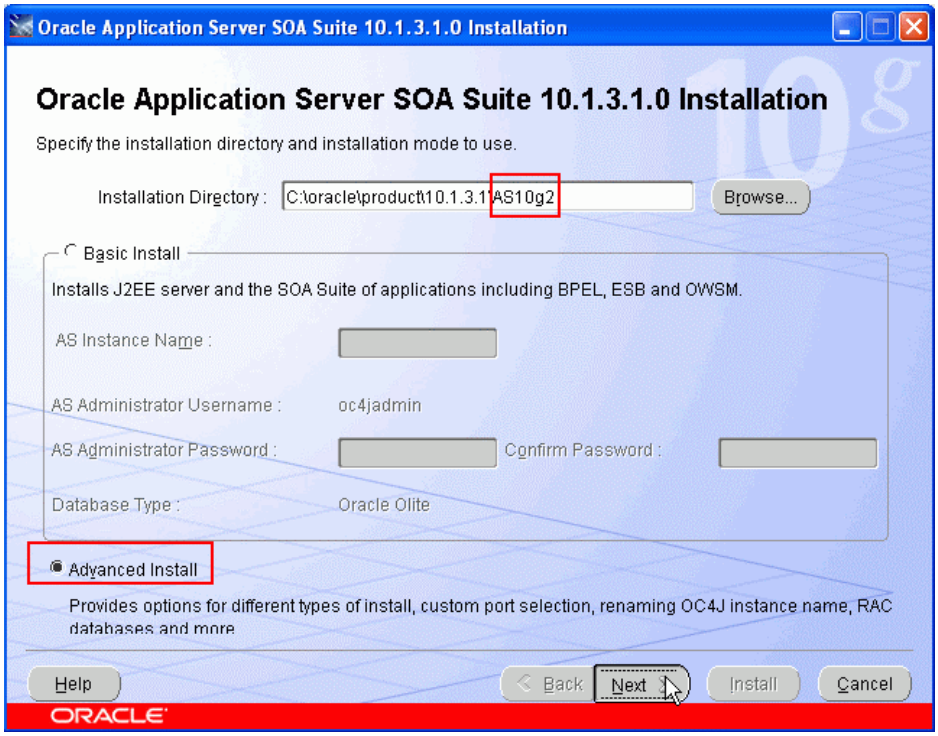
Installing the Second OC4J Instance

Next you install the second Oracle Application Server instance. This time, however, you install only the OC4J and OPMN components; the Oracle HTTP Server installed in the initial instance is used to receive and forward requests within the cluster.

To install the second OC4J instance, perform the following steps:

1. Launch the Oracle Universal Installer as you did previously.

   On the Installation screen of the installer, click **Browse** to navigate to the same application server home directory as you did previously, and then add a number to the subdirectory to distinguish it from the previous application server home directory. This entire path is referred to elsewhere in this tutorial as *<AS2_HOME>*. Select the **Advanced Install** option and click **Next**.



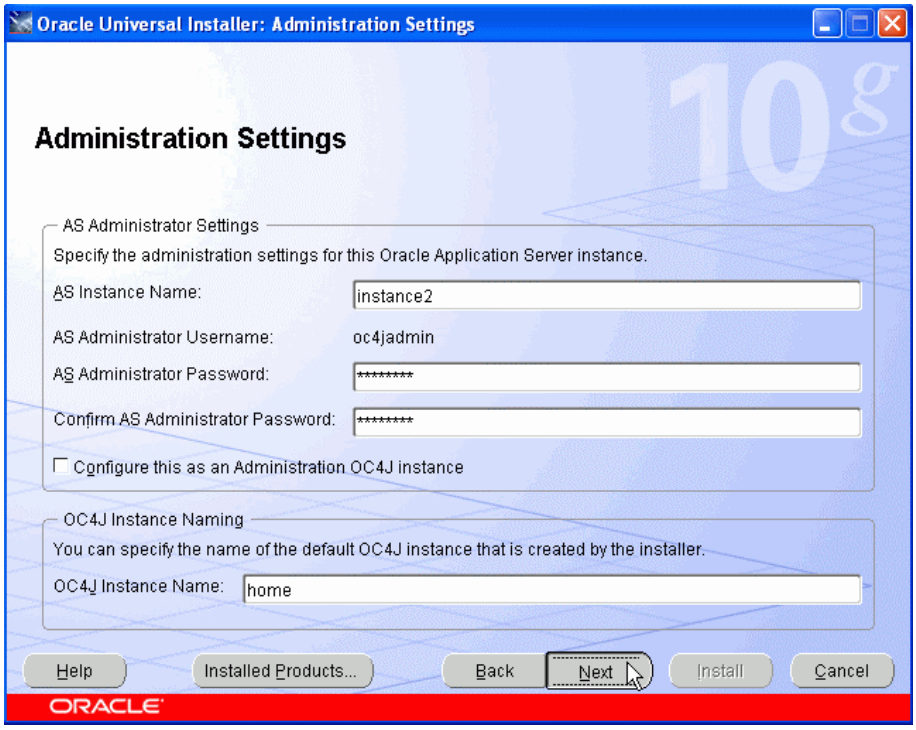
2. On the Select Installation Type screen of the installer, select the **J2EE Server** option. Click **Next**.
3. On the Specify Port Configuration Options screen of the installer, select the **Automatic** option and click **Next**.
4. On the Administration Settings screen of the installer, specify the Oracle Application Server instance name. Because this is the second instance, you may want to use **`instance2`** (as the example does).

   Specify the same Administrator Account Password that you specified for the administrator account. This is required to enable you to install across all nodes in the cluster. The example uses **`welcome1`** as the password.

   This time do *not* select the "Configure this as an Administration OC4J Instance" check box. Because this OC4J is managed by the other instance hosting Application Server Control, you disable this application on the new instance.

   Specify the same OC4J instance name that you specified in the first instance. The example uses **`home`** as the OC4J instance name. This enables you to install your application to the same named instance group within the cluster.

   Click **Next**.



5. On the Cluster Topology Configuration screen of the installer, select the **Access this OC4J Instance from a separate Oracle HTTP Server** check box to enable the local Web listener of this OC4J instance to receive requests routed from your OHS instance.

   Select the **Configure this OC4J instance to be part of an Oracle Application Server cluster topology** check box. Then provide the same multicast address and port combination as you set for the first instance.

6. Click **Next**.

7. On the Summary screen of the installer, click **Install** to install the instance.

8. When installation is complete, you should see a Succeeded status on the Configuration Assistants screen of the installer for the two configuration assistants. Click **Next**.

9. The End of Installation screen lists a variety of valuable information; this information is also contained in the file `<AS2_HOME>\install\readme.txt`. Click **Exit** when finished to exit the installer.

10. If you want to see details of the actions performed by the installer, after closing the installer you can open the `C:\Program Files\Oracle\Inventory\logs\installActions<datetime>.log` file.

11. The OC4J instance is started at the end of the installation process. To verify this, open a command prompt and type the following command:
    `<AS2_HOME>/opmn/bin/opmnctl status`

    Here is an example:
    `C:\oracle\product\10.1.3.1\AS10g2\opmn\bin\opmnctl status`

```
C:\WINDOWS\system32\cmd.exe
C:\oracle\product\10.1.3.1\AS10g2\opmn\bin>opmnctl status

Processes in Instance: instance2.pgamer-lap.us.oracle.com
-------------------------------+--------------------+---------+---------
ias-component                  | process-type       |     pid | status
-------------------------------+--------------------+---------+---------
OC4JGroup:default_group        | OC4J:home          |    5896 | Alive
ASG                            | ASG                |     N/A | Down

C:\oracle\product\10.1.3.1\AS10g2\opmn\bin>
```

You have effectively clustered your two Oracle Application Server instances.

Launching the Application Server Control Console

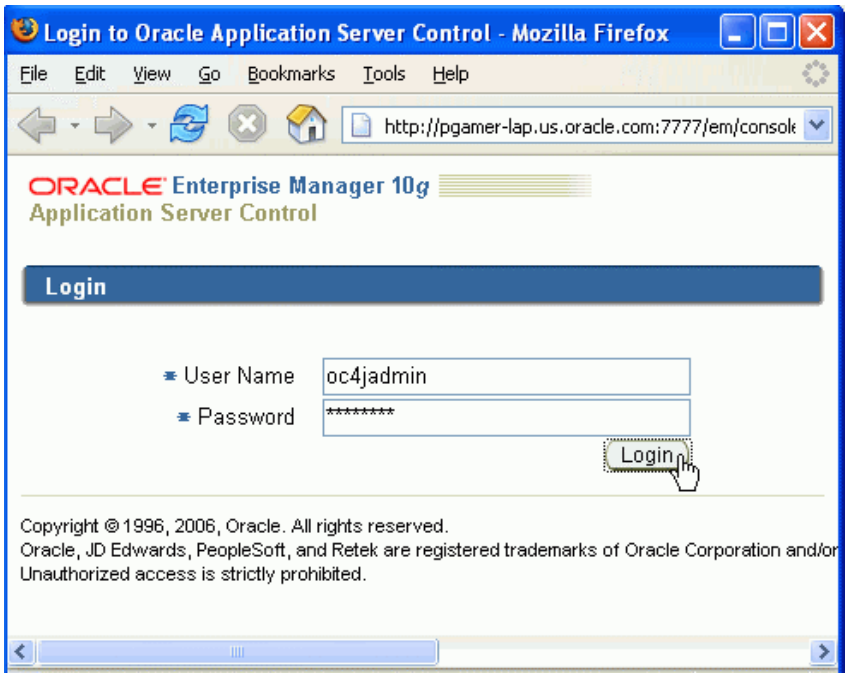
Now you launch Application Server Control, the Web-based user interface for deploying, configuring, and monitoring applications within OC4J. With Application Server Control, you also can manage the OC4J server instance and the Web services used by your application.

Perform the following steps:

1. Open a Web browser. Supply your host name and the OHS port noted at the end of installation in the following URL in your Web browser: `http://<hostname>:<port>/em`.

   Example: `http://myhost.oracle.com:7777/em/`

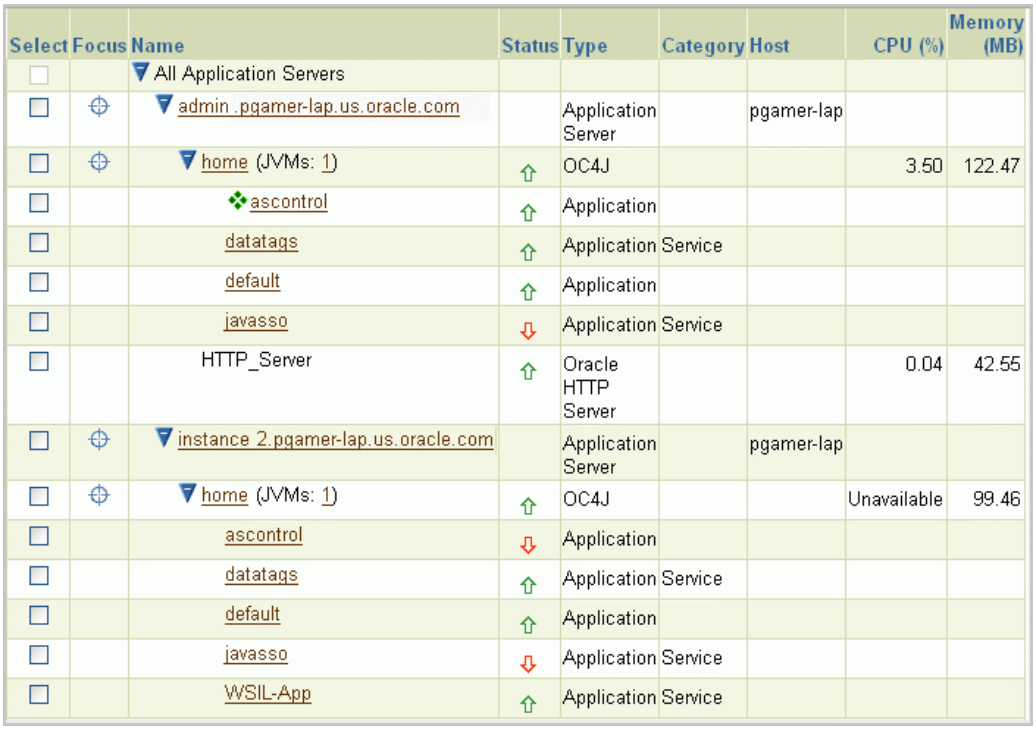
2. Log in to Application Server Control by providing the password that you specified when installing the Application Server (**`welcome1`** in the example), and then click **Login**.



3. The top-level Cluster Topology page provides you with a view of your clustered environment. The top of the page shows an overview of the hosts, servers, and instances.

4. On the Members section of the page, you can see your two Oracle Application Server instances. Expand each instance to drill down to the components installed on each. Expand each OC4J instance (the `home` nodes) to see installed applications.

   Note the "Groups" section at the bottom of the page. A group is a synchronized set of OC4J instances that belong to the same cluster topology. Because the OC4J instances you created are members of the same cluster, they implicitly belong to the `default_group` of instances. You will see how operations can be performed on groups in the next chapter of this tutorial.



Select	Focus	Name	Status	Type	Category	Host	CPU (%)	Memory (MB)
		All Application Servers						
		admin.pgamer-lap.us.oracle.com		Application Server		pgamer-lap		
		home (JVMs: 1)		OC4J			3.50	122.47
		ascontrol		Application				
		datatags		Application	Service			
		default		Application				
		javasso		Application	Service			
		HTTP_Server		Oracle HTTP Server			0.04	42.55
		instance 2.pgamer-lap.us.oracle.com		Application Server		pgamer-lap		
		home (JVMs: 1)		OC4J			Unavailable	99.46
		ascontrol		Application				
		datatags		Application	Service			
		default		Application				
		javasso		Application	Service			
		WSIL-App		Application	Service			

The application server is now ready for you to deploy applications.

Note: If you are unable to see both Application Server instances in the cluster, it may be because you are not connected directly to the network. If you are connected via VPN, you can use peer-to-peer clustering as described in the *OC4J Configuration and Administration Guide* (http://download-west.oracle.com/docs/cd/B25221_02/web.1013/b14432.pdf) in the section "Configuring Static Discovery Servers." This tutorial uses multicast (rather than peer-to-peer) clustering.

Summary

In this chapter, you prepared for deployment of the application by installing Oracle Application Server 10*g*.

In short, here are the key tasks that you performed in this chapter:

- Downloaded Oracle Application Server
- Installed an Oracle Application Server instance that includes Oracle HTTP Server and OC4J
- Installed a second application server instance that includes only OC4J to be managed by the first instance in a cluster
- Launched the Application Server Control console and viewed the installed instances

13

Deploying the Application to Oracle Application Server 10*g*

You are now ready to deploy the application to the `default-group` group of OC4J instances within the cluster (if you have not configured a cluster, you can deploy to a single OC4J instance). You use JDeveloper to package the application for deployment, and you use Application Server Control to complete the deployment.

If you did not successfully complete creating all the pages of the application, you may access the solution for the previous section in the `\solutions` subdirectory of your `\setup` directory (see the section "Setting Up Your Environment" in Chapter 1 for instructions on creating a `\setup` directory). Extract the `SRDEMOCh11.zip` file into a directory of your choosing and open the **`SRDEMO.jws`** workspace in JDeveloper to continue working on it.

This chapter contains the following topics:

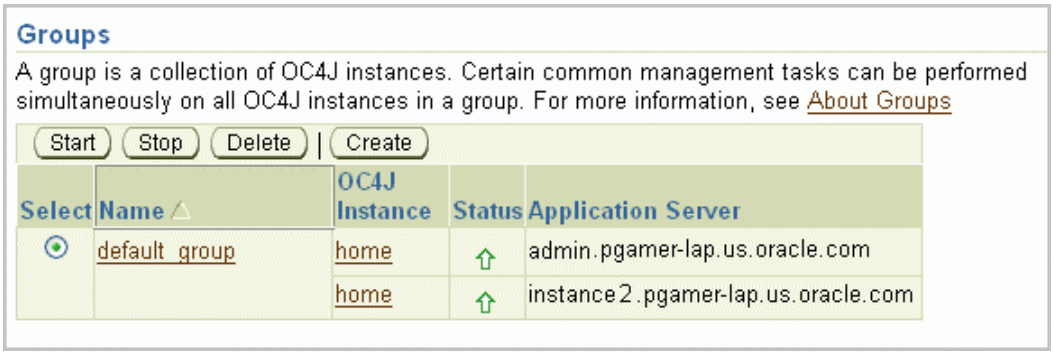
- Deployment Overview
- Making the Application Distributable
- Creating a Deployment Project
- Creating Deployment Profiles
- Creating Deployment Descriptors
- Creating the Data Sources
- Deploying the Application
- Testing the Application
- Using Application Server Control to Explore the Application
- Clustering and Rolling Upgrade (optional)
- Summary

Deployment Overview

Deploying an application essentially consists of the following:

- Uploading the archive
- Binding the Web modules within the application to the Web site within OC4J that will receive requests from Oracle HTTP server.
- Configuring the application by modifying the OC4J-specific deployment descriptors through Application Server Control
- Deploying the archive

In the browser that is displaying Application Server Control, notice the Groups section at the bottom of the page, as shown in the following screenshot:



OC4J instances that share the same name—in this case, `home`—form a "group" of instances. Operations such as start, stop or deploy applications can be performed simultaneously across all of the instances within a group.

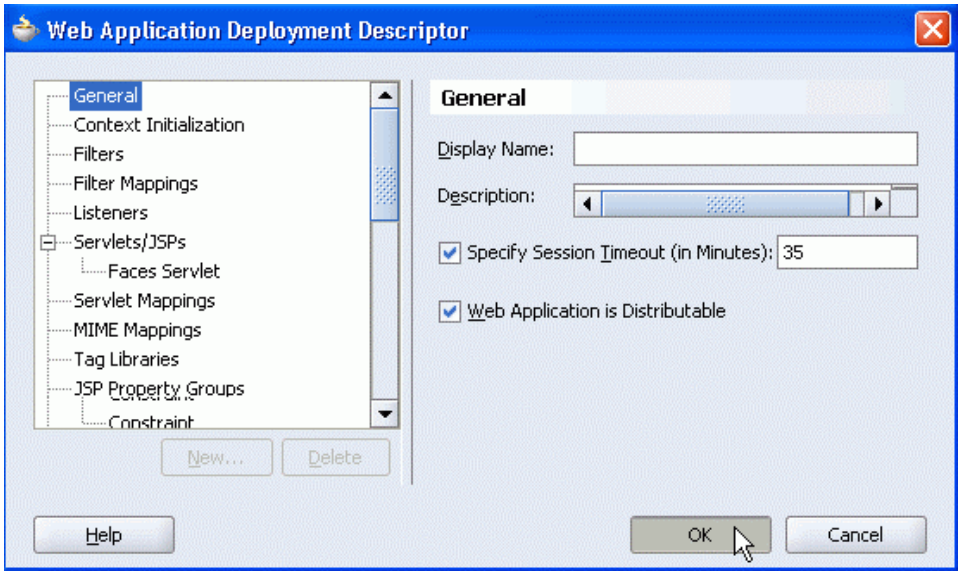
Making the Application Distributable

By default, a Java EE application can run in only a single JVM. However, an application that is marked as distributable can run simultaneously in multiple Web containers. This improves scalability, because it is possible to spread the load across multiple servers. It can also improve availability by providing transparent failover between servlet instances.

To make the application distributable, perform the following steps:

1. With the SRDEMO application open in JDeveloper, expand the **ViewController** project and the **Web Content** and **WEB-INF** nodes.
2. Right-click **web.xml** and select **Properties** from the context menu.
3. In the Web Application Deployment Descriptor dialog box, select **General** in the tree at the left.

4. Select the **Web Application is Distributable** check box, and then click **OK**.



Creating a Deployment Project

Java EE applications are server-side applications that contain standard components such as servlets, JSPs, and EJBs. Applications are deployed in the form of modules that package all necessary code for deployment to a Java EE server such as Oracle Application Server 10*g*. Modules are archive files in a standard format, specified by the Java EE architecture. So the first step in deploying an application is to assemble it into a Java EE application or module.

To keep all the elements of your application organized and cleanly separated, you create a project to hold the deployment components for your application. This project contains the deployment profiles and files.

To create the deployment project, perform the following steps:

1. In the Applications Navigator, right-click the **SRDEMO** application node and select **New Project** from the context menu.
2. Ensure that **Empty Project** is selected in the Items list, and then click **OK**.
3. In the Create Project dialog box, enter a Project Name of **`Deployment`** and click **OK**.

You now have a project to use to manage the deployment for your application.

Creating Deployment Profiles

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 to be included in a deployment package. Creating the deployment profile does not create the deployment package, but merely describes how it is to be created when deployment takes place.

There are three parts of the deployment package for the service request application:

- Model project (`.jar` file)
- ViewController project (`.war` file)
- Deployment project (`.ear` file)

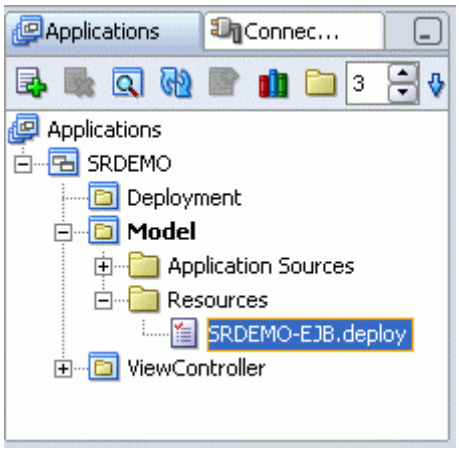
You create deployment profiles for each of the three parts in this section of the tutorial.

Creating the Model Project Deployment Profile

The first deployment profile that you create is for the Model project. The contents of this project are primarily the Java classes that make up the data model portion of the application. The deployment type for this project is an EJB JAR (Java Archive) file.

To create the deployment profile for the Model project, perform the following steps:

1. Right-click the **Model** project in the Applications Navigator and select **New** from the context menu.
2. In the New Gallery, expand **General** and select **Deployment Profiles** from the Categories list. Select **EJB JAR File** in the Items list and click **OK**.
3. In the Create Deployment Profile dialog box, enter **`SRDEMO-EJB`** as the Deployment Profile Name, and then click **OK**.
4. In the EJB JAR Deployment Profile Properties dialog box, click **OK** to accept the default values.
5. Click **Save All** to save your work. The Applications Navigator should now look like the following screenshot:

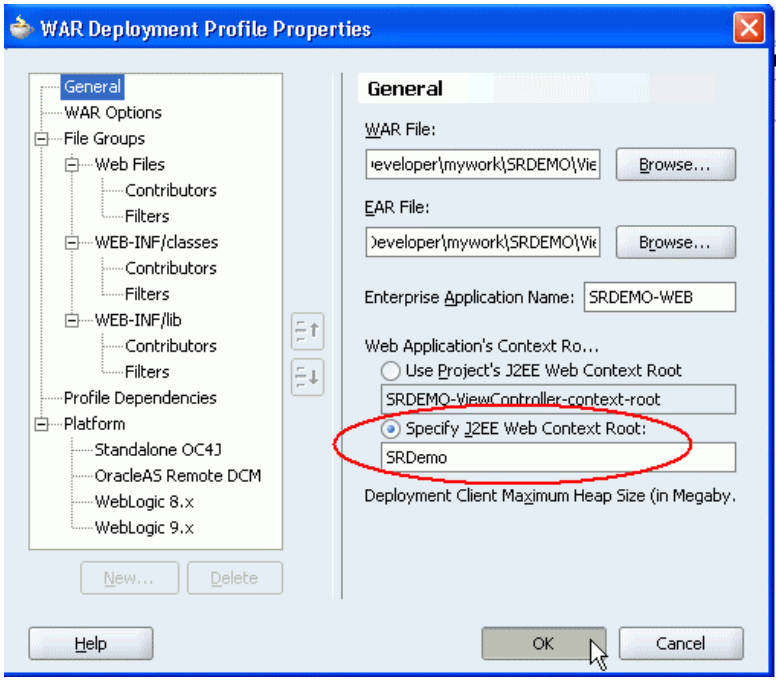


Creating the ViewController Project Deployment Profile

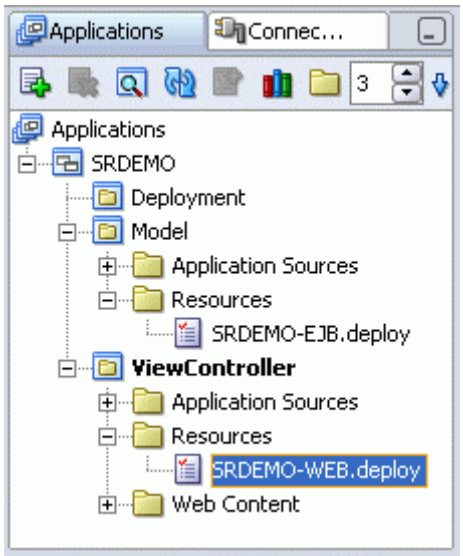
You now create a deployment profile for the ViewController project. This project is where you created the user interface components of the application. The deployment file for this project is a `.war` file (Web Archive) for the Web components.

1. Right-click the **ViewController** project in the Applications Navigator, and select **New** from the context menu.
2. In the New Gallery, expand **General** and select **Deployment Profiles** from the Categories list. Select **WAR File** in the Items list and click **OK**.
3. In the Create Deployment Profile dialog box, enter **`SRDEMO-WEB`** as the Deployment Profile Name, and then click **OK**.
4. In the WAR Deployment Profile Properties dialog box, for the Web Application's Context Root select the **Specify J2EE Web Context Root** option and enter **`SRDemo`** for the value. This becomes part of the URL that is used to access the application.

   Click **OK** to accept the remaining default values in the WAR Deployment Profile Properties dialog box.



5. Click **Save All** to save your work. The Applications Navigator should now look like the following screenshot:

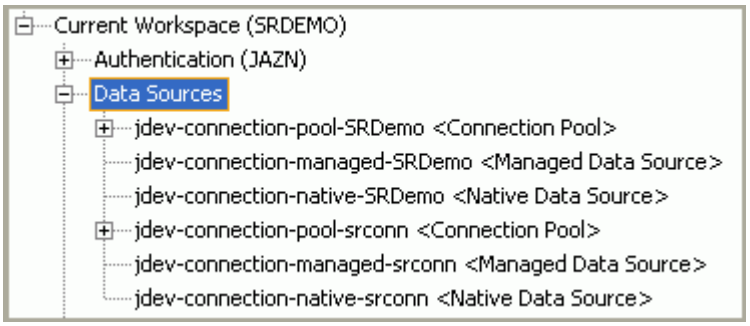


Creating the Deployment Project Deployment Profile

Now that you have created the `.jar` and `.war` files, you can assemble the application into a deployable package.

In the assembly part of deployment, you create a deployment profile that includes any `.jar` and `.war` files you need for your application, along with other server configuration files that may be required. Two configuration files that are required are:

- The `data-sources.xml` file that contains the configuration for the installed data sources of an application server. Data sources are actual Java objects that represent the physical data storage systems, such as databases. From these objects a Java EE application can retrieve JDBC connections to the databases being represented. This layer of abstraction provides benefits such as connection pooling, configuration flexibility and application portability. JDeveloper automatically creates an OC4J data source for each database connection in your JDeveloper installation; you can view these by selecting Tools → Embedded OC4J Server Preferences.



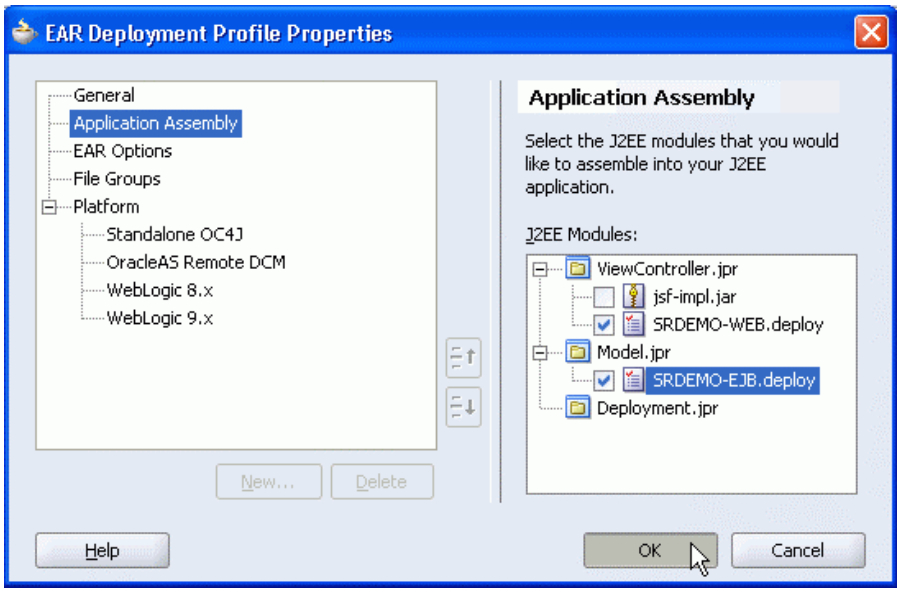
- The `jazn-data.xml` file, previously described in Chapter 6 of this tutorial, that defines the realms (users and roles) for container-managed application security

To create this deployment profile, perform the following steps:

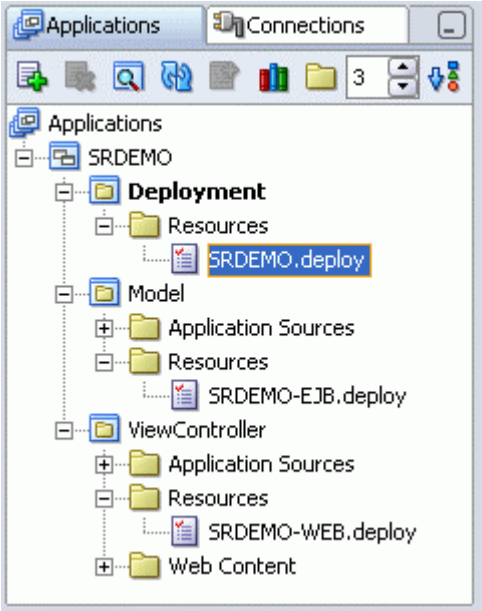
1. In the Applications Navigator, right-click the **Deployment** project and select **New**.
2. In the New Gallery, expand **General** and select **Deployment Profiles** from the Categories list. Select **EAR File** in the Items list and click **OK**.
3. Change the profile name to **SRDEMO** and click **OK**.
4. In the EAR Deployment Profile Properties dialog box, select the **Application Assembly** category, and then select the following Java EE modules to include in the `.ear` file:

   ViewController.jpr → SRDEMO-WEB.deploy
   Model.jpr → SRDEMO-EJB.deploy

   Click **OK** to accept the other defaults.



5. Click **Save All** to save your work. The Applications Navigator should now look like the following screenshot:



Creating Deployment Descriptors

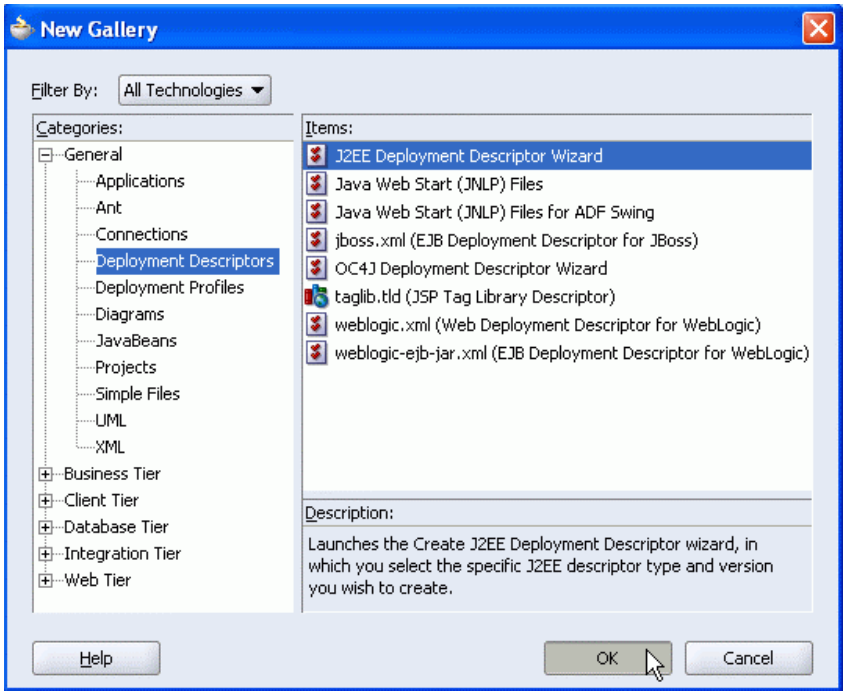
Until now you have been running the application within JDeveloper's embedded OC4J container. When you deploy to a Java EE platform like Oracle Application Server 10*g*, there are very specific requirements for names and locations of files that are to be deployed. In this section you create generic templates for deployment descriptors that meet these requirements. Then you add application-specific information to these templates.

Deployment descriptors are configuration files that are associated with projects and deployed with Java EE applications and modules. Deployment descriptors contain the declarative data required to deploy the components as well as the assembly instructions that describe how the components are composed into an application.

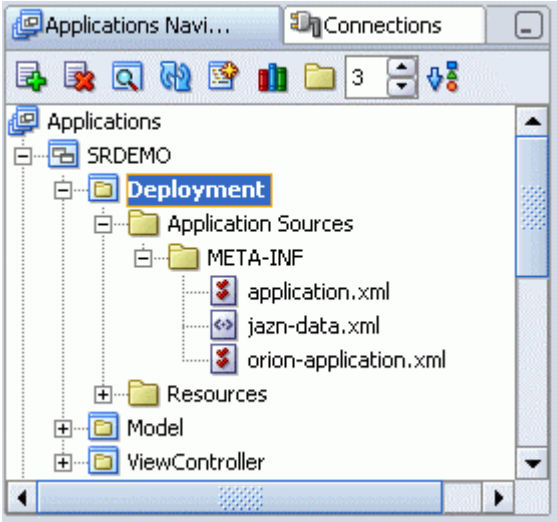
Creating the Deployment Descriptor Templates

To create templates for the deployment descriptors that are needed for this application, perform the following steps:

1. Right-click the **Deployment** project and select **New** from the context menu.
2. In the New Gallery, select **Deployment Descriptors** from the Categories list. In the Items list, select **J2EE Deployment Descriptor Wizard** and click **OK**.



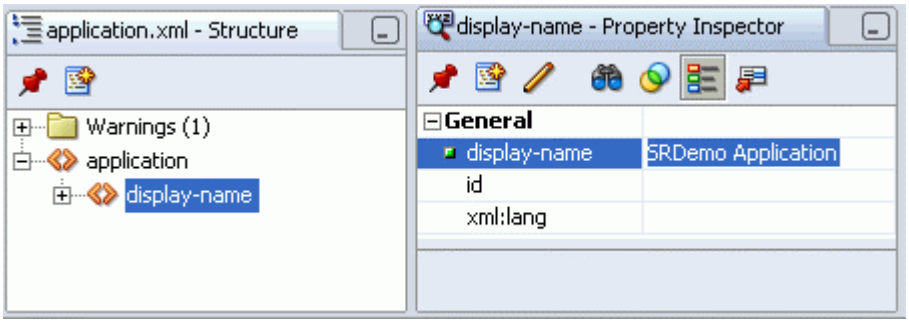
3. If the Welcome page of the Create Deployment Descriptor wizard displays, click **Next**.
4. On the Select Descriptor page of the wizard, select **application.xml** and click **Next**.
5. On the Select Version page of the wizard, select **5.0** and click **Finish**.
6. Right-click the **Deployment** project and select **New** from the context menu.
7. In the New Gallery, select **Deployment Descriptors** from the Categories list. In the Items list, select **OC4J Deployment Descriptor Wizard** and click **OK**.
8. If the Welcome page of the Create Deployment Descriptor wizard displays, click **Next**.
9. On the Select Descriptor page of the wizard, select **jazn-data.xml** and click **Next**.
10. On the Select Version page of the wizard, select **10.0** and click **Finish**.
11. Invoke the OC4J Deployment Descriptor wizard once again, this time selecting the **orion-application.xml** deployment descriptor and version **10.0**. When you are finished, the Applications Navigator should look like the following screenshot:



Modifying the Deployment Descriptor Templates

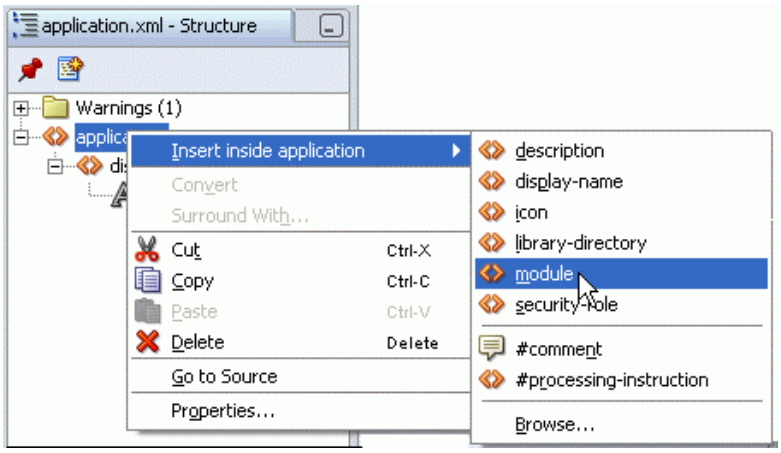
Now that you have created the templates for the deployment descriptors, you can add application-specific information to each. To modify the deployment descriptors, perform the following steps:

1. If **application.xml** is open in the editor, switch to it by clicking its tab; otherwise double-click it in the Applications Navigator to open it in the editor.
2. In the Structure window, select the node **display-name**, and then in the Property Inspector change the **display-name** to `SRDemo Application`.

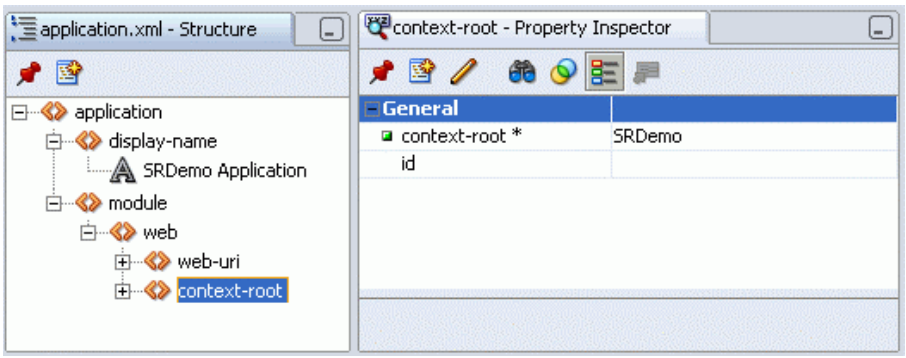


Next you insert two modules into the application: the Web module and the EJB module.

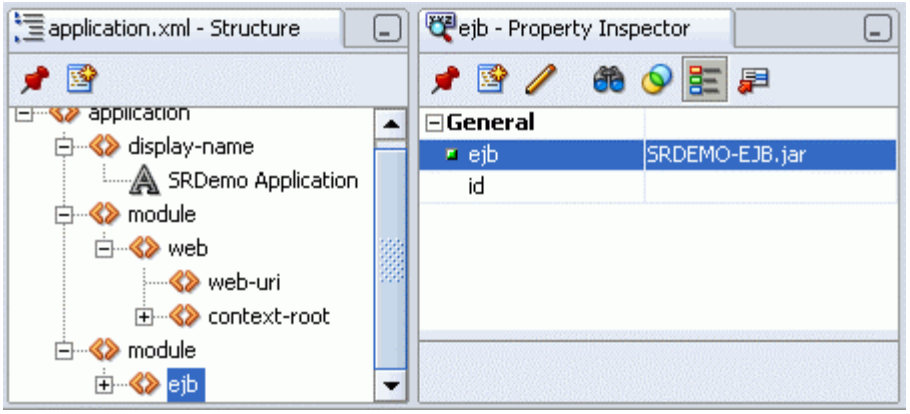
3. To insert the Web module, perform the following steps:
   - In the Structure window, right-click the **application** node and select **Insert inside application → module** from the context menu.



- Right-click the **module** node and select **Insert inside module → web** from the context menu and enter an id of **`srdemo`**.
- Right-click the **web** node and select **Insert inside web → web-uri**, and then in the Property Inspector change the **web-uri** to **`SRDEMO-WEB.war`**.
- Again right-click the **web** node and select **Insert inside web → context-root**, and then in the Property Inspector enter **`SRDemo`** as the value for **context-root**, as shown in the following screenshot:



4. Now you insert the EJB module by performing the following steps:
   - In the Structure window, right-click the **application** node and select **Insert inside application → module** from the context menu.
   - Right-click the **module** node and select **Insert inside module → ejb** from the context menu.
   - In the Property Inspector, enter **`SRDEMO-EJB.jar`** as the value for the **ejb** property.



5. The `orion-application.xml` file should point to the location of the `jazn-data.xml` file and specify the data source to use. Switch to **orion-application.xml** in the editor by clicking its tab, or double-click it in the Applications Navigator to open it in the editor.

6. To make the file easier to read, right-click in the editor and select **Reformat** from the context menu.

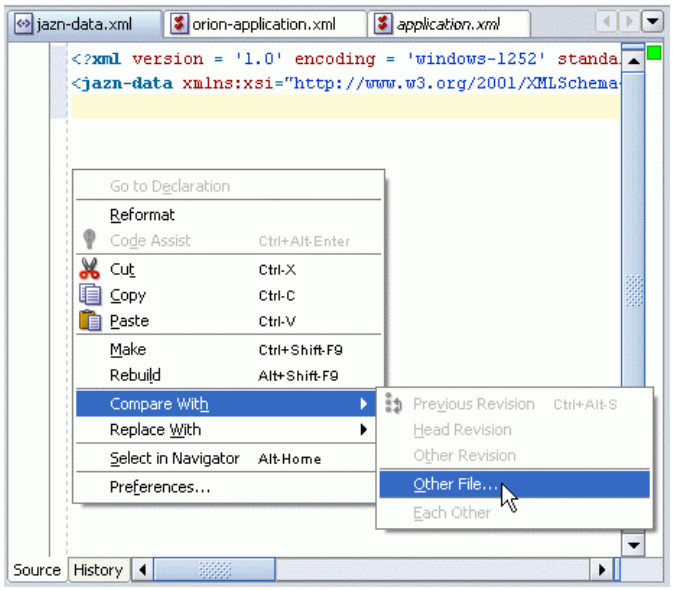
7. Add the following attribute to the opening `orion-application` tag:
   `default-data-source="jdbc/SRDemoDS"`

   The entire file should now read (with the added portion in bold):
   ```
   <?xml version = '1.0' encoding = 'windows-1252'?>
   <orion-application xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
        xsi:noNamespaceSchemaLocation=
         "http://xmlns.oracle.com/oracleas/schema/orion-application-10_0.xsd"
        default-data-source="jdbc/SRDemoDS">
      <jazn location="./jazn-data.xml" provider="XML"/>
   </orion-application>
   ```

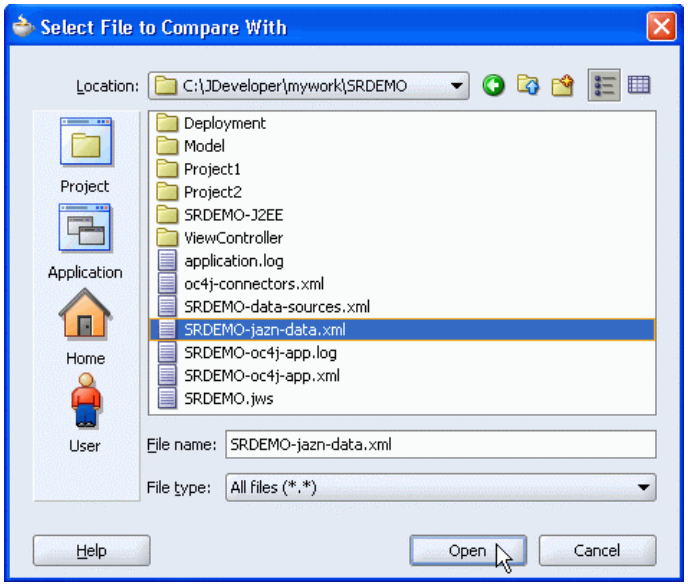
   Ensure that the right margin does not contain any red areas indicating any errors in the XML.

8. Switch to **jazn-data.xml** in the editor by clicking its tab, or double-click it in the Applications Navigator to open it in the editor.

9. When you were creating the Service Request application, you created users and roles in a file named `SRDemo-jazn-data.xml`. Because you have already defined this for the embedded OC4J data, you can copy the definitions into the Java EE deployment descriptor. Right-click in the editor and select **Compare With → Other File** from the context menu.



10. In the Select File to Compare With dialog box, navigate to the main application directory **SRDEMO**, select **SRDEMO-jazn-data.xml**, and click **Open**.

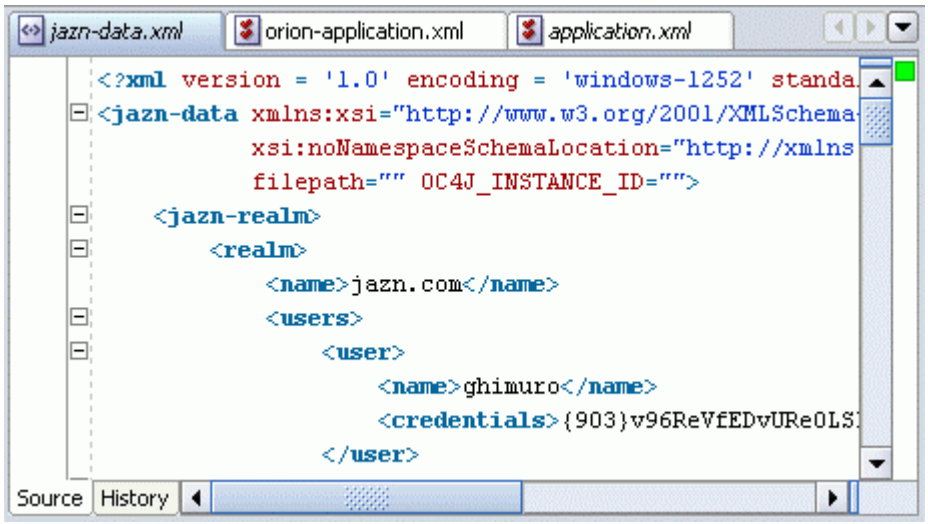


11. In the Compare files, select in the view of `SRDEMO-jazn-data.xml` all the text beginning with the line **`<jazn-realm>`** and press **[Ctrl]+[C]** to copy the text. You can then close the Compare if you wish.

12. In **jazn-data.xml**, delete the closing **`</jazn-data>`** tag. If there is not a complete closing **`</jazn-data>`** tag, then just delete the forward slash (**/**) prior to the closing bracket. The file should now look like this:

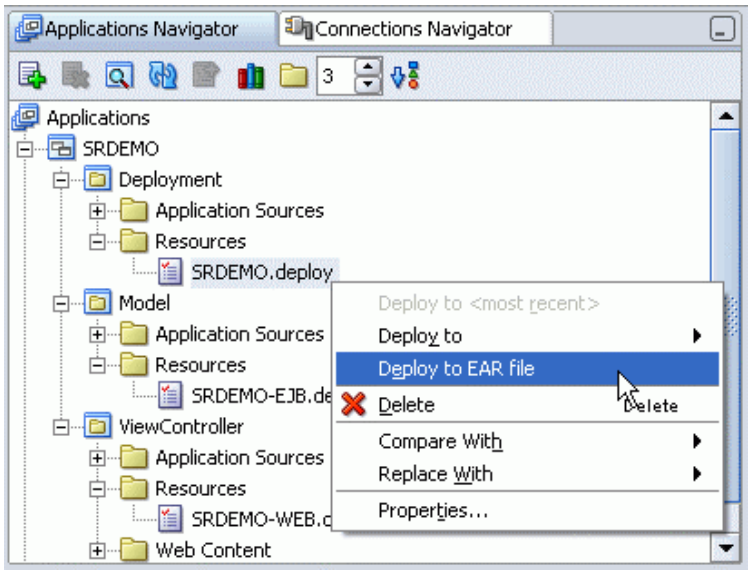
```
<?xml version = '1.0' encoding = 'windows-1252' standalone =
'yes'?>
<jazn-data xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=http://xmlns.oracle.com/oracleas/
schema/jazn-data-10_0.xsd
filepath="" OC4J_INSTANCE_ID="">
```

13. Paste the copied text on the next line.
14. Right-click in the editor and select **Reformat** from the context menu. The file should look similar to the following screenshot:



Ensure that the right margin does not contain any red areas indicating any errors in the XML.

15. Click **Save All** to save your work.
16. Now that you have all of the necessary files for deployment, you can create the EAR file that you deploy in the next section to Oracle Application Server 10*g*. In the Applications Navigator, right-click **SRDEMO.deploy** in the Deployment project and select **Deploy to EAR file** from the context menu.



The Deployment Log window shows successful deployment.

```
Deployment - Log
---- Deployment started. ----    Aug 9, 2006 1:49:55 PM
Target platform is Standard J2EE.
Wrote WAR file to D:\JDev10131\SRDemoWork\SRDEMO\ViewController\dep
loy\SRDEMO-WEB.war
Wrote EJB JAR file to D:\JDev10131\SRDemoWork\SRDEMO\Model\deploy\S
RDEMO-EJB.jar
Wrote EAR file to D:\JDev10131\SRDemoWork\SRDEMO\Deployment\deploy\
SRDEMO.ear
Elapsed time for deployment:  50 seconds
---- Deployment finished. ----    Aug 9, 2006 1:50:41 PM

Messages   Deployment
```

You have now created the EAR file that you need to successfully deploy your application to a Java EE server. Make note of the location of the EAR file that is shown in the log window.

Creating the Data Sources

Applications deployed to Oracle Application Server 10*g* use a data source and connection pool to manage database access. When you are developing the application, JDeveloper creates a `data-sources.xml` file for you. It uses that file during development and internal testing.

Although JDeveloper can create a data source and package it for deployment, it is a better practice to create a data source directly on the application server. You can give the data source a logical name that you associate with a physical connection at the time of deployment. If the database connection is not available during deployment, the deployment fails. To change to a different database, you have only to modify the data source of the deployed application to connect to a different database. This ensures that changes to the database do not require repackaging and redeployment of the application. You do have to restart OC4J after making changes to the `data-sources.xml` file.

To facilitate tasks such as creating data sources and deploying applications within a cluster, you can now perform such operations simultaneously across a *group* of OC4J instances — a set of instances that belong to the same cluster. As noted in the previous chapter, because the two OC4J instances you created are members of the same cluster, they are added by default to the `default_group` of instances. You create a managed data source, an OC4J-specific implementation that acts as a wrapper to a JDBC driver or data source. But first, you have to create a connection pool, which is used by managed data sources to efficiently manage connections.

As part of this exercise, you create the necessary connection pool and data source simultaneously on the `home` instance group. You then deploy the application to the same group.

Creating the Connection Pool

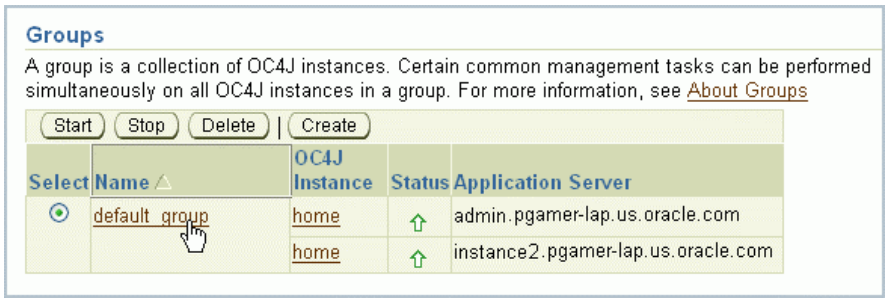
To create the connection pool, perform the following steps:

1. Launch Application Server Control as you did previously by entering the following URL in your Web browser: `http://<hostname>:<port>/em`.

   For example: `http://myhost.oracle.com:7777/em/`

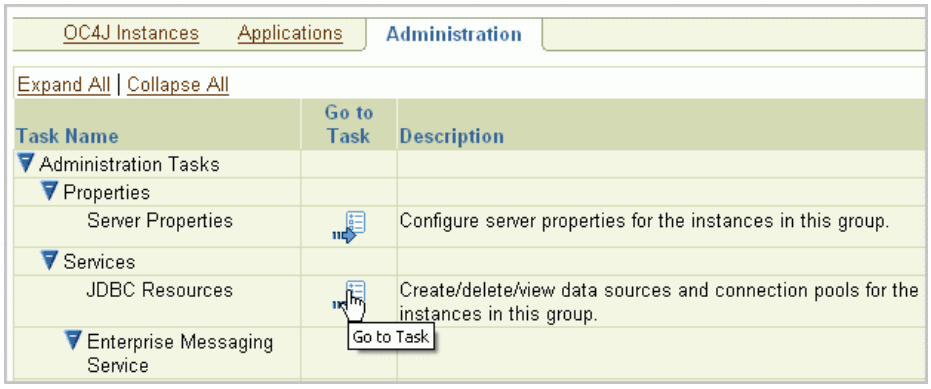
2. Log in to Application Server Control by providing the password that you specified when installing the Application Server (**`welcome1`** in the example), and then click **Login**.

3. On the Cluster Topology page, click the **default_group** link under the **Groups** section at the lower part of the page. You have now selected the `home` group of OC4J instances.

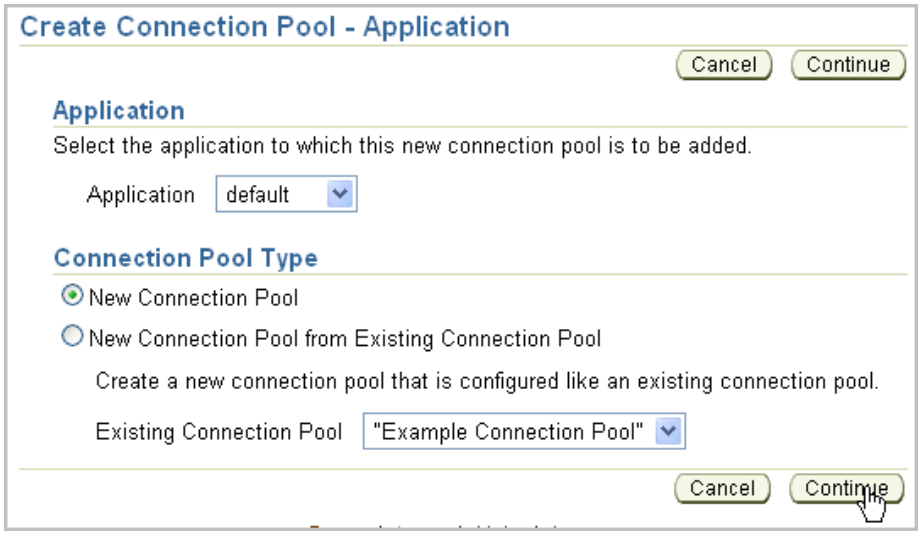


4. On the "Group: default_group" page, click the **Administration** tab.

5. On the Group: default_group Administration page, click **Go to Task** for the **JDBC Resources** task.



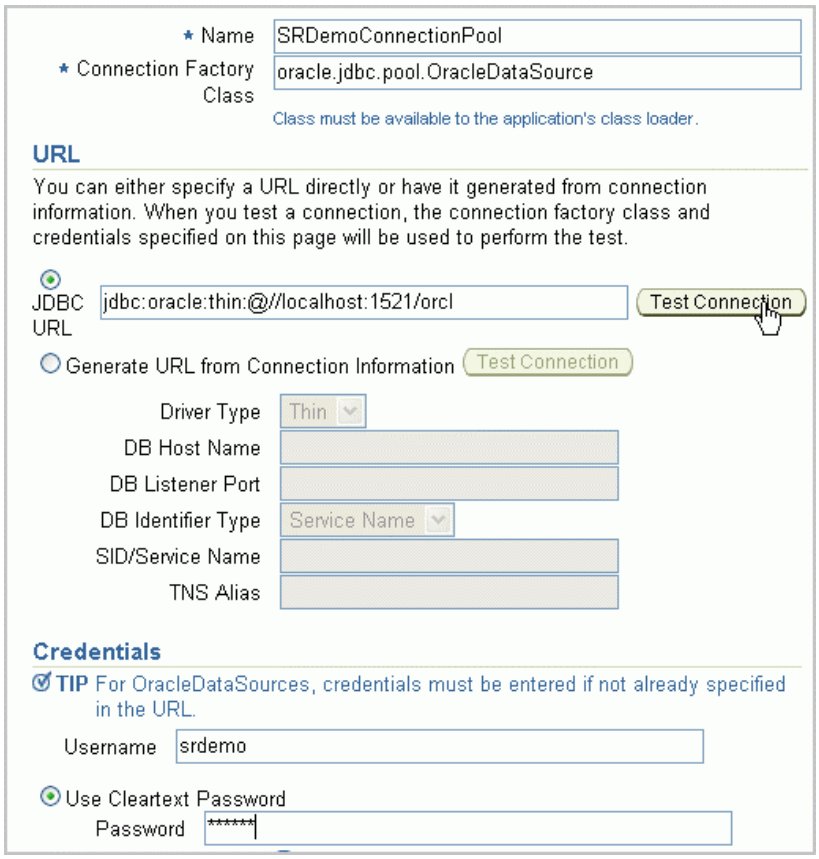
6. On the JDBC Resources page, click **Create** under **Connection Pools**.
7. On the Create Connection Pool – Application page, select the **New Connection Pool** option and click **Continue**.



8. Use the following information to create a connection pool:

Field	Value
Name	`SRDemoConnectionPool`
Connection Factory	`oracle.jdbc.pool.OracleDataSource` (the default value)
JDBC URL	`jdbc:oracle:thin:@//localhost:1521/orcl` (your host, port, and database)
Credentials: Username	`srdemo`
Credentials: Cleartext Password	`oracle`

9. Test the connection by clicking **Test Connection**.



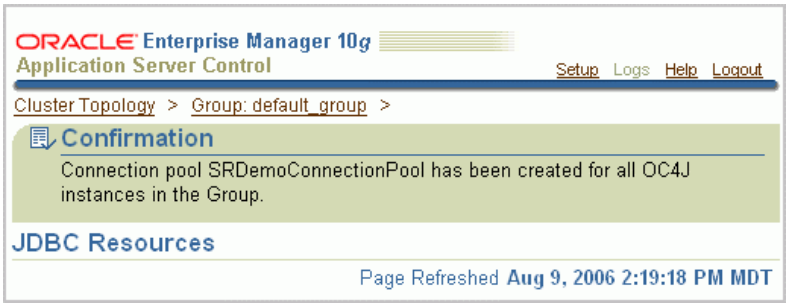
10. In the Test Connection window, click **Test**. If successful, you return to the Create Connection Pool page, where you should see this message:

    `Connection established successfully for all OC4J instances in the Group.`

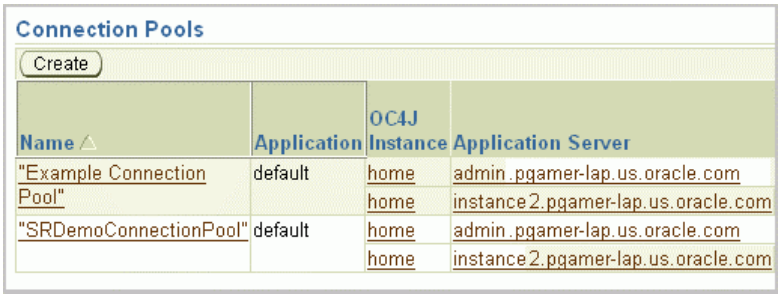
    Click **Finish**.

11. The JDBC Resources page shows a confirmation message:

    `Connection pool SRDemoConnectionPool has been created for all OC4J instances in the Group.`



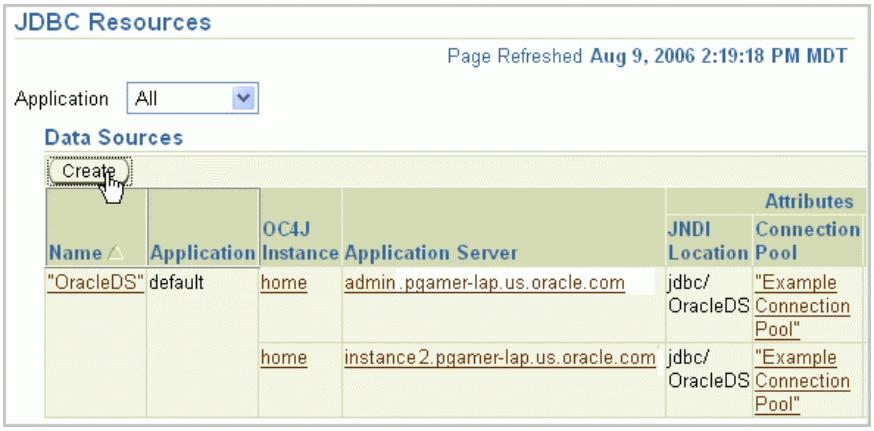
The new connection pool is shown under the Connection Pools section.



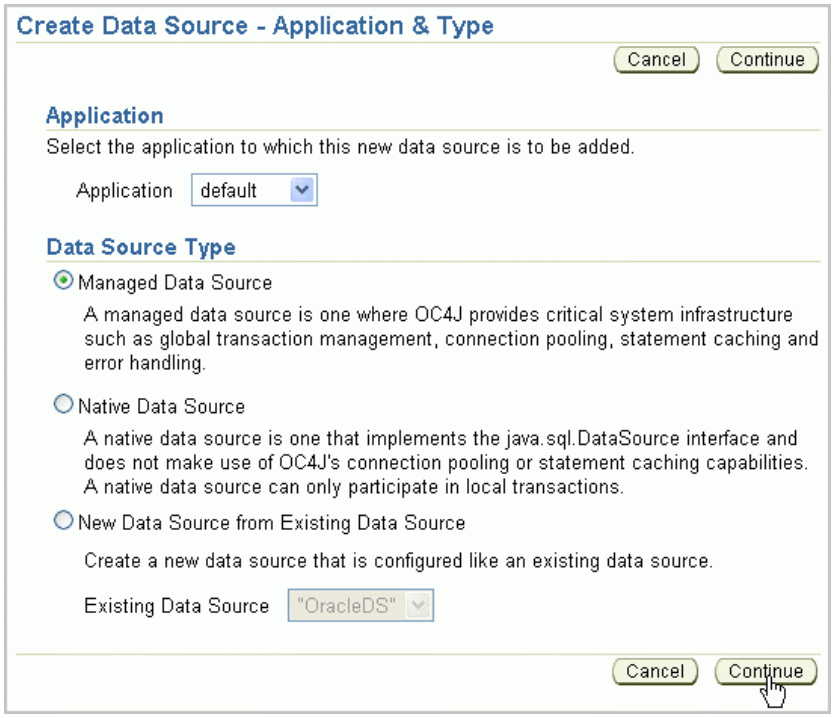
Creating the Data Source

Now that the connection pool is created, you can perform the following steps to create the data source that actually provides the connection to the database:

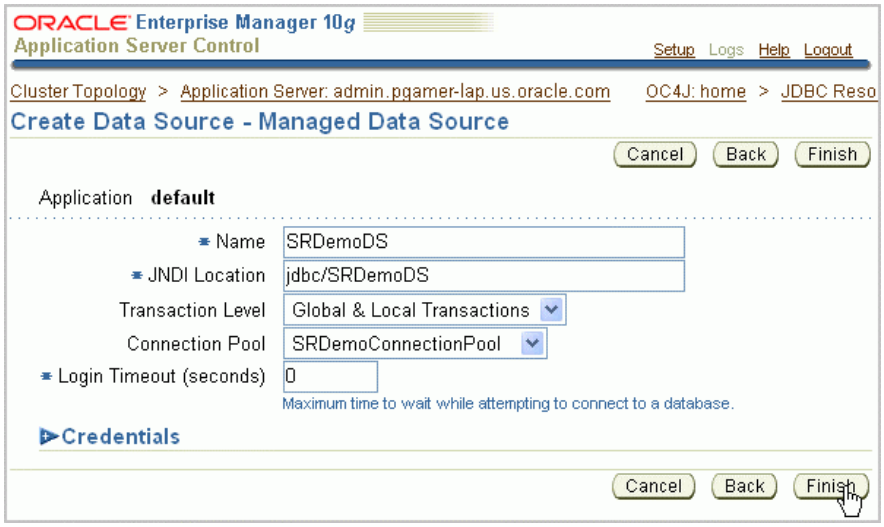
1. Click **Create** in the Data Sources table.



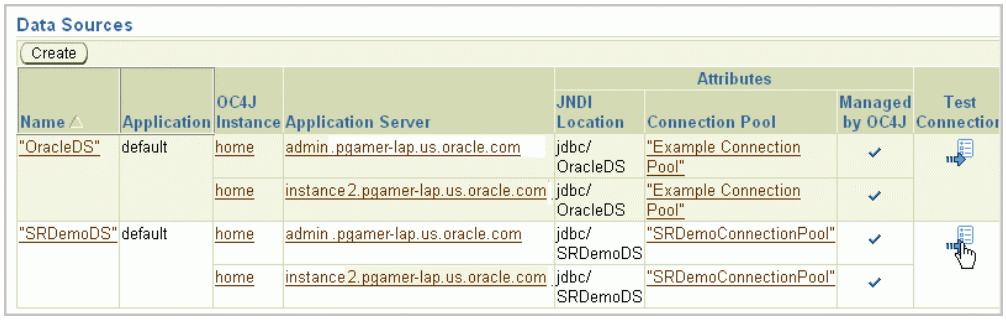
2. On the “Create Data Source – Application & Type” page, in the Application section of the page select **default** from the Application drop-down list. The data source should be available to the default application and all of its child applications, including SRDEMO. Select the **Managed Data Source** option and click **Continue**.



3. On the "Create Data Source – Managed Data Source" page, set Name to **`SRDemoDS`**. The data source must be named the same as the data source that you used during development. Set JNDI Location to **`jdbc/SRDemoDS`**. Select **SRDemoConnectionPool** from the Connection Pool drop-down list. You do not need to specify credentials because those were set in the connection pool definition. Click **Finish**.

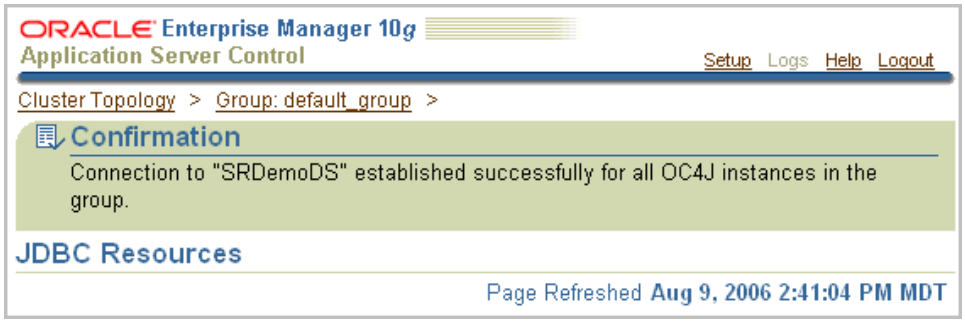


4. On the JDBC Resources page, click **Test Connection** to test the SRDemoDS data source.



5. On the TestConnection: “SRDemoDS:” page, click **Test**.
6. On the JDBC Resources page, if the test is successful you will see this message:

   ```
   Connection to "SRDemoDS" established successfully for all OC4J
   instances in the Group.
   ```



You are now ready to deploy the application.

Deploying the Application

JDeveloper provides a one-click option to deploy an application to an application server. by right-clicking the deployment profile and selecting the target application server.

An alternative method of deployment is to use Application Server Control. This approach cleanly splits the role of the application developer from that of the application deployer, and also enables the application to be deployed to multiple OC4J instances in a cluster. This is the approach taken in this tutorial to deploy the application to the `default_group` of OC4J instances within the cluster.

Deploying an application essentially consists of the following tasks:

- Upload the archive
- Bind the Web modules within the application to the Web site within OC4J that will receive requests from Oracle HTTP server
- Configure the application by modifying the OC4J-specific deployment descriptors through Application Server Control
- Deploy the archive

A key benefit of performing configuration tasks at deployment is that you can save them to a reusable *deployment plan* that you can apply when you redeploy the application. A deployment plan is a client-side aggregation of all the configuration data needed to deploy an archive into OC4J. Any data that you set during installation is persisted to the deployment plan, in addition to the data specified in any OC4J deployment descriptors packaged with the EAR.

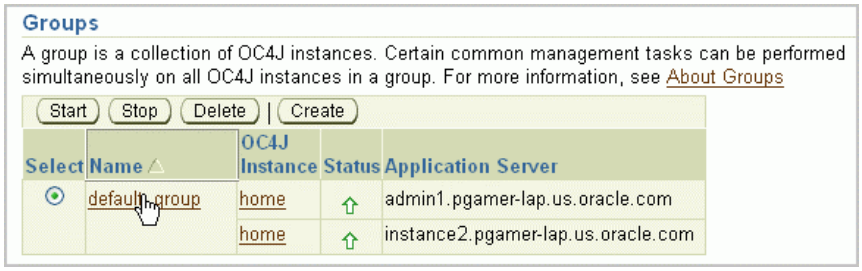
All of the data specified in a deployment plan can be edited using the *deployment plan editor* component of Application Server Control, which you use in this tutorial, or with the Configure Application dialog in

JDeveloper that is invoked when you deploy to an application server from JDeveloper. This means that you can edit the packaged descriptors prior to deployment, and then when you deploy the application, new OC4J descriptors are generated that use this data.

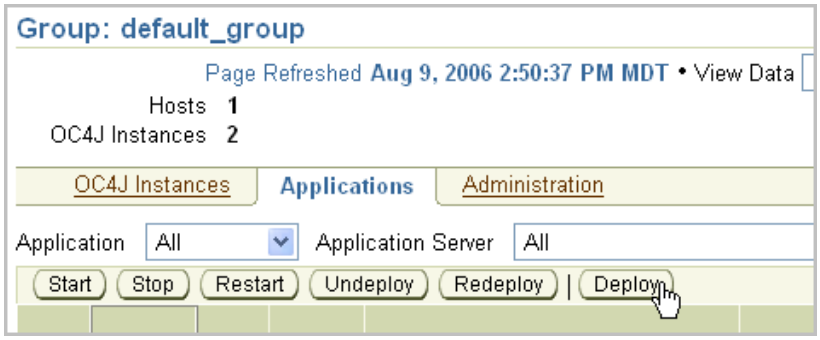
Notice the Groups section at the bottom of the Cluster Topology page in Application Server Control (if you are on a different page, you can navigate to the Cluster Topology page by clicking its link in the breadcrumbs at the top of the page). OC4J instances that share the same name—in this case, `home`—form a group of instances. Operations such as starting, stopping or deploying applications can be performed simultaneously across all of the instances within a group.

To deploy the application, perform the following steps:

1. In the Cluster Topology page (you can return to this by clicking its link in the breadcrumbs at the top of the page), click the **default_group** link under Groups.



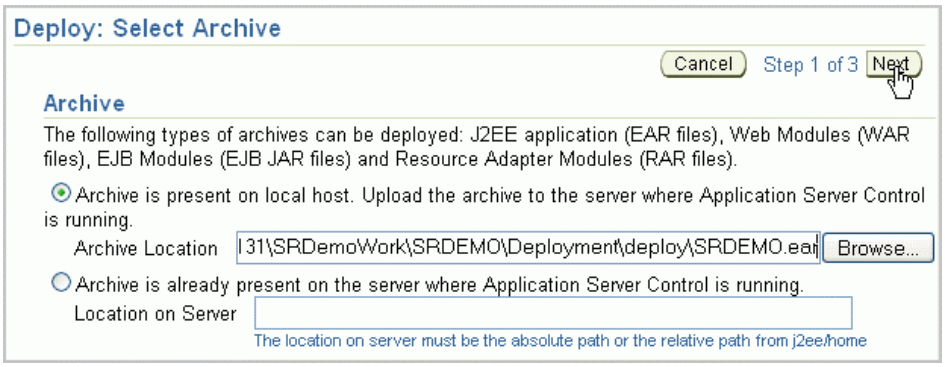
2. On the "Group: default_group" page, click the **Applications** tab and then click **Deploy**.



3. On the Select Archive page of the Deployment wizard, click **Browse** in the Archive section to navigate to the location of your EAR file, which is in the `\Deployment\deploy` subdirectory of your SRDEMO application.

   For example: `C:\JDeveloper\jdev\mywork\SRDEMO\Deployment\deploy`

   Select the **SRDEMO.ear** file and click **Open**. On the Select Archive page of the wizard, click **Next**.

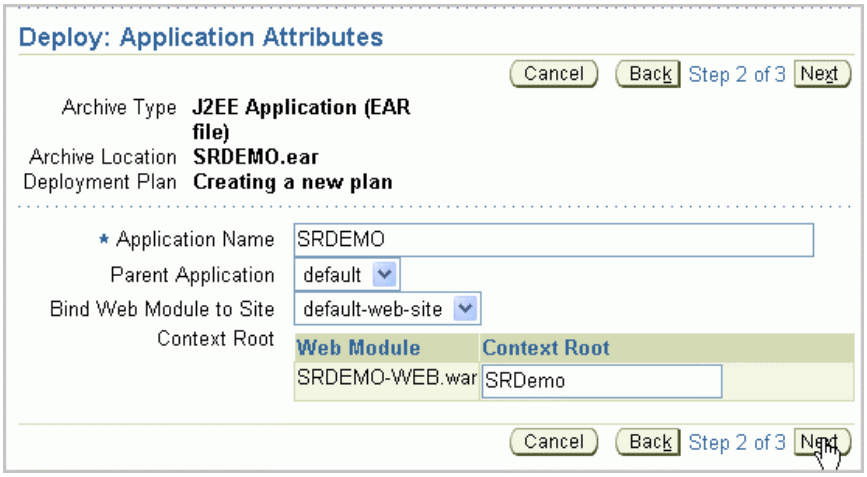


4. On the Application Attributes page of the deployment wizard, enter the Application Name of **`SRDEMO`**. This name identifies this application in the list of applications on the OC4J Applications page.

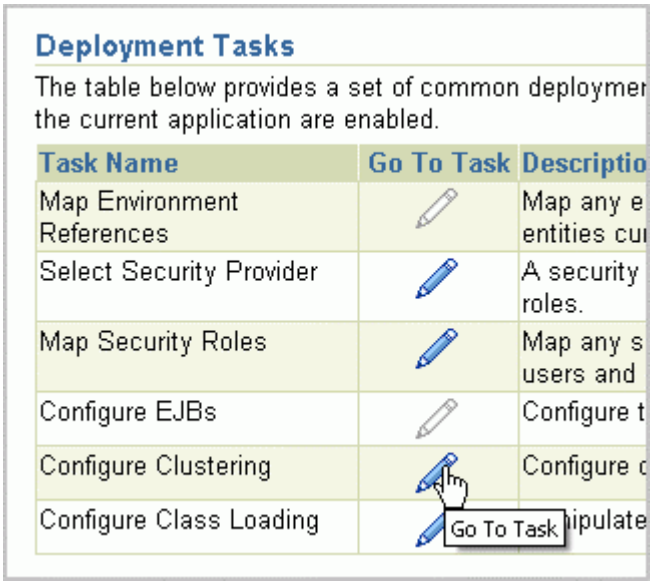
   Verify that the context root is set for each Web module in the application (in this case, there is only one Web module). This value is read from the Java EE standard `application.xml` deployment descriptor. Users access the Web module by appending this value to the URL. Note that you bind the Web module to `default-web-site`, which defines the internal Web listener that receives requests from OHS.

   It is possible to create additional Web sites within an OC4J instance for use by applications. For example, you may want to create a Web site to be used exclusively by Application Server Control, effectively dividing management and general application Web access. Or you may need to create a secure Web site for receiving HTTPS requests. For purposes of this tutorial, however, the `default` Web site is sufficient.

   Click **Next**.



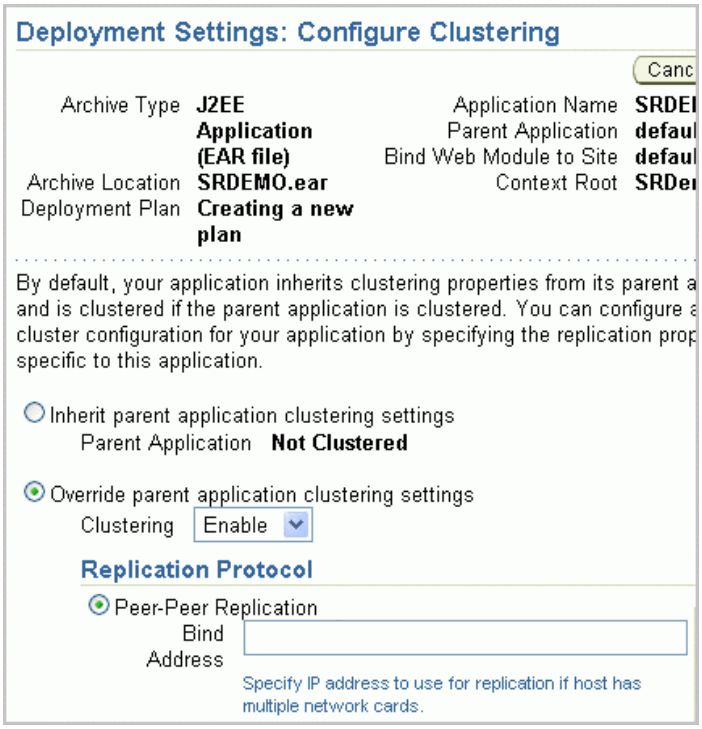
5. Now you are ready to perform install-time configuration. On the Deployment Settings page of the wizard, click **Go To Task** next to the **Configure Clustering** deployment task.



6. On the Configure Clustering page, select the **Override parent application clustering settings** option and select **Enable** from the Clustering drop-down list to enable clustering.

   You can now see the various clustering configuration options.

   Select the **Peer-to-Peer Replication** option to enable dynamic peer-to-peer clustering. You do not need to specify a bind address.

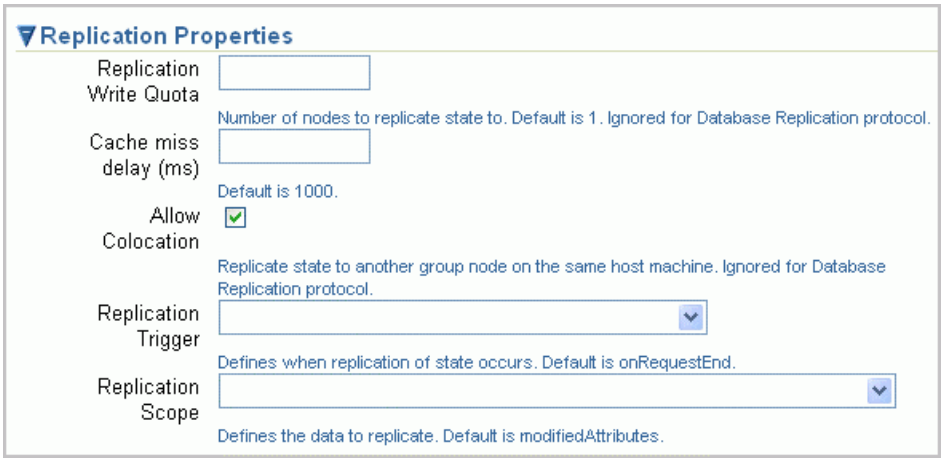


7. At the bottom of the Configure Clustering page, expand **Replication Properties**. You can use these properties to define such parameters as what data is replicated and when replication occurs.

Ensure that the **Allow Colocation** check box is selected, which enables data to be replicated to OC4J instances running on the same physical machine.

Do not change the Replication Trigger setting; leave it blank to use the default value of onRequestEnd.

You don't need to set Replication Scope, since the default configuration will cause all modified attributes on the session to be replicated.



8. Scroll to the bottom or the top of the page and click **OK**. The Deployment Settings page displays the message:

   ```
   Deployment plan has been updated successfully.
   ```

   Click **Deploy**. A Processing page displays progress messages during deployment.

Here's what happens:

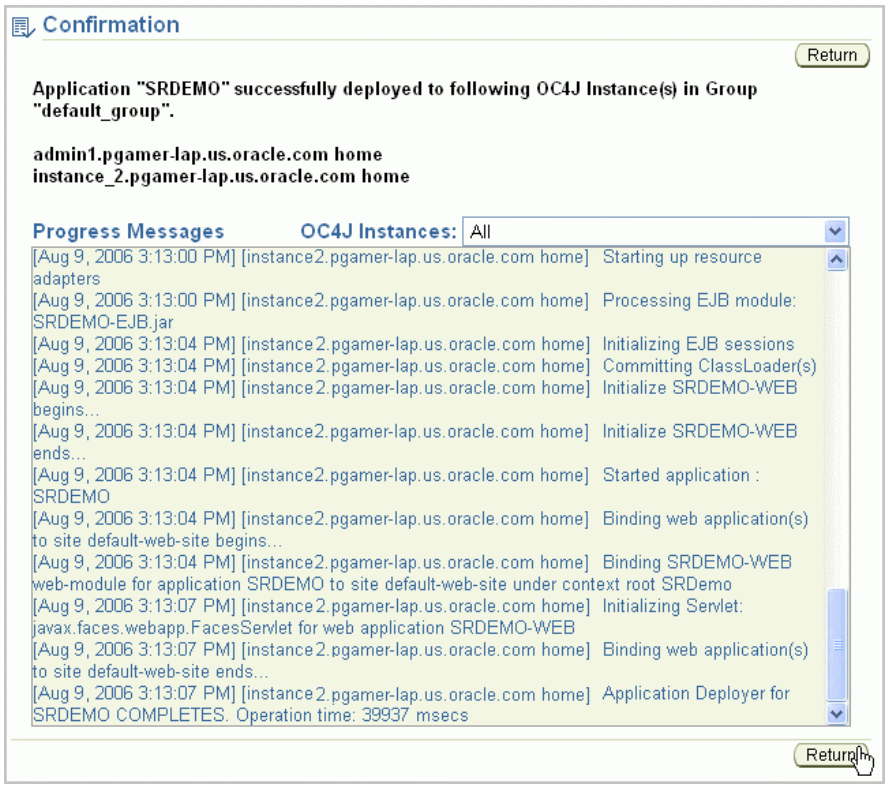
- OC4J copies the EAR file to the master deployment directory on the OC4J instance running in the administration (`admin`) Oracle Application Server instance. The default location is the `<OC4J_HOME>/j2ee/home/applications/` directory.
- OC4J opens and parses the Java EE standard `application.xml` descriptor packaged within the EAR file. This file lists all of the modules contained within the EAR file. OC4J notes these modules and initializes the EAR environment.
- OC4J reads the module deployment descriptors for each module type — Web module (WAR), EJB module, connector module, or client module — into memory. The JAR and WAR file environments are also initialized.
- The archive and the deployment plan are sent to the OC4J server. OC4J uses the contents of the deployment plan to generate the various OC4J-specific descriptors within the `<OC4J_HOME>/j2ee/home/application-deployments` directory.
- OC4J reacts to the configuration details contained in both the Java EE deployment descriptors and any OC4J-specific deployment descriptors. OC4J notes any Java EE component configurations that require action by OC4J, such as wrapping EJBs with their interfaces.
- OC4J writes out new OC4J-specific configuration files to the `<OC4J_HOME>/j2ee/home/application-deployments/<app_name>` directory, according to the contents of the deployment plan. Note that if one or more OC4J-specific deployment descriptors were supplied, you may notice that OC4J added additional elements to the generated files.

- Any generated classes, such as EJB interface wrapper classes, are compiled and put into new subdirectories of this directory. For example, EJB wrapper classes are generated within an archive named deployment-cache.jar within the `<OC4J_HOME>/j2ee/home/application-deployments/<app_name>/<jar_name.jar>/` directory, where `<jar_name.jar>` corresponds to the name of a deployed EJB JAR.
- The same process is then repeated on the second OC4J instance (`instance2` in the example).

You can watch this progress in the Application Server Control Console. Note that deployment continues even if you close your browser.

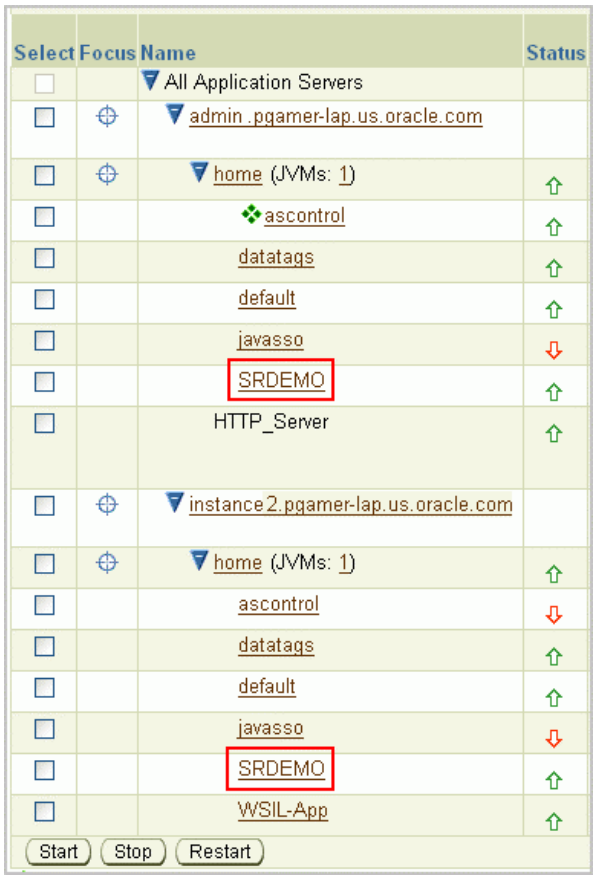
After successful deployment, a confirmation page displays the message:

```
Application "SRDEMO" successfully deployed to all applicable OC4J
Instances in Group "default_group".
```



9. After deployment, click **Return**.

10. Click the **Cluster Topology** link in the breadcrumbs to return to the Cluster Topology page. You should now be able to drill down on each OC4J instance and see that the application has been deployed and started on each.



Testing the Application

You have now deployed the application to Oracle Application Server 10*g*. You can test the application with a Web browser using the context root that you specified for the application.

1. Open a Web browser and enter the URL `http://<hostname>:<port>/SRDemo`

   For example: **`http://localhost:80/SRDemo`**

2. This directs you to the logon page of the application. Use the following as login credentials:

Field	Value
Username	`sking`
Password	`welcome`

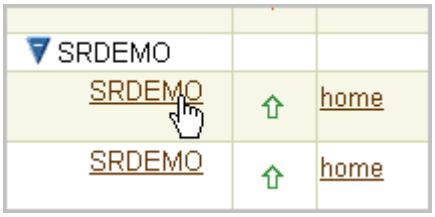
3. Click the links on the Welcome page to explore the pages of the application. *Leave the browser open* when you finish exploring the application.

Using Application Server Control to Explore the Application

You can use Application Server Control to monitor and manage deployed applications. In this section, you explore your application and the application server with Application Server Control by performing the following steps:

1. Open a second Web browser.
2. Enter the URL `http://<hostname>:<port>/em`.

   For example: **`http://localhost:7777/em`**
3. Log in to Application Server Control with the password you used when you installed the Application Server. The example uses the password **`welcome1`**.
4. On the Cluster Topology page, select **Applications** from the View By drop-down menu.
5. Expand the **SRDEMO** application and click the link to one of the deployed SRDEMO applications.



6. You can use Enterprise Manager to explore a number of aspects of the application, such as the following:

   Click the **Home** tab to view specifics about the application you just deployed.

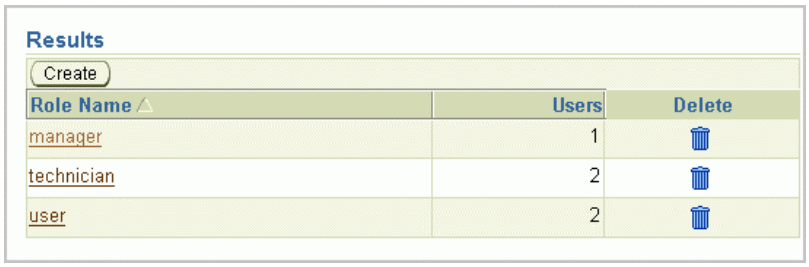
7. Click the **Performance** link to see graphs of application performance.

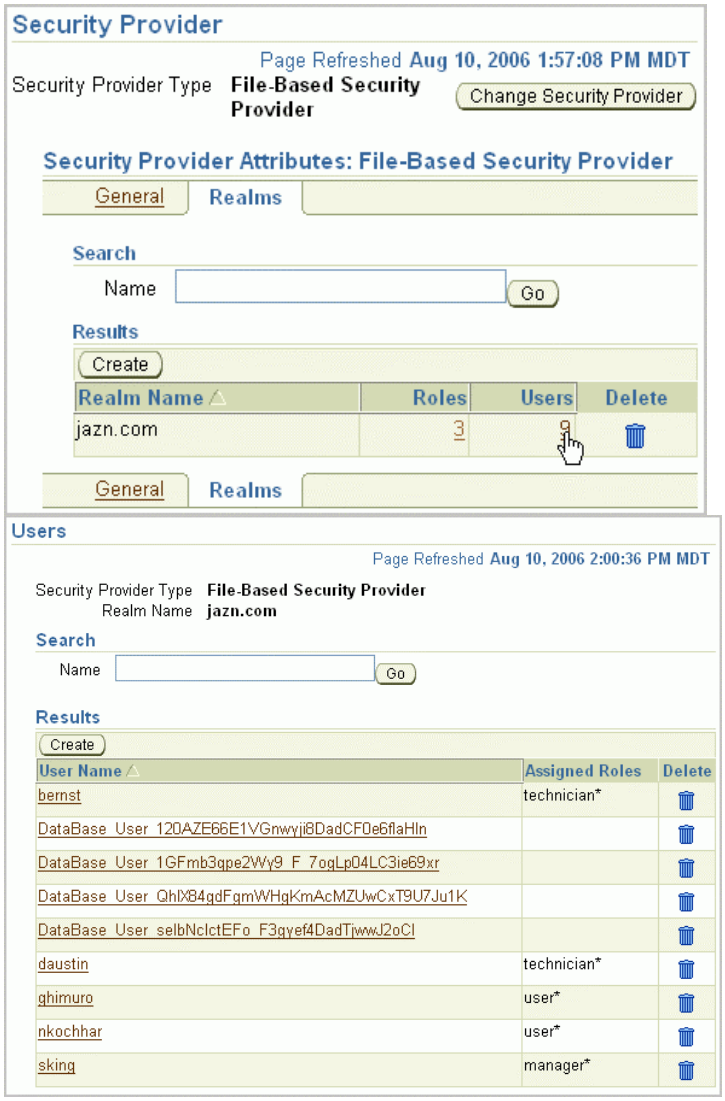
8. Click the **Administration** tab to see all of the deployment aspects of the application, including details of the JAZN security implementation.

9. Click **Go to Task** next to **Security Provider**.

10. On the Security Provider page, the General tab displays the path to the `jazn-data.xml` file and the default realm. Click the **Realms** tab.

11. On the Realms page, click the links under **Roles** and **Users** to see the roles and users defined for the application. You could modify these roles here, although you do not do so for this tutorial.





Clustering and Rolling Upgrade

This is an optional exercise. The goal is to illustrate the state replication and rolling upgrade features provided in Oracle Application Server.

Note: If you have not configured the Application Server for clustering, then you cannot perform this exercise.

View Session Replication in Action

OC4J provides a flexible framework for creating a clustered environment for development and production purposes. In this context, a cluster is defined as two or more OC4J server nodes hosting the same set of applications. The OC4J clustering framework supports:

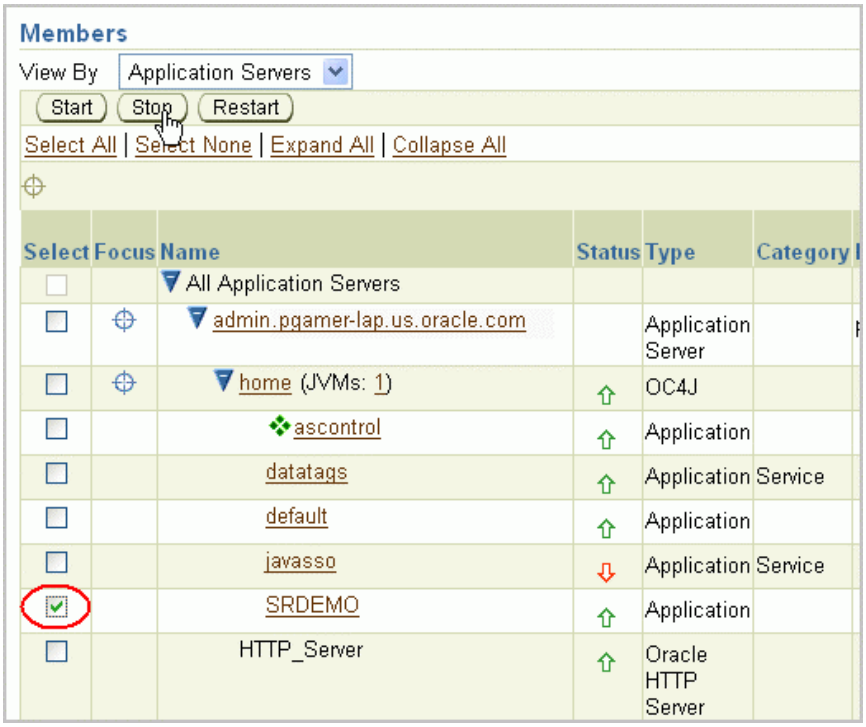
- Replication of objects and values contained in an HTTP session or a stateful session Enterprise JavaBean (SFSB) instance
- In-memory replication using multicast or peer-to-peer communication, or persistence of state data to a database

- Load balancing of incoming requests across OC4J instances
- Transparent failover across applications within the cluster
- Configuration within an OC4J instance at either the global server or application level

You can see evidence that HTTP session state is being replicated by checking the name of the logged-in user. The User object is stored on the session and is persisted across both OC4J nodes within your cluster.

To demonstrate, you stop the application on the OC4J instance that is installed with your HTTP server instance. You then reload the application to see that you are still logged in.

1. Click the **Cluster Topology** link in the breadcrumbs at the top of the page to return to the Cluster Topology page.
2. Expand the `home` instance of your administrator instance—that is, the instance that Application Server Control is installed on (`admin` in the example).
3. Select the check box next to **SRDEMO**, and then click **Stop** to stop the application instance.



   On the Confirmation page, click **Yes**.

   You should receive a confirmation message indicating that the SRDEMO application in the `admin` instance has been stopped. You now have only one instance of the application running on your non-administrator instance—the instance that does not have Application Server Control installed.

4. In the other browser window where the application is running, reload the SRDEMO application (**View → Reload** in Firefox). The "You are logged in as:" field should continue to display the name of the logged-in user. Do not log off, but leave this browser open for the next exercise. **Note:** The first time that you reload the application, you may receive a "File Not Found" error. If this happens, just enter the URL in the browser again. You should then see the logged-in user without logging in again.

Perform a Rolling Upgrade

A rolling upgrade is a process in which you simultaneously re-deploy an application to multiple OC4J nodes within a cluster, while still maintaining application state.

While Oracle Application Server undeploys then redeploys the application on one OC4J instance, existing user sessions and incoming requests continue to be serviced by the application instance running on another OC4J instance within the cluster. This is an extremely useful feature in a real-world production environment, essentially allowing you to redeploy an application across a cluster without having to first stop or it cancel existing user sessions.

To perform the rolling upgrade, you first modify the original EAR file, and then redeploy it to your cluster using the `admin_client.jar` command line tool. The utility deploys the EAR to each instance sequentially, so that at least one instance of the application is always available.

1. In JDeveloper, in the Applications Navigator expand the **ViewController** project, and then expand the **Web Content** and **css** nodes. Double-click **jdeveloper.css** to open it in the editor.
2. Change the color for the `H1` tag to **`FF0000`**, the hexadecimal notation for the color red.

```
H1 {
        font-family: Arial, Helvetica, sans-serif;
        font-size:170%;
        color:#FF0000;
        border : solid #CCCC99;
        width : 100%;
        border-width : 0px 0px 2px 0px;
}
```

3. Click **Save All** to save your work.
4. In the Applications Navigator, expand the **Deployment** project, and then expand the **Resources** node.
5. Right-click **SRDEMO.deploy** and select **Deploy to EAR file** from the context menu.
6. Now redeploy the application using the `admin_client.jar` utility. To do so, open a command prompt and navigate to the `ORACLE_HOME/j2ee/home` directory on the `admin` OC4J instance (the instance hosting your Application Server Control).
7. Type the following command to invoke `admin_client.jar` and redeploy the modified EAR:

```
java -jar admin_client.jar deployer:cluster:opmn://
<hostame>:<opmnRequestPort>/<oc4jGroupName>
oc4jadmin <password> -redeploy -file <path/filename.ear>
-deploymentName <appName> -sequential -keepsettings
```

Supply the following parameters:

Parameter	Value
`password`	Supply your oc4jadmin account password, which should be `welcome1`.
`hostname`	Specify the name of the OC4J host machine.
`opmnRequestPort`	The OPMN request port for the node, which is defined in the `<port>` element within the `opmn.xml` file installed in the `ORACLE_HOME/opmn/conf` directory for the **administrative** OC4J instance. Look for this entry in the file:

Parameter	Value
	`<port local="6100" remote="6200" request="6003"/>`
`oc4jGroupName`	Set this value to the name of your OC4J instance group; in this case, `default_group`.
`Path/filename`	Supply the path and file name to your modified EAR.
`appName`	Supply the name of the application, which should be `SRDEMO`.

For example:

```
C:\oracle\product\10.1.3\AS10g\j2ee\home>java -jar
admin_client.jar
deployer:cluster:opmn://localhost:6003/default_group oc4jadmin
welcome1 -redeploy -file
c:\JDeveloper\mywork\SRDEMO\Deployment\deploy\SRDEMO.ear
-deploymentName SRDEMO -sequential -keepsettings
```

Note: As the command executes, continue with Step 8.

The SRDEMO application is now stopped, undeployed, redeployed, and then restarted to each of your Oracle Application Server instances.

Note that the deployment plan you created the first time the application was deployed is reused during the redeployment. As such, your previous configuration is automatically applied to the redeployed application.

8. Continue to reload the application in your Web browser as the `redeploy` command is executing. You should continue to see the same login name displayed with each click, eventually noting the change in application appearance as the modified version is deployed, as shown in the following screenshot:

This illustrates that state has effectively been maintained during the deployment of the new, modified version of the application.

Summary

In this chapter, you deployed the application and performed a rolling upgrade of the application.

You performed the following key tasks in this chapter:

- Made the application distributable so that it can run simultaneously in multiple Web containers
- Created a Deployment project to contain deployment details
- Created deployment profiles for the Model (`.jar` file), ViewController (`.war` file), and Deployment (`.ear` file) projects

- Created deployment descriptors to port application details to Java EE
- Used Application Server Control to create the connection pool and data source for the application
- Deployed the application to the OC4J cluster on the application server
- Tested the application in a browser with the application server URL
- Viewed application metrics, users, and roles in Application Server Control
- Performed rolling upgrade on the application in the cluster while preserving application state