

---

# **Appendix A**

## **Practices and Solutions**

---

## Table of Contents

Practices for Lesson 1 .....	4
Practices for Lesson 2 .....	5
Practice 2-1: Creating JDeveloper Connections .....	6
Practice 2-2: Browsing the FOD Schema Data.....	8
Practices for Lesson 3 .....	9
Practice 3-1: Run the EJB/JSF Application.....	10
Practice 3-2: Exploring the Model Project.....	11
Practice 3-3: Exploring the UI Project.....	12
Practices for Lesson 4 .....	14
Practice 4-1: Developing a Simple Servlet Application .....	15
Practice 4-2: Developing a Servlet Application to Access a Database.....	17
Practices for Lesson 5 .....	21
Practice 5-1: Developing a Simple JSP Application.....	22
Practice 5-2: Developing a JSP Application Using a JavaBean .....	27
Practice 5-3: Developing a JSP Application Using Custom Tags and Expression Language.....	32
Practices for Lesson 6 .....	35
Practice 6-1: Accessing Database Connectivity in a Servlet Application by Implementing Dependency Injection (DI) .....	36
Practice 6-2: Injecting an Enterprise Java Bean by Implementing DI.....	38
Practices for Lesson 7 .....	40
Practice 7-1: Creating a Stateless Session Bean .....	41
Practice 7-2: Creating a Sample Test Java Client for the CreditCardValidator Stateless Session Bean .....	43
Practice 7-3: Calling a Stateless Session Bean in a Stateful Session Bean .....	45
Practice 7-4: Creating a Sample Test Java Client for the ShoppingCart Stateful Session Bean .....	48
Practice 7-5: Using Interceptors in the ShoppingCart Session Bean.....	50
Practices for Lesson 8 .....	53
Practice 8-1: Coding a Simple Entity Bean .....	54
Practice 8-2: Using the JDeveloper Entity Bean Wizard.....	58
Practice 8-3: Customizing Entity Beans .....	59
Practice 8-4: Creating and Testing a Session EJB .....	62
Practices for Lesson 9 .....	64
Practice 9-1: Adding Named Queries to Person .....	65
Practice 9-2: Adding Named Queries to Product.....	67
Practice 9-3: Adding Named Queries to Category .....	69
Practices for Lesson 10 .....	71
Practice 10-1: Creating the ValidateCreditCardService Web Service by Using the Top-Down Web Service Development Approach .....	72
Practice 10-2: Testing the Web Service by using JDeveloper.....	77
Practices for Lesson 11 .....	78
Practice 11-1: Creating the ProductBrowsing Managed Bean .....	79
Practice 11-2: Creating a JSF Page.....	83

Practice 11-3: Adding Products to the Page .....	85
Practices for Lesson 12 .....	89
Practice 12-1: Creating the Edit and Add Pages .....	90
Practice 12-2: Building the AddProduct Page .....	92
Practice 12-3: Creating the Add Product Functionality .....	94
Practices for Lesson 13 .....	98
Practice 13-1: Adding the Category Tree Selection Event .....	99
Practice 13-2: Complete and Connect the Edit Product Page.....	103
Practices for Lesson 14 .....	106
Practice 14-1: Creating a JMS Connection Factory and Queue Destination in WebLogic Server .....	107
Practice 14-2: Creating a Message-Driven Bean to Send Emails.....	109
Practice 14-3: Creating a Session Method for Sending Email Messages .....	112
Practice 14-4: Creating a Sample Test Client in JDeveloper.....	115
Practice 14-5: Testing the Confirmation Email .....	117
Practices for Lesson 15 .....	118
Practice 15-1: Working with Container-Managed Transactions .....	119
Practice 15-2: Working with EJB Client's Transactional Context.....	121
Practices for Lesson 16 .....	126
Practice 16-1: Enabling Form-based Login Authentication .....	127
Practice 16-2: Creating Security Roles and URL Constraints in the Deployment Descriptors .....	128
Practice 16-3: Creating the Users and Groups in WebLogic Server .....	132
Practice 16-4: Testing the Security Implementation.....	134

---

## Practices for Lesson 1

---

**There is no practice for this lesson.**

---

## Practices for Lesson 2

---

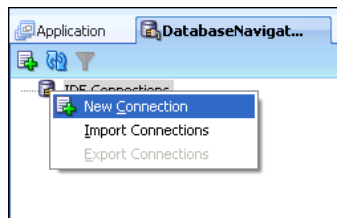
The aim of the practices for this lesson is to explore and configure basic components in the course development environment. You perform the following set of tasks in this practice set:

- Creating JDeveloper connections for the FOD database schema
- Examining some of the seeded data in the FOD schema tables

## Practice 2-1: Creating JDeveloper Connections

In this practice, you launch Oracle JDeveloper and create connections for it to the database and application server.

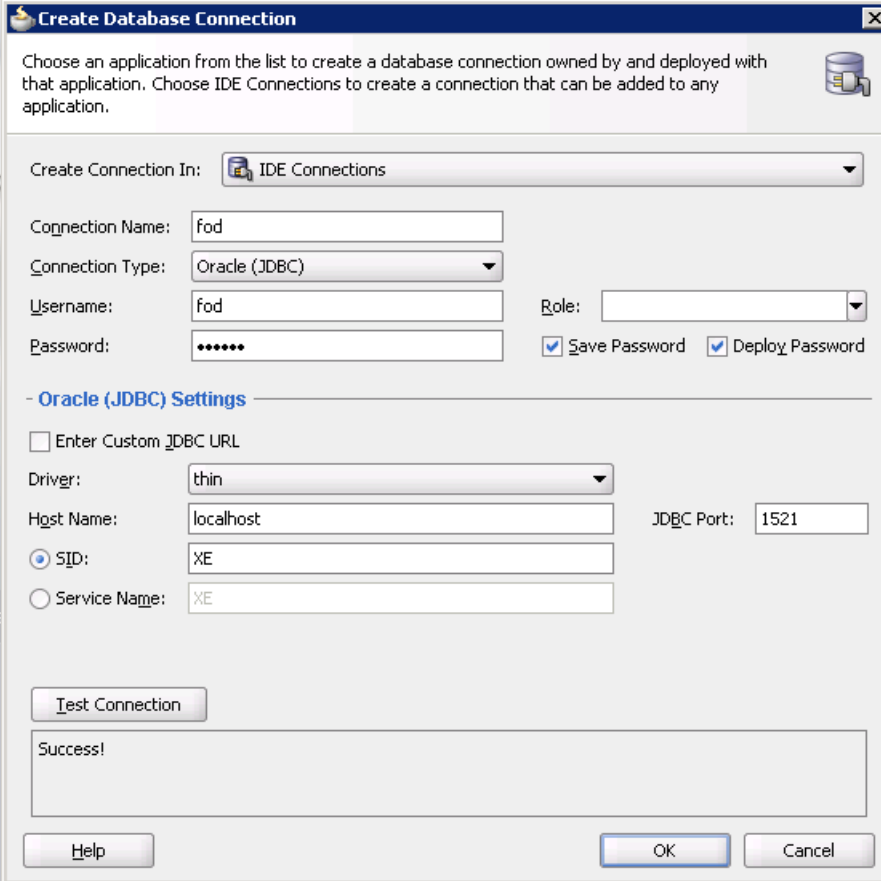
- 1) Start Oracle JDeveloper:
  - a) Click Start > Programs > Oracle Fusion Middleware > JDeveloper Studio.
  - b) In the Select Role dialog box, select **Default Role**, deselect **Always prompt for role**, and click **OK**.
  - c) When JDeveloper starts, you may receive a message to migrate from a previous version of the product. Click **No**.
  - d) If the Configure File Type Associations window is displayed, click **OK**.
  - e) In the Tips of the Day window, deselect the **Show tips at startup** option and click **Close**.
- 2) Create a database connection (FOD) that connects to the Oracle XE Database instance. General IDE level connections are created in the Database Navigator.
  - a) Open the DatabaseNavigator page by selecting View > Database Navigator.
  - b) Right-click IDE Connections and select **New Connection**.



- c) Use the following connection details or use the screenshot as a visual guide:

Step	Screen/Page Description	Choices or Values
i.	Step 1 of 4: Type	Connection Name: fod Connection Type: Oracle (JDBC)
ii.	Step 2 of 4: Authentication	Username: fod Password: fusion Select the <b>Save Password</b> check box. Select the <b>Deploy Password</b> check box.
iii.	Step 3 of 4: Connection	Driver: thin Host Name: localhost JDBC Port: 1521 SID: XE
iv.	Step 4 of 4: Test	Click Test Connection to check whether the connection is successfully established. If the status is "Success," click OK.

## Practice 2-1: Creating JDeveloper Connections (continued)



The image shows a 'Create Database Connection' dialog box. At the top, it says 'Choose an application from the list to create a database connection owned by and deployed with that application. Choose IDE Connections to create a connection that can be added to any application.' Below this is a dropdown menu 'Create Connection In:' with 'IDE Connections' selected. The 'Connection Name' is 'fod', 'Connection Type' is 'Oracle (JDBC)', 'Username' is 'fod', and 'Password' is masked with dots. There are checkboxes for 'Save Password' and 'Deploy Password', both checked. A section titled '- Oracle (JDBC) Settings' contains an unchecked checkbox 'Enter Custom JDBC URL'. Below it, 'Driver' is 'thin', 'Host Name' is 'localhost', 'JDBC Port' is '1521', 'SID' is selected with value 'XE', and 'Service Name' is 'XE'. A 'Test Connection' button is present, and below it, a text area shows 'Success!'. At the bottom are 'Help', 'OK', and 'Cancel' buttons.

Create Database Connection

Choose an application from the list to create a database connection owned by and deployed with that application. Choose IDE Connections to create a connection that can be added to any application.

Create Connection In: IDE Connections

Connection Name: fod

Connection Type: Oracle (JDBC)

Username: fod

Password: \*\*\*\*\*

Role:

☒ Save Password ☒ Deploy Password

- Oracle (JDBC) Settings

☐ Enter Custom JDBC URL

Driver: thin

Host Name: localhost

JDBC Port: 1521

☒ SID: XE

☐ Service Name: XE

Test Connection

Success!

Help OK Cancel

## ***Practice 2-2: Browsing the FOD Schema Data***

In this practice, you use JDeveloper to browse a database schema object by using the database connection created in practice 2-1. You examine some of the FOD schema table structures and their data.

- 1) On the Database Navigator tabbed page, expand IDE Connections > fod database connection and locate and examine the structure of the PERSONS table in the FOD schema.
  - a) In the Connections pane, click and expand the fod connection.  
**Note:** Expand the Database node, if necessary.
  - b) The FOD schema name should appear. Click the expand icon to expose the nodes for the different types of database objects.
  - c) Expand the Tables node.
  - d) Double-click the PERSONS table entry. JDeveloper opens a window showing the table definition in the Structure pane.
- 2) Examine the data stored in the PERSONS table.
  - a) Click the Data subtab at the bottom of the PERSONS tabbed page.
  - b) Approximately 50 rows of data should be present. The email address is used by the course application as the user ID for login purposes.

- 3) Optionally, examine the data in other tables, such as CATEGORIES and PRODUCTS.  
**Note:** Use the technique described in step 2 of this practice section. You can insert, update, and delete data in the Data tabbed page. Alternatively, right-click the fod connection and select **SQL Worksheet**. By using the SQL Worksheet, you can enter SQL statements to query and manipulate the data.

- 4) Close all open database table tabbed pages.

**Note:** Select each tabbed page and click **File > Close**, or click the Close icon next to the name on the tab.



## Practices for Lesson 3

During this course, you create several small applications that demonstrate a number of Java EE development techniques. One of the applications that you will build is an MVC application based on an EJB model with a Session Façade and a JSF user interface. In this practice, you run, and then explore a pre-built application as a sample of the application you will build in later lessons.

### ***Practice 3-1: Run the EJB/JSF Application***

In this practice, you run the application and explore how each part works.

- 1) Open `Application_03.jws`.
  - a) In the Application Navigator, select **Open Application**.
  - b) Navigate to `D:/labs/labs/Application_03` folder.
  - c) Double-click `Application03.jws`.
- 2) Expand **UI** in the Application Navigator.
- 3) Expand **Web Content**.
- 4) Right-click `Main.jspx` and select **Run** from the Context Menu. This action deploys and starts the Weblogic server, deploys the Model and UI projects, and runs the Main page.
- 5) On the running Main page, expand one of the categories in the category tree and select one of the subcategories.
- 6) Notice how the products that belong to that category are displayed in the products table.
- 7) Select other categories to see the products table change.
- 8) Add a product of your own by using the Add Product button at the upper-right corner of the Main page.
  - a) Click **Add a Product**.
  - b) Notice that there are field level hints for each of the fields on the page.
  - c) Try to enter a Name that is less than five characters. There is a validator on the field that requires at least five characters for the name. Notice the error, then enter a name with at least five characters.
  - d) Enter values in all the fields and click **Save**.
  - e) Notice that your product is added to the currently selected category.
- 9) Edit your product.
  - a) Click the product name of the product you just added.
  - b) On the Edit page, change any or all of the values and click Save.
  - c) Notice that the changes you made are reflected in the products table.
- 10) Close the browser after you finish exploring the application.

### ***Practice 3-2: Exploring the Model Project***

In this practice, you explore the Model project in the application.

- 1) Expand **Model** in the Application Navigator.
- 2) Expand **Application Sources**.
- 3) Notice the two packages, `oracle.model` and `oracle.services`. The `model` package contains the entity beans for the project while the `services` project contains the session beans.
- 4) Expand `oracle.model`.
- 5) Notice that there are three entity beans: `Category`, `Person`, and `Product`.
- 6) Double-click `Category.java` to see the code in the source editor.
- 7) Notice that there are three Named Queries. These queries are written specifically to support parts of the application that you will be building. For example, the `Category.findRoot` query returns only those categories where there is no parent—that is, the Root categories. It excludes any category that has a parent. This query is used in building the category tree component that you saw in the application that is running.
- 8) While you are in this package, explore the other entity beans to see what queries or methods they contain.
- 9) Expand the `oracle.services` package.
- 10) Notice that there are three items in the package: `SessionEJBBean.java` is the session bean, and `SessionEJB.java` and `SessionEJBLocal.java` are the remote and local interfaces to the session bean, respectively.
- 11) Open `SessionEJBBean.java`.
- 12) Notice that there are methods that expose each of the Named Queries defined in the entity beans. These methods publish access to those queries through the local and remote interfaces.
- 13) When you are done, close the files.

### ***Practice 3-3: Exploring the UI Project***

In this practice, you explore the UI project and its components.

- 1) Expand **UI** in the Application Navigator.
- 2) Expand **WEB-INF**.
- 3) Open `faces-config.xml`.
- 4) The file opens with the visual editor. The Diagram of the application should be displayed, if not, click the Diagram tab at the bottom of the editor.
- 5) Notice that there are three pages: `Main.jspx`, `EditProduct.jspx`, and `AddProduct.jspx`.
- 6) Between the pages are Navigation cases, which control which page can call or navigate to which page. Buttons or links on the pages use these Navigation cases.
- 7) Double-click `Main.jspx`.
- 8) Notice that there are two areas on the page: one that displays Categories and the other that displays Products. These areas are coordinated through an event. When the user clicks a new category, the event fires and refreshes the Products table with the related products.
- 9) Notice that the Structure window shows the structure of the entire page.
- 10) In the Visual Editor, click in the categories tree. Note that the Structure window reflects what item you selected.
- 11) Find and click `af:tree - categoryTree` in the Structure window, if it is not selected.
- 12) Expand the Behavior node in the Properties Inspector. If you cannot find the Properties Inspector, open it from the **Menu View > Property Inspector** or press and hold **Ctrl + Shift + I**.
- 13) Note that the Selection Listener property points to a method in `productBrowsingBean`. This method traps the selection event and sets a variable to hold the current selected category. This is the first part of coordinating the category tree and the Products table. This application uses this managed bean to hold values and methods required for this application.
- 14) Click **productsTable**.
- 15) In the Properties Inspector, expand Behavior and notice the **PartialTriggers** property. This is set to `::categoryTree` so that if there is a change in the category tree (such as a user selecting a new category), the product table refreshes itself. This is the other part of the coordination mechanism.
- 16) Expand the Application Sources > `oracle.ui.backing` package.
- 17) Open `ProductBrowsingBean.java`.

### ***Practice 3-3: Exploring the UI Project (continued)***

- 18) Explore this class. Note that there are a number of set and get methods. Most of those correspond to the components you add to the three pages. These are added automatically in some cases, which you will see later.
- 19) Examine the code in `getProductsForSelectedCategory()`. You can use the search or the Structure window to find the method. Notice that it uses `selectedCategory`, which is set in the category selection event that you just explored. This method retrieves and returns `productList` for the selected category.
- 20) Go back to `Main.jspx`.
- 21) Select **productsTable**.
- 22) Notice that the value in Property Inspector is `{productBrowsingBean.productsForSelectedCategory}`. This value is the result of the method you just examined. When this table needs to be refreshed, it calls `getProductsForSelectedCategory` and displays the results.
- 23) Explore any part of the application that you want.
- 24) When you are done, close the application. You can also remove the application from the IDE if you choose.

## Practices for Lesson 4

The aim of the practices for this lesson is to develop a servlet application. You learn to develop a simple servlet application by using Oracle JDeveloper 11g integrated development environment (IDE), and execute it from a Web browser. Afterwards, you modify a login HTML form that accepts a username and password, and makes a call to the servlet application. The servlet application extracts the user credentials from the HTML form and validates it against a database table.

Following are the tasks that you need to perform for this practice set:

- Develop a simple servlet application in JDeveloper 11g.
- Test the servlet in a Web browser.
- Modify a login HTML form to access the servlet.
- Modify the servlet application to extract the user credentials, connect to a database table, and validate the user.

## Practice 4-1: Developing a Simple Servlet Application

In this practice, you develop a servlet application by using JDeveloper 11g.

- 1) In JDeveloper, open `Application_04.jws`.  
**Note:** This opens a skeleton application, the Servlets project, for the servlet application components.
  - a) In JDeveloper, select **File > Open**.
  - b) In the Open window, open the `Application_04` directory.
  - c) Double-click the `Application_04.jws` file.
- 2) Create an HTTP servlet called `LoginValidatorServlet` in the Servlets project.  
Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Application Navigator	Right-click <b>Servlets</b> . Select <b>New</b> .
b.	New Gallery	Categories: <b>Web Tier &gt; Servlets</b> Items: <b>HTTP Servlet</b> Click <b>OK</b> .
c.	Create HTTP Servlet: Welcome	Click <b>Next</b> .
d.	Create HTTP Servlet - Step 1 of 3	Class: <code>LoginServlet</code> Package: <code>oracle.servlets</code> Generate Content Type: <code>HTML</code> Implement Methods: Select the <code>doGet ()</code> option. Select the <code>doPost ()</code> option. Click <b>Next</b> .
e.	Create HTTP Servlet - Step 2 of 3	Name: <code>LoginServlet</code> URL Pattern: <code>/loginervlet</code> Click <b>Next</b> .
f.	Create HTTP Servlet - Step 3 of 3	Click <b>Finish</b> .

- 3) Analyze the code in the `doGet ()` and `doPost ()` methods in the `LoginServlet` servlet class.
- 4) Compile the servlet class. In the Application Navigator pane of JDeveloper, right-click `LoginServlet.java` and select the **Make** option from the shortcut menu. Check the Message log to verify the successful compilation of the application.

### ***Practice 4-1: Developing a Simple Servlet Application (continued)***

- 5) Test the servlet application. In the Application Navigator pane, right-click `LoginServlet.java` and select the **Run** option from the shortcut menu. Verify the output of the servlet application in the Web browser.



## Practice 4-2: Developing a Servlet Application to Access a Database

In this practice, you develop a login validation application by using the HTML form and HTTP servlet.

- 1) Modify an HTML form for logging in to an application.
  - a) Expand the **Servlets** project and the **Web Content** folder. Open the `Login.html` file in the code editor by double-clicking the file.

There are two text areas in this form. (You can switch between the visual and code views using the Design and Source tabs.) The text areas have labels, but no names.
  - b) Select each input item and using the Property Inspector, set the Value properties to `user_name` and `user_password`, respectively. These are the values that you pass to the request object and, therefore, it is important to follow standard naming conventions so that you can refer to the value later. To view the Property Inspector, if not available, select **View** > **Property Inspector**.
  - c) Modify the form so that the action tag points to `LoginServlet`. In the Structure window, select the **form** node. In the Property Inspector pane, set the Action property to `/loginservlet`.
  - d) Save the `Login.html` file.
- 2) Modify the HTTP servlet to handle the login. Add the following additional code in the `LoginServlet.java` file.
  - a) Import the `java.sql` package and declare the following variables (in bold) with their respective values:

```
...  
  
import java.sql.*;  
  
...  
  
public class LoginServlet extends HttpServlet {  
    private static final String CONTENT_TYPE = "text/html";  
        charset=windows-1252";  
    private static final String DB_URL =  
        "jdbc:oracle:thin:@localhost:1521:XE";  
    private static final String DB_USERNAME = "fod";  
    private static final String DB_PASSWORD = "fusion";  
    private static Connection con;  
    private String pass;  
    private String name;  
    ...  
}
```

- b) Define and implement the `configureConnection()` method to create a JDBC connection to the database.

## ***Practice 4-2: Developing a Servlet Application to Access a Database (continued)***

```
...

public void configureConnection() throws SQLException {
    try{
        Class.forName("oracle.jdbc.OracleDriver");
        con = DriverManager.getConnection(DB_URL, DB_USERNAME,
                                         DB_PASSWORD);

        con.setAutoCommit(true);
    }
    catch (Exception e){
        System.out.println("Connection failed: " +e.toString());
    }
}

...
```

- c) Define and implement the `getConnection()` method to retrieve the JDBC connection.

```
...

public Connection getConnection() throws SQLException
{
    configureConnection();
    return con;
}

...
```

- d) Define and implement the `verifyPassword()` method to validate the user's credentials.

```
...

protected boolean verifyPassword(String theuser, String
password) {
    String originalPassword=null;

    try {
        con=getConnection();
        Statement stmt= con.createStatement();
        stmt.executeQuery("select password from login where
                           uname='"+theuser+"'");
        ResultSet rs = stmt.getResultSet();
        if(rs.next()) {
            originalPassword=rs.getString(1);
        }
        stmt.close();
    }
```

## Practice 4-2: Developing a Servlet Application to Access a Database (continued)

```
        if(originalPassword.equals(password)) {
            return true;
        }
        else {
            return false;
        }
    }
    catch (Exception e){
        System.out.println("Exception in
                           verifyPassword()="+e.toString());
        return false;
    }
}
...

```

- e) Add the code to the `doPost()` method to retrieve the `user_name` parameter from the request object. Remember that the value being passed is the parameter that you specified in step 1.b, (not “User Name”). The code extracts the user credentials from the HTML form, validates the user, and prints appropriate messages in HTML. Note that the `out.println` statements replace the default messages in the `doPost` method.

```
...

public void doPost(HttpServletRequest request,
                  HttpServletResponse response) throws ServletException,
                  IOException{
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();

    name = request.getParameter("user_name");
    pass = request.getParameter("user_password");
    boolean result = verifyPassword(name, pass);
    out.println("<html>");
    out.println("<body>");
    if (result == true){
        out.println ("Hello " + name + ": Your login module is
                                working great!");
    }
    else{
        out.println ("Invalid user name or password");
    }
    out.println("</body></html>");
    out.close();
}

...

```

### ***Practice 4-2: Developing a Servlet Application to Access a Database (continued)***

- f) Right-click `LoginServlet.java` and select **Make** to compile the class.
- 3) Run `Login.html` to test the functionality.
  - a) Right-click `Login.html` and select **Run**.
  - b) Enter the username as `sking` and password as `oracle`, and then click Login.
- 4) When you are done, close the browser and close `Application_04`. You can also remove it from the IDE if you want.

In Web-based applications, it is common that data from one page is used on others. There are several ways to exchange data between pages. One way is to use explicit objects such as the JSP Session object. You can also use a JavaBean to store data, as well as execute business logic and data validation. With JSPs, you can also create custom tags that perform specific formatting or logic that you may use across applications.

In this practice, you create a JavaServer Page that pass values using Session variables. You also use a JavaBean to execute some business logic. In the last section of this practice, you create several custom tags to perform special formatting for your page.

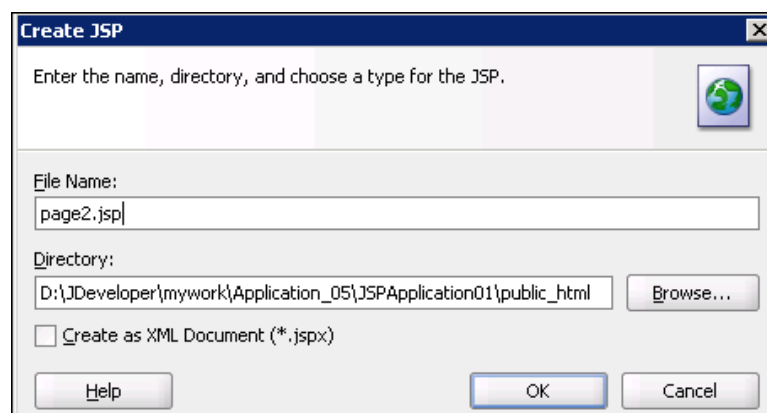
## Practice 5-1: Developing a Simple JSP Application

In this practice, you create a JavaServer Pages (JSP) application that interacts with two other pages. The first page accepts a couple of values and when the user clicks the Submit button, the second page appears.

The second page displays the values for the first page, creates session variables, and accepts two additional values. When the user clicks the Submit button, the third page appears displaying the values from the first two.

- 1) Open `Application_05.jws`.  
**Note:** This opens a skeleton of all the JSP application projects used in this practice.
  - a) In JDeveloper, select **File > Open** or choose **Open Application** in the Application Navigator.
  - b) In the Open window, open the `Application_05` directory.
  - c) Double-click the `Application_05.jws` file.
- 2) Expand the **JSPApplication01** project in the Application Navigator pane.
- 3) Create a JSP page called **page2.jsp** in the JSPApplication01 project. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Application Navigator	Right-click <b>JSPApplication01</b> . Select <b>New</b> .
b.	New Gallery	Categories: <b>Web Tier &gt; JSP</b> Items: <b>JSP</b> Click <b>OK</b> .
c.	Create JSP	File Name: <code>page2.jsp</code> Click <b>OK</b> .



### Practice 5-1: Developing a Simple JSP Application (continued)

- 4) Add two text fields and a submit button to the JSP page and include a form tag when prompted. Name the text fields `streetAddress` and `city`. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Component Palette	Select <b>HTML &gt; Forms</b> .
b.	Component Palette	Click <b>Forms &gt; Text Field</b> .
c.	Insert Text Field Wizard	Name: <code>streetAddress</code> Click <b>OK</b> .
d.	Insert Form	Action: <code>page3.jsp</code> Click <b>OK</b> .
e.	page2.jsp	Enter <code>Street Address</code> : as plain text before the text field. Click to the right of the empty text field and press Enter to insert a new line.
f.	Component Palette	Select <b>HTML &gt; Forms</b> .
g.	Component Palette	Click <b>Forms &gt; Text Field</b> .
h.	Insert Text Field Wizard	Name: <code>city</code> Click <b>OK</b> .
i.	page2.jsp	Enter <code>City</code> : as plain text before the text field. Click to the right of the empty text field and press Enter to insert a new line.
j.	Component Palette	Select <b>HTML &gt; Forms</b> .
k.	Component Palette	Click <b>Forms &gt; Submit Button</b> .
l.	Insert Submit Button Wizard	Value: <code>Next page</code> Click <b>OK</b> .

Street Address:

City:

Next Page

### Practice 5-1: Developing a Simple JSP Application (continued)

- 5) In the next set of steps, you add tags to display the `firstName` and `lastName` fields from `page1.jsp`. You also add `Set` tags to store those values on the implicit `session` object. This makes those values available on any page accessed within the current user session. In `page3.jsp`, you use the `session` and `request` objects to display the values the user has entered in both `page1.jsp` and `page2.jsp` pages. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	page2.jsp	Place the cursor to the left of the form (the dotted gray line) and press Enter to insert a new line. This inserts a new line above the form tag.
b.	Component Palette	Select <b>JSTL &gt; Core</b> .
c.	Component Palette	Click <b>Core &gt; Out</b> .
d.	Insert Out	Value: <code>\${param.firstName}</code> Click <b>OK</b> .
e.	page2.jsp	Place the cursor to the right of the new <code>Out</code> tag and press Enter to add a new line.
f.	Component Palette	Select <b>JSTL &gt; Core</b> .
g.	Component Palette	Click <b>Core &gt; Out</b> .
h.	Insert Out	Value: <code>\${param.lastName}</code> Click <b>OK</b> .

The screenshot shows a JSP page with two JSTL `Out` tags at the top, each displaying the value of a parameter. Below them is a form area enclosed in a dotted gray border. The form contains two input fields labeled "Street Address:" and "City:". At the bottom of the form area is a button labeled "Next Page".

- 6) Now that you have added the display tags, you add tags to store the values of `firstName` and `lastName` on the `session` object. You use a `Set` tag for this. Use the following details to accomplish this task:



### Practice 5-1: Developing a Simple JSP Application (continued)

Step	Screen/Page Description	Choices or Values
a.	page2.jsp	Place the cursor to the right of the second Out tag and press Enter to add a new line.
b.	Component Palette	Select <b>JSTL &gt; Core</b> .
c.	Component Palette	Click <b>Core &gt; Set</b> .
d.	Set – Property Inspector	Value: <code>\${param.firstName}</code> Var: <code>firstName</code> Scope: <code>session</code>
e.	page2.jsp	Place the cursor to the right of the Set tag and press Enter to add a new line.
f.	Component Palette	Select <b>JSTL &gt; Core</b> .
g.	Component Palette	Click <b>Core &gt; Set</b> .
h.	Set – Property Inspector	Value: <code>\${param.lastName}</code> Var: <code>lastName</code> Scope: <code>session</code>

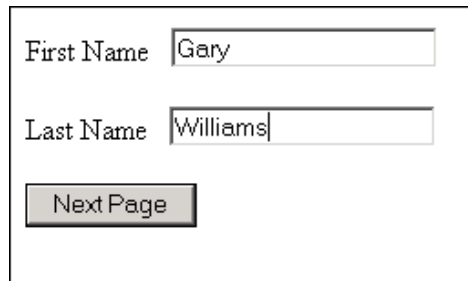
The screenshot shows a JSP page with the following elements:

- Two JSTL Out tags: `Out - ${param.firstName}` and `Out - ${param.lastName}`.
- Two JSTL Set tags: `Set - firstName` and `Set - lastName`.
- A dashed box containing two text input fields labeled "Street Address:" and "City:".
- A "Next Page" button below the dashed box.

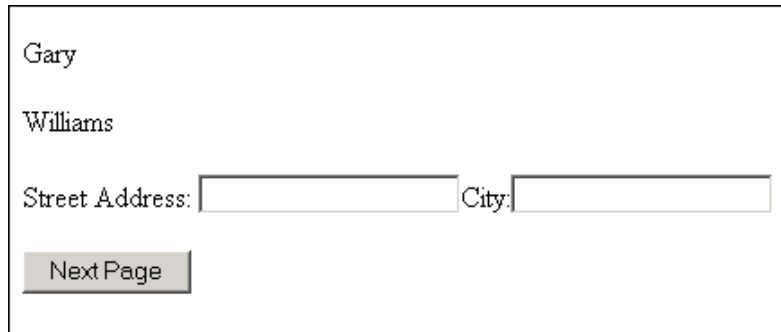
- 7) Run `page1.html` to test the functionality.
- a) Right-click `page1.jsp` and select **Run**.
  - b) Enter values in the First Name and Last Name fields, and then click Next Page.
  - c) Enter values in the Street Address and City fields, and then click Next Page.

### ***Practice 5-1: Developing a Simple JSP Application (continued)***

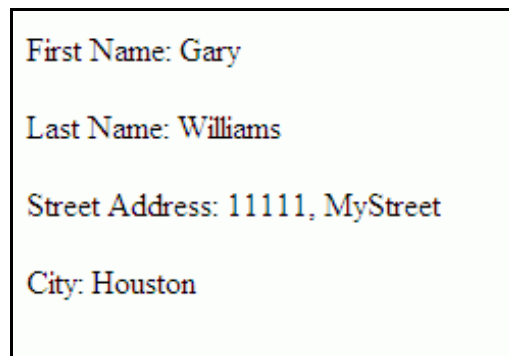
- d) The result pages should look something like the following:



A screenshot of a web form. It contains two text input fields. The first is labeled "First Name" and contains the text "Gary". The second is labeled "Last Name" and contains the text "Williams". Below these fields is a button labeled "Next Page".



A screenshot of a web page showing the output of the first form. It displays the text "Gary" and "Williams" on separate lines. Below these is a form for "Street Address" and "City", each with an empty text input field. At the bottom is a button labeled "Next Page".



A screenshot of a web page showing the final output. It displays the text "First Name: Gary", "Last Name: Williams", "Street Address: 11111, MyStreet", and "City: Houston" on separate lines.

- e) When you have verified that your page works, close the browser.

## ***Practice 5-2: Developing a JSP Application Using a JavaBean***

In this practice, you modify a JavaBean that exposes a couple of properties that you use in your JSP application. You modify the code in the JSP page to invoke the JavaBean. You also add the code to include an HTML document in the JSP page.

The BookCart application enables you to add and remove books from a shopping cart. You can also view the details of the books that you want to add in the shopping cart.

- 1) In the Application Navigator, expand **JSPApplication02**.

**Note:** Minimize or close the JSPApplication01 project.

- 2) Navigate to `CartBean.java` in the Application Navigator and double-click it to open in the code editor. Modify the code in the `CartBean` class as follows:

**Hint:** Expand `JSPApplication02 > Application Sources > oracle.jsp > CartBean.java`.

- a) Define the following two properties (in bold) in the `CartBean` class:

```
...  
  
public class CartBean {  
    Vector vector = new Vector();  
    String submit = null;  
    String item = null;  
    ...  
}
```

- b) Define and implement the `addItem()` and `removeItem()` methods to add and remove books, respectively in a shopping cart.

```
...  
  
public CartBean() {  
}  
  
    private void addItem(String name) {  
        vector.addElement(name);  
    }  
  
    private void removeItem(String name) {  
        vector.removeElement(name);  
    }  
    ...  
}
```

- c) Define and implement the `getItems()` method to retrieve the shopping details from the shopping cart.

## ***Practice 5-2: Developing a JSP Application Using a JavaBean (continued)***

```
...  
    public String[] getItems() {  
        String[] s = new String[vector.size()];  
        vector.copyInto(s);  
        return s;  
    }  
...
```

- d) Define and implement the `processRequest()` method that you will be using in your JSP page to invoke the `CartBean`.

```
...  
  
public String[] getItems() {  
    String[] s = new String[vector.size()];  
    vector.copyInto(s);  
    return s;  
}  
  
public void processRequest() {  
    if (submit == null)  
        addItem(item);  
    if (submit.equals("Add"))  
        addItem(item);  
    else if (submit.equals("Remove"))  
        removeItem(item);  
    reset();  
}  
...
```

- 3) Save and compile the `CartBean.java` file.
  - a) Select the **Save** option from the File menu, or click the Save icon in the toolbar.
  - b) Right-click `CartBean.java` in the Application Navigator and select the **Make** option from the context menu.
  - c) View the Message-Log to verify the successful compilation.
- 4) Navigate to `BookCatalog.jsp` in the Application Navigator and double-click it to open in the visual editor. Click the **Source** tab to view the JSP code. Modify the code in the `BookCatalog.jsp` as follows:

**Hint:** Expand `JSPApplication02 > Web Content > BookCatalog.jsp`.

## ***Practice 5-2: Developing a JSP Application Using a JavaBean (continued)***

- a) Add the following code (in bold) to reference the CartBean in the JSP application, and to set the bean properties.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ page contentType="text/html; charset=windows-1252"%>
<html>
<jsp:useBean id="cart" scope="session"
              class="oracle.jsp.CartBean" />

<jsp:setProperty name="cart" property="*" />

...
```

- b) Add the following scriptlet to invoke the bean's processRequest () method.

```
...

<html>
  <jsp:useBean id="cart" scope="session"
              class="oracle.jsp.CartBean" />
  <jsp:setProperty name="cart" property="*" />

  <%
      cart.processRequest ();
  %>

...
```

- c) Add the <%@ include %> directive to insert an HTML file (Books.html) in the JSP.

```
...

<html>
  <jsp:useBean id="cart" scope="session"
              class="oracle.jsp.CartBean" />

  <jsp:setProperty name="cart" property="*" />
  <%
      cart.processRequest ();
  %>

  <%@ include file ="Books.html" %>

...
```

## Practice 5-2: Developing a JSP Application Using a JavaBean (continued)

- d) Add the following scriptlet to show the details of the selected book.

```
...

<%@ include file = "Books.html" %>
<div align="center">
  <table width="66%" border="0" height="159" align="left">
    <tr>
      <td width="54%" height="206">
        <p align="left"><font face="Comic Sans MS">Book
Details: </font><br>
        <br/>
        <%
          String res = "";
          res = request.getParameter("item");
          out.print(cart.showDetails(res));
        %>
      </td>
    </tr>
  </table>
</div>
...
```

- e) Add the following scriptlets to retrieve the content from the cart and display the selected books.

```
...

<p align="left">
  <font face="Comic Sans MS">
    You have the following items in your cart:
  </font>
</p>
<p align="left">
  <select name="select" size="10">
    <%
      String[] items = cart.getItems();
      for (int i=0; i<items.length; i++) {
    %>
    <option>
      <% out.print((items[i])); %>
    </option>
    <%
      }
    %>
  </select>
</p>
...
```

### ***Practice 5-2: Developing a JSP Application Using a JavaBean (continued)***

- 5) Save and compile the JSPApplication02 project.
- 6) Run Books.html to test the functionality.
  - a) Right-click Books.html and select **Run**.
  - b) Select any book from the Add/Remove Book drop-down list.
  - c) Click the Add button. Note that the book has been added in the cart.
  - d) Select the same book in the Add/Remove Book drop-down list and click the Remove button. Note that the book has been removed from the cart.
  - e) Add and remove any number of books.
  - f) Select any book in the Add/Remove Book drop-down list and click the Show\_Details button to view the details of the book.
  - g) Close the browser window.

### ***Practice 5-3: Developing a JSP Application Using Custom Tags and Expression Language***

In this practice, you develop your own JSP tags and use it to build a JSP application. You create a simple tag handler class (`SearchBookTag.java`) where you define the functionality of the JSP tag. You also develop a JSP tag library file (`MyTagLibrary.tld`) to reference the Java class in the JSP application.

The `BookDetails` application displays the details of a book in different formats. It extracts the book details from the `BookDetailsBean.java` bean. The different formatting functions are defined in the `Functions.java` file.

- 1) In the Application Navigator, expand **JSPApplication03**.
- 2) Open `bookdetails.oracle.tag > SearchBookTag.java`.
- 3) Modify the code in the `SearchBookTag` class as follows:
  - a) Implement the `doTag()` method to initialize an instance of the `BookDetailsBean` bean. The tag handler pretends to search for a book, and stores the result in a scoped variable.

```
...

public void doTag() throws JspException {
    BookDetailsBean book = new BookDetailsBean( BOOK_TITLE,
                                                BOOK_AUTHOR, BOOK_DESCRIPTION, BOOK_ISBN );
    getJspContext().setAttribute( this.var, book );
}

...
```

- b) Save and compile the file.
- 4) Open `bookdetails.oracle.el > Functions.java`.
- 5) Modify the code in the `Functions` class as follows:
  - a) Implement the `reverse()` and `caps()` methods. The `reverse()` method reverses any given string, and the `caps()` method changes all the characters in a string to uppercase.

```
...

public class Functions {

    public static String reverse( String text ) {
        return new StringBuffer( text ).reverse().toString();
    }

    public static String caps( String text ) {
        return text.toUpperCase();
    }

}
```



### ***Practice 5-3: Developing a JSP Application Using Custom Tags and Expression Language (continued)***

- b) Save and compile the file.
- 6) View the contents of Web Content > WEB-INF > MyTaglibrary.tld.
  - a) Open the tag library in the code editor and analyze the contents of the <tag> and <function> elements in the file. Observe and relate the code that you just added to SearchBookTag.java and Function.java.
- 7) Open BookDetails.jsp. Switch to the Source code view and modify the code as follows:
  - a) Add the <%@ taglib> directive to refer to MyTaglibrary.tld.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ page contentType="text/html; charset=windows-1252"%>

<%@ taglib prefix="ns" uri="/WEB-INF/MyTagLibrary.tld"%>

<html>

...
```

- b) Add the custom tag in the JSP file.

```
...

</br>
    <b><u>Result:</u></b>

    <br></br>

    <ns:searchBook var="book"/>

    <table border="1">

...
```

- c) Add the following expression language in the JSP code to retrieve the book title and apply the formatting functions.

```
...

    <tr>

        <td width="91">Title</td>

        <td width="182">
            ${book.title}
        </td>
```

### ***Practice 5-3: Developing a JSP Application Using Custom Tags and Expression Language (continued)***

```
        <td width="248">
            ${ns:caps(book.title)}
        </td>

        <td width="271">
            ${ns:reverse(book.title)}
        </td>

    </tr>

    ...

```

- d) Save the JSP file.
- 8) Save and compile the JSPApplication03 project.
- 9) Run BookDetails.jsp to test the functionality.
  - a) Right-click BookDetails.jsp and select **Run**.
  - b) Observe the contents in each column to verify the functionality.
  - c) Close the browser when you are done.

## Practices for Lesson 6

In this practice set, you learn to access a database resource in a servlet application by injecting a `javax.sql.DataSource` object's instance variable with the `@Resource` annotation. You modify the `LoginServlet` servlet class to access and query the `Login` table stored in the `FOD` database schema.

You perform the following set of tasks in this practice set:

- Use the `@Resource` annotation in the `LoginServlet` servlet class to access the `Login` table.
- Test the servlet application.
- Use the `@EJB` annotation in a servlet class to reference/inject an Enterprise JavaBean.

## Practice 6-1: Accessing Database Connectivity in a Servlet Application by Implementing Dependency Injection (DI)

In this practice, you modify the `LoginServlet` servlet class to access the `Login` table stored in the `FOD` database schema. You inject the database table in the servlet class by using the `@Resource` annotation.

The primary task of this practice is to modify the database connectivity code to access the database table. The existing logic validates the username and password against predefined database constants (hard-coded in the application). You can make the database connectivity code much simpler in the application by enabling dependency injection.

- 1) In JDeveloper, open `Application_06.jws`.  
**Note:** This opens a skeleton application, the `Servlets` project, for the servlet application components.
  - a) In JDeveloper, select **File > Open**.
  - b) In the Open window, open the `Application_06` directory.
  - c) Double-click the `Application_06.jws` file.
- 2) Modify the code in the `LoginServlet` servlet class to access and query the `Login` table:
  - a) Navigate to `DI_Application01 > Application Sources > oracle.servlets` in the Application Navigator and double-click `LoginServlet.java` to open it in the code editor.
  - b) Add the `@Resource` annotation inside the `LoginServlet` class definition. Use the name attribute of the `@Resource` annotation to specify the data source name. Specify the data source name as `jdbc/fodDS`. Press and hold `ALT + Enter` to add the `javax.annotation.Resource` package that defines the `@Resource` annotation.

**Note:** `jdbc/fodDS` is the data source JNDI name that was automatically created when you configured the database connection in JDeveloper to connect to the `FOD` database schema. This data source is only for testing an application, and not being used in any production environment.

```
...  
    @Resource(name = "jdbc/fodDS")  
...
```

```
import javax.servlet.*;  
import javax.servlet.http.*;  
  
public class LoginServlet extends HttpServlet {  
    @Resource(name = "jdbc/fodDS")  
}
```

- c) Define a `javax.sql.DataSource` instance as the injection field.

## ***Practice 6-1: Accessing Database Connectivity in a Servlet Application by Implementing Dependency Injection (DI) (continued)***

```
...  
@Resource (name = "jdbc/fodDS")  
private javax.sql.DataSource myDs;  
...
```

- d) Define the instance variables for the Connection object.

```
...  
@Resource (name = "jdbc/fodDS")  
private javax.sql.DataSource myDs;  
private Connection con;  
...
```

- e) Modify the code in the `configureConnection()` method to establish a database connection.

```
...  
public void configureConnection() throws SQLException {  
    try{  
        con = myDs.getConnection();  
    }  
    catch (Exception e){  
        System.out.println("Connection failed: "  
                           +e.toString());  
    }  
}  
...
```

- 3) Right-click `LoginServlet.java` and select **Make** to compile the class.
- 4) Run `Login.html` to test the functionality.
  - a) Right-click `Login.html` and select **Run**.
  - b) Enter the username as `sking` and password as `oracle`, and click Login.
- 5) When you are done, close the browser.

## Practice 6-2: Injecting an Enterprise Java Bean by Implementing DI

The primary task of this practice is to inject an Enterprise JavaBean (EJB) in a Java EE application by implementing dependency injection.

In this practice, you modify the `ValidationServlet` servlet class to reference an EJB by incorporating the `@EJB` annotation. The application takes a credit card type and a credit card number, and verifies the credit card to be valid or invalid. The credit card validation logic is implemented in the `CreditCardValidatorBean` session bean. The `ValidationServlet` servlet accepts the credit card details from the `CreditCardValidationForm` HTML form and passes it on to the EJB for validation. Based on the response from the EJB, the servlet displays appropriate message.

- 1) In the `Application_06` workspace, expand the `DI_Application02` project.
- 2) Modify the code in the `ValidationServlet` servlet class to access and query the `CreditCardValidatorBean` session bean:
  - a) Navigate to `DI_Application02 > Application Sources > org.demo.controller` in the Application Navigator and double-click `ValidationServlet.java` to open it in the code editor.
  - b) Add the `@EJB` annotation inside the `ValidationServlet` class definition. Use the `mappedName` attribute of the `@EJB` annotation to specify the session bean that the servlet has to invoke. Press and hold `ALT + Enter` to add the `javax.ejb.EJB` package that defines the `@EJB` annotation.

```
...
public class ValidationServlet extends HttpServlet {

    // add code ....
    @EJB(mappedName = "CreditCardValidatorSessionEJB")
    ...
}
```

**Note:** You will know more about the EJBs and their implementation in the later practices. The `mappedName` attribute refers to the global JNDI name of the target EJB.

- c) Define a `CreditCardValidator` (Enterprise JavaBean's remote interface) instance as the injection field.

```
...
    @EJB(mappedName = "CreditCardValidatorSessionEJB")
    private CreditCardValidator ccEJB;
    ...
}
```

- d) Add the code in the servlet's class `doPost()` method to invoke the Enterprise JavaBean's `validateCreditCard()` business method that implements the logic to validate a credit card.

```
...
public void doPost(HttpServletRequest request,
```

## ***Practice 6-2: Injecting an Enterprise Java Bean by Implementing DI (continued)***

```
        HttpServletResponse response) throws
                                ServletException, IOException {
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    ccType = request.getParameter("card_type");
    ccNum =
Integer.parseInt(request.getParameter("card_number"));
    boolean result = ccEJB.validateCreditCard(ccType, ccNum);
    ...
}
```

- 3) Compile DI\_Application02.
- 4) Run CreditCardValidationForm.html to test the functionality.
  - a) Right-click CreditCardValidationForm.html and select **Run**.
  - b) Enter the card type as AMEX and card number as 123456789, and click **Enter**.
- 5) When you are done, close the browser and close Application\_06. You can also remove it from the IDE if you want.

---

## Practices for Lesson 7

---

In this practice set, you learn to work with session beans by using JDeveloper. You create both the stateless and stateful session beans and execute them by creating a sample test Java client. You perform the following set of tasks in this practice set:

- Creating a stateless session bean and implementing a business method
- Creating a test client to invoke the stateless session bean
- Creating a stateful session bean that calls the stateless session bean by implementing dependency injection. Here you expose a set of methods to store the client's information.
- Creating a test client to invoke the stateful session bean
- Creating an interceptor class to be used in a session bean



## Practice 7-1: Creating a Stateless Session Bean

Use the JDeveloper wizard to create a stateless session bean. Expose a business method in the stateless session bean that takes the user ID and credit card number of a client (as the method parameters) and validates the credit card number against simple logic. After the validation, the method returns an appropriate message to the client.

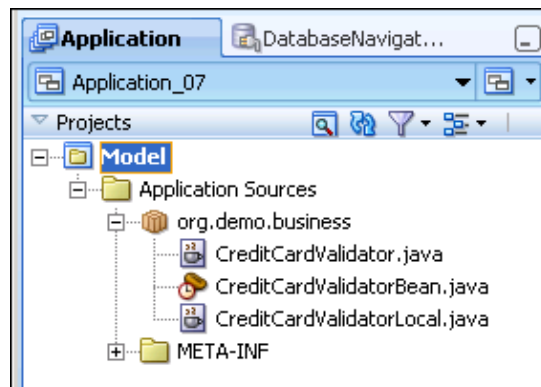
- 1) Open `Application_07`.

**Note:** This opens a skeleton course application, the Model project, for the bean components.

- 2) In JDeveloper, create a stateless session bean called `CreditCardValidator` in the Model project. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Application Navigator	Right-click <b>Application_07</b> > <b>Model</b> . Select <b>New</b> .
b.	New Gallery	Categories: <b>Business Tier</b> > <b>EJB</b> Items: <b>Session Bean</b> Click <b>OK</b> .
c.	Create Session Bean - Step 2 of 5	EJB Name: <code>CreditCardValidator</code> Session Type: <code>Stateless</code> (verify) Transaction Type: <code>Container</code> (verify) Mapped Name: <code>CreditCardValidatorSessionEJB</code> Click <b>Next</b> .
d.	Create Session Bean - Step 3 of 5	Bean Class: <code>org.demo.business.CreditCardValidatorBean</code> Click <b>Next</b> .
e.	Create Session Bean - Step 4 of 5	Implement a Remote Interface: <code>Selected</code> (verify) Implement a Local Interface: <code>Selected</code> (verify) Click <b>Finish</b> .

- 3) The Application Navigator should look like the following:



## **Practice 7-1: Creating a Stateless Session Bean (continued)**

- 4) In `CreditCardValidatorBean.java` (displayed in the source code editor pane of JDeveloper), define a public method—`validateCC()`—that takes two integer parameters (user ID and credit card number). Implement the `validateCC()` method by coding a simple logic to validate the credit card number against a predefined value. Enter the following code (**shown in bold**) before the no-argument constructor in the `CreditCardValidatorBean.java` class:

```
...
    public CreditCardValidatorBean() {
    }

    public String validateCC(int uid, int ccNum){
        if (uid==10 && ccNum==123456789)
            return "Authentic";
        else
            return "Invalid";
    }
...

```

- 5) The `validateCC()` method should be reflected in the session bean's remote and local interfaces. Execute the following steps to accomplish this task:
- Open `CreditCardValidator.java`.
  - Add the following code in the `CreditCardValidator.java` class (session bean's remote interface) to define the `validateCC()` business method:

```
...
@Remote
public interface CreditCardValidator {
    public String validateCC(int uid, int ccNum);
}

```

- Open `CreditCardValidatorLocal.java` (the session bean's local interface) and define the `validateCC()` business method:

```
...
@Local
public interface CreditCardValidatorLocal {
    public String validateCC(int uid, int ccNum);
}

```

- 6) Save and compile the session bean:

## **Practice 7-2: Creating a Sample Test Java Client for the CreditCardValidator Stateless Session Bean**

In this practice, you generate a Java client for the CreditCardValidator bean.

- 1) Use JDeveloper to generate a sample Java client for the CreditCardValidator bean:
  - a) In the Applications Navigator pane, right-click CreditCardValidatorBean.java and select **New Sample Java Client**.
  - b) In the Create Sample Java Client window, specify the following details and click **OK**.

Client Project: **Model.jpr**

Client Class Name: org.demo.client.CreditCardValidatorClient

Application Server Connection: **IntegratedWLSConnection**.

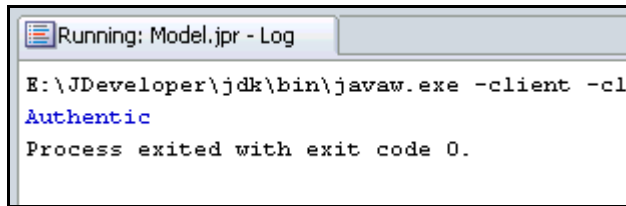
- 2) Add the following code in the main() method of the CreditCardValidatorClient class (the test client's class) to set the parameters and execute the bean's business method:

```
...
// Call any of the Remote methods below to access the EJB
CreditCardValidator creditCardValidator = (CreditCardValidator) ...
System.out.println(creditCardValidator.validateCC(10,
123456789));
    } catch (Exception ex) {
...

```

- 3) Save and compile the test client.
- 4) Execute the CreditCardValidator session bean in the integrated WebLogic Server instance in JDeveloper:
  - a) In the Applications Navigator pane, right-click CreditCardValidatorBean.java and select **Run**.
  - b) Observe the Running: DefaultServer – Log tabbed page and verify that the WebLogic Server instance has been initialized, and the application has deployed successfully.
- 5) Execute the CreditCardValidatorClient test client to test the session bean.
  - a) In the Applications Navigator pane, navigate to Model > Application Sources > org.demo.client node, right-click CreditCardValidatorClient.java, and select Run.
  - b) View the result of executing the test client on Running: Model.jpr - Log.

## ***Practice 7-2: Creating a Sample Test Java Client for the CreditCardValidator Stateless Session Bean (continued)***



```
Running: Model.jpr - Log
E:\JDeveloper\jdk\bin\javaw.exe -client -cl
Authentic
Process exited with exit code 0.
```

**Note:** Try to execute the test client by changing the values in the parameters field of the validateCC() method. For example:

```
...
// Call any of the Remote methods below to access the EJB
// creditCardValidator.validateCC( uid, ccNum );
System.out.println(creditCardValidator.validateCC(10,
100000000));
} catch (Exception ex) {
...

```

### Practice 7-3: Calling a Stateless Session Bean in a Stateful Session Bean

In this practice, you create an instance of the `CreditCardValidator` stateless session bean in the `ShoppingCart` stateful session bean by implementing dependency injection. You define instance variables to store the user ID and credit card number in the `ShoppingCart` bean and also generate the accessor methods. You then define a business method—`generateBill()`—within the `ShoppingCart` bean that invokes the `CreditCardValidator` bean and returns an appropriate message to the client.

- 1) Create a stateful session bean called `ShoppingCart` in the `Model` project. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Application Navigator	Right-click <b>Application_07 &gt; Model</b> . Select <b>New</b> .
b.	New Gallery	Categories: <b>Business Tier &gt; EJB</b> Items: <b>Session Bean</b> Click <b>OK</b> .
c.	Create Session Bean - Step 2 of 5	EJB Name: <code>ShoppingCart</code> Session Type: <code>Stateful</code> (verify) Transaction Type: <code>Container</code> (verify) Mapped Name: <code>ShoppingCartSessionEJB</code> Click <b>Next</b> .
d.	Create Session Bean - Step 3 of 5	Bean Class: <code>org.demo.business.ShoppingCartBean</code> Click <b>Next</b> .
e.	Create Session Bean - Step 4 of 5	Implement a Remote Interface: <code>Selected</code> (verify) Implement a Local Interface: <code>Selected</code> (verify) Click <b>Finish</b> .

- 2) Open `ShoppingCartBean.java` and add the following set of code in the `ShoppingCart` session bean to call and invoke the `CreditCardValidator` stateless session bean, by implementing dependency injection:

- a) Declare the `@EJB` annotation in the `ShoppingCart` session bean (`ShoppingCartBean.java`) to obtain a reference of the `CreditCardValidator` bean. Also define an instance variable of the `CreditCardValidator` bean in `ShoppingCartBean.java`.

**Note:** Import the `javax.ejb.EJB` package to support the `@EJB` annotation.

```
...
@Stateful(name = "ShoppingCart", mappedName =
           "ShoppingCartSessionEJB")
@Remote
@Local
```

### Practice 7-3: Calling a Stateless Session Bean in a Stateful Session Bean (continued)

```
public class ShoppingCartBean implements ShoppingCart,
ShoppingCartLocal {

    @EJB
    private CreditCardValidator ccv;

    ...
}
```

- b) Define two instance variables in `ShoppingCartBean.java` to store the user ID and credit card number of the client:

```
private int uid;
private int ccNum;
```

- c) Generate the accessors methods for the `uid` and `ccNum` instance variables. To perform this task, right-click the source code editor of JDeveloper (inside `ShoppingCartBean.java`) and select **Generate Accessors**.

In the Generate Accessors window, select the `uid` and `ccNum` check boxes and click **OK**.

**Note:** Do not generate accessors for the `ccv` instance variable.

- d) Define and implement the following business method (`generateBill()`) in `ShoppingCartBean.java` that invokes the `CreditCardValidator` bean to validate the credit card number. Based on the validation, the method returns an appropriate message to the client.

```
...

public ShoppingCartBean() {
}

public String generateBill() {
    String res = ccv.validateCC(getUid(), getCcNum());
    if (res.equals("Authentic"))
        return "Order Confirmed";
    else
        return "Invalid Transaction";
}

...
```

- 3) The `generateBill()`, `setUid()`, and `setCcNum()` methods should be reflected in the session bean's remote and local interfaces. Execute the following steps to accomplish this task:
- Open `ShoppingCart.java` in the source code editor.
  - Add the following code in the `ShoppingCart.java` class (session bean's remote interface) to define the `generateBill()`, `setUid()`, and `setCcNum()` methods:

### ***Practice 7-3: Calling a Stateless Session Bean in a Stateful Session Bean (continued)***

```
...  
@Remote  
public interface ShoppingCart {  
    public void setUid(int uid);  
    public void setCcNum(int ccNum);  
    public String generateBill();  
}
```

- c) Add the following code in the `ShoppingCartLocal.java` class (session bean's local interface) to define the `generateBill()`, `setUid()`, and `setCcNum()` methods:

```
...  
@Local  
public interface ShoppingCartLocal {  
    public void setUid(int uid);  
    public void setCcNum(int ccNum);  
    public String generateBill();  
}
```

- 4) Save and compile the `Model` project.

## ***Practice 7-4: Creating a Sample Test Java Client for the ShoppingCart Stateful Session Bean***

In this practice, you generate a Java client for the ShoppingCart bean.

- 1) Use JDeveloper to generate a sample Java client for the ShoppingCart bean:
  - a) In the Navigator, right-click ShoppingCartBean.java and select **New Sample Java Client**.
  - b) In the Create Sample Java Client window, specify the following details and click **OK**.

Client Project: Model.jpr

Client Class Name: org.demo.**client**.ShoppingCartClient

Application Server Connection: IntegratedWLSConnection.

- 2) Add the following code in the main() method of the ShoppingCartClient class (the test client's class) to set the parameters and execute the bean's business method:

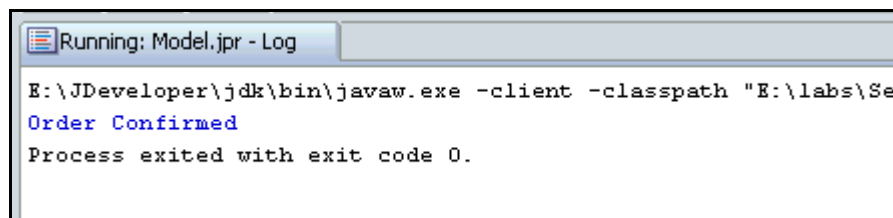
```
...
    ShoppingCart shoppingCart =
        (ShoppingCart) context.lookup( ... );

    shoppingCart.setUId(10);
    shoppingCart.setCcNum(123456789);
    System.out.println( shoppingCart.generateBill( ) );

} catch (Exception ex) {
...

```

- 3) Save and compile the test client.
- 4) Execute the ShoppingCartBean session bean in the integrated WebLogic Server instance in JDeveloper:
  - a) In the Applications Navigator pane, right-click ShoppingCartBean.java and select **Run**.
  - b) Observe the Running: DefaultServer - Log tabbed page and verify that the WebLogic Server instance has been initialized, and the application has deployed successfully.
- 5) Execute the ShoppingCartClient test client to test the session bean.
- 6) View the result of executing the test client on the Running: Model.jpr - Log tabbed page.





### ***Practice 7-4: Creating a Sample Test Java Client for the ShoppingCart Stateful Session Bean (continued)***

- 7) Execute the test client again after changing the values in the parameters field of the `setCcNum()` method. For example:

```
...  
shoppingCart.setUid(10);  
shoppingCart.setCcNum(123878789);  
System.out.println( shoppingCart.generateBill( ) );  
...
```

## Practice 7-5: Using Interceptors in the ShoppingCart Session Bean

Interceptors are used to intercept either a business method or a life-cycle event. An interceptor that intercepts a business method is typically called an `AroundInvoke` method because it can be defined by annotating the method with an `@AroundInvoke` annotation.

In this practice, you develop an external interceptor class that defines a method, which can be used to determine the time that a method takes to execute. You can then use the interceptor class in any of the session beans to find out the exact time a bean's method take to execute.

- 1) Create the `BeanMethodProfile` interceptor class that defines the `profile()` method. The `profile()` method calculates the time that a session bean's method takes to execute (if the interceptor class is used in that session bean). Use the following details to create the Java class:

Step	Screen/Page Description	Choices or Values
a.	Application Navigator	Right-click <b>Application_07 &gt; Model</b> . Select <b>New</b> .
b.	New Gallery	Categories: <b>General &gt; Java</b> Items: <b>Java Class</b> Click <b>OK</b> .
c.	Create Java Class	Name: <code>BeanMethodProfile</code> Package: <code>org.demo.interceptors</code> Click <b>OK</b> .

- 2) Add the following packages in the `BeanMethodProfile` class:

```
import javax.interceptor.AroundInvoke;  
import javax.interceptor.InvocationContext;
```

**Note:** If the IDE indicates it cannot find the classes that you need to add the Java EE 1.5 API library to the Model project to support these packages, execute the following set of tasks to add the library:

- a) Right-click Model and select **Project Properties** from the shortcut menu.
  - b) In the Project Properties window, select **Libraries and Classpath** in the left pane of the window. You can see classpath entries and the libraries description of the default project libraries in the right pane.
  - c) Click the Add Library button. Select **Java EE 1.5 API** from the Add Library window and click **OK**. The library is added to the Model project.
- 3) Define and implement the `profile()` method in the `BeanMethodProfile` class.

```
import javax.interceptor.AroundInvoke;  
import javax.interceptor.InvocationContext;
```

## Practice 7-5: Using Interceptors in the ShoppingCart Session Bean (continued)

```
public class BeanMethodProfile {  
  
    ...  
  
    private Object profile(InvocationContext invCtx)  
                           throws Exception {  
  
        long time1 = System.nanoTime();  
        Object result = invCtx.proceed();  
        long time2 = System.nanoTime();  
        System.out.println(invCtx.getMethod().getName() + " took: " +  
                           ((time2 - time1)/ 1000.0) + " nano seconds.");  
        return result;  
    }  
  
    ...  
}
```

- 4) Add the `@AroundInvoke` annotation to the `profile()` method.

```
public class BeanMethodProfile {  
  
    ...  
  
    @AroundInvoke  
    private Object profile(InvocationContext invCtx)  
                           throws Exception {  
  
        long time1 = System.nanoTime();  
        Object result = invCtx.proceed();  
        long time2 = System.nanoTime();  
        System.out.println(invCtx.getMethod().getName() + " took: " +  
                           ((time2 - time1)/ 1000.0) + " nano seconds.");  
        return result;  
    }  
  
    ...  
}
```

- 5) Save and compile the `BeanMethodProfile` Java class.
- 6) Add the `BeanMethodProfile` interceptor class to the `ShoppingCartBean` session bean.
- a) Open `ShoppingCartBean.java`.
  - b) Add the following code (in bold) in the `ShoppingCartBean` class to implement the `BeanMethodProfile` interceptor class.

```
...  
  
import javax.interceptor.Interceptors;
```

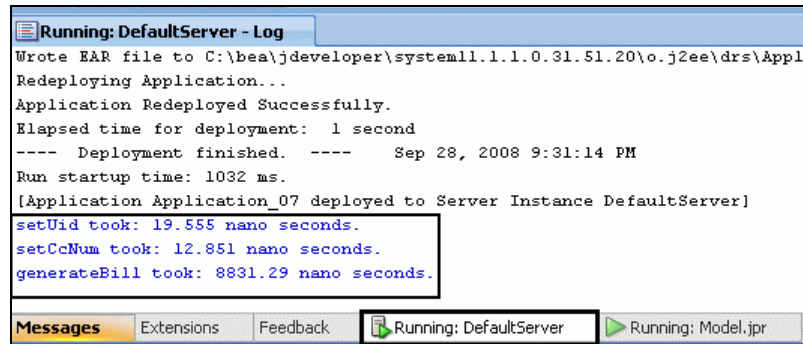
## Practice 7-5: Using Interceptors in the ShoppingCart Session Bean (continued)

```
import org.demo.interceptors.BeanMethodProfile;

@Stateful(name = "ShoppingCart", mappedName =
"ShoppingCartSessionEJB")
@Remote
@Local
@Interceptors(BeanMethodProfile.class)

...
```

- 7) Run the ShoppingCartBean session bean in the integrated WebLogic Server instance in JDeveloper.
- 8) Run the ShoppingCartClient test client to test the ShoppingCartBean session bean.
- 9) View the result of executing the test client in **Running: Model.jpr - Log**. To view the execution time of the ShoppingCartBean session bean's methods, click the **Running: DefaultServer** tab in the Log tabbed page.



## Practices for Lesson 8

In this practice set, you use JPA entity beans to create an application model layer. You create several beans and a Session bean that provides client access. You also use a JDeveloper wizard to create a simple Java client to test your Session bean.

You perform the following set of tasks in this practice set:

- Create a simple entity bean by coding the bean.
- Use the JDeveloper wizards to create a set of entity beans.
- Create and manage a Session bean that provides client access to the entity beans.
- Creating a test client to invoke the session bean

## ***Practice 8-1: Coding a Simple Entity Bean***

In this practice, you create a simple entity bean in a new Application.

- 1) Create an Application.
  - a) Invoke the **New Application** Wizard.
  - b) Name the Application **Application\_08**.
  - c) Set the Application Package Prefix to **oracle**.
  - d) Select **Java EE Web Application** as the Application Template.
  - e) Rename the ViewController project **UI**.
  - f) The Default Package should be **oracle.view**.
  - g) Ensure that the name of the Model project is **Model**.
  - h) Ensure that the Default Package is **oracle.model**.
  - i) Ensure that the EJB Version is **Enterprise JavaBeans 3.0**.
  - j) Click **Finish**.
- 2) Create an Offline Database based on the FOD schema.
  - a) Invoke the **New Gallery** from the Model project.
  - b) Select **Database Tier > Offline Database Objects > Copy Database Objects to a Project**.
  - c) Create a new connection named FOD based on FOD / fusion. (Click the green plus symbol next to Connection.)
  - d) Name the Offline Database FOD.
  - e) Select **PERSONS** and **PERSON\_SEQ** from the list of available objects. (Click Query to retrieve the list of available objects.)
  - f) Click **Next**, then **Finish** to create the Offline Database.
- 3) Next, create a Persistence Unit.
  - a) Invoke the **New Gallery** on the Model project
  - b) Select **Business Tier > TopLink/JPA > JPA Persistence Unit**.
  - c) Name the Persistence Unit FOD, and click **OK** to create the Persistence Unit.
- 4) Set the Development Database to the Offline Database you just created (FOD).
  - a) Open `persistence.xml` in the visual editor.
  - b) Select **FOD** in the Structure window.
  - c) Select **Connection** in the visual editor.
  - d) Use the drop-down list to select **FOD (Oracle)** for the Development Database.
- 5) Create a Java class named `Person` in the `oracle.model` package. It should be `public` with a default constructor.

## Practice 8-1: Coding a Simple Entity Bean (continued)

- 6) Add the Entity annotation.
  - a) `@Entity(name="Person")`
  - b) Add the import of `javax.persistence.Entity`.
  - c) Change the Person class to implement `Serializable` and include the import.
- 7) Change the constructor to `protected`.
- 8) Add private class variables and column annotations for the following:

Column	Name	Type
ID	id	private Long
FIRST_NAME	firstName	private String
LAST_NAME	lastName	private String
EMAIL	email	private String
PHONE_NUMBER	phoneNumber	private String

**Note:** You will need to include the import for `javax.persistence.Column`.

- 9) Add the `@Table` annotation because the source table for this Person entity is `PERSONS`.
- 10) Identify the `id` attribute as the `Id` for the entity.
  - a) Add the `@Id` annotation to the attribute.
  - b) Add `nullable=false` to the `@Column` annotation.
    - i) Add annotation to use a generated value for ID as follows:  
GeneratedValue using the generator named `PER_SEQ` with a strategy of `GenerationType.SEQUENCE`.  
SequenceGenerator using `sequenceName PERSON_SEQ`.
    - ii) You will need to import:  
`javax.persistence.GenerationType`  
`javax.persistence.GeneratedValue`  
`javax.persistence.SequenceGenerator`

The class should look like the following:

```
@Entity(name="Person")
@Table(name="PERSONS")
public class Person implements Serializable{

    @Id
    @Column(name="PERSON_ID", nullable=false)
    @GeneratedValue(generator = "PER_SEQ",
                    strategy = GenerationType.SEQUENCE)
    @SequenceGenerator(name="PER_SEQ",
                      sequenceName = "PERSON_SEQ")
    private Long id;
    @Column(name="FIRST_NAME")
    private String firstName;
```

## Practice 8-1: Coding a Simple Entity Bean (continued)

```
@Column(name="LAST_NAME")
private String lastName;
@Column(name="EMAIL")
private String email;
@Column(name="PHONE_NUMBER")
private String phoneNumber;

protected Person() {
}
}
```

11) Add a generic `findAll` query as follows:

a) Add a `@NamedQueries` annotation like:

```
@NamedQueries({ })
```

b) Inside the braces add annotation for the `NamedQuery`:

```
@NamedQueries({
    @NamedQuery(name = "Person.findAll",
        query = "select o from Person o")
})
```

12) The class should now look like the following:

```
package oracle.model;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity(name="Person")
@NamedQueries({
    @NamedQuery(name="Person.findAll", query = "select o from
Person o")
})
@Table(name = "PERSONS")

public class Person implements Serializable{

    @Id
    @Column(name="PERSON_ID", nullable=false)
    @GeneratedValue(generator = "PER_SEQ",
        strategy = GenerationType.SEQUENCE)
```



### **Practice 8-1: Coding a Simple Entity Bean (continued)**

```
@SequenceGenerator(name="PER_SEQ",
                    sequenceName = "PERSON_SEQ")
private Long id;
@Column(name="FIRST_NAME")
private String firstName;
@Column(name="LAST_NAME")
private String lastName;
@Column(name="EMAIL")
private String email;
@Column(name="PHONE_NUMBER")
private String phoneNumber;

protected Person() {
}
}
```

13) Add accessors for each of the attributes in the Person Entity.

- a) Right-click one of the attributes and select **Generate Accessors**.
- b) Click **Select All** in the Generate Accessors dialog box.
- c) Click **OK**.

14) Compile and save your work

You have now created an entity bean completely from scratch. In the next section, you use JDeveloper wizards to create the remaining entity beans.

## ***Practice 8-2: Using the JDeveloper Entity Bean Wizard***

In this practice, you create two entity beans using the Entity Bean wizard.

- 1) Invoke the **New Gallery** on the Model project.
- 2) Select **Business Tier > EJB > Entities from Tables**.
- 3) In the Create Entities from Tables Wizard:
  - a) Select **FOD** as the Persistence Unit.
  - b) Select **Online Database Connection** and choose **FOD** as the connection.
  - c) On the Select Tables page, click **Query** to populate the Available list.
  - d) Select **Categories** and **Products** and move them to the Selected list.
  - e) If it is not already, set the package name to `oracle.model`.
  - f) In Entity Details, change the names of both the entities to singular (Products to **Product**, Categories to **Category**).
  - g) Click **Finish** to create the entities.
- 4) Examine `persistence.xml`.
  - a) Open `persistence.xml`.
  - b) In the Structure window, expand **Mapped Classes > oracle.model** and notice the Category and Product entities are now mapped.
- 5) Examine `Product.java` and `Category.java` to see what was created for you.

### **Practice 8-3: Customizing Entity Beans**

In this practice, you customize the Product and Category entity beans you created with the Entity Bean wizard.

The wizard creates attributes for each column in the source table. Your application may not need every attribute. It also creates set and get methods for each attribute that you may or may not need. You will also rename some attributes to better fit the business use of the attributes.

- 1) Open `Product.java` in the visual editor.
- 2) Remove the following attributes and related accessors:

```
attributeCategory
createdBy
creationDate
lastUpdatedBy
lastUpdateDate
objectVersionId
```

- 3) Remove the same attributes from the constructor. The attributes you removed will be underlined in red to show there is a problem. Remove them from the method signature as well. You will notice the removed attributes will be in a light gray font.
- 4) Change the no-arg constructor to protected.
- 5) Right-click anywhere in the editor and select **Make**.
- 6) Add annotation to use a generated value for Id as follows:
  - a) `GeneratedValue` using the generator named `PROD_SEQ` with a strategy of `GenerationType.SEQUENCE` (You will need to import `javax.persistence.GeneratedValue` and `javax.persistence.GenerationType`.)
  - b) `SequenceGenerator` named `PROD_SEQ` using the `sequenceName` `PRODUCT_SEQ`. (You will need to import `javax.persistence.SequenceGenerator`.)

The code should look like the following:

```
@Id
@Column(name="PRODUCT_ID", nullable = false)
@GeneratedValue(generator = "PROD_SEQ",
                  strategy = GenerationType.SEQUENCE)
@SequenceGenerator(name="PROD_SEQ",
                  sequenceName = "PRODUCT_SEQ")
private Long productId;
```

- 7) Now that you have Id to be generated from a sequence, remove the `setProductId()` method and `productId` from the constructor. Remove it from the constructor argument list and the method.
- 8) Make the class to check for errors.

### Practice 8-3: Customizing Entity Beans (continued)

9) Open `Category.java` in the visual editor.

10) Remove the following attributes and related accessors:

```
createdBy  
creationDate  
lastUpdatedBy  
lastUpdateDate  
objectVersionId
```

11) Remove the same attributes from the constructor. The attributes you removed will be underlined in red to show there is a problem. Remove them from the method signature as well. You will notice the removed attributes will be in a light gray font.

12) Change the no-arg constructor to **protected**.

The attributes defined in this Entity may be accessed from any number of client applications. It is important to name the attributes so that they are readily recognizable for what they represent. In some cases, the generated names may not reflect their content precisely, especially when they are created as part of a relationship.

13) Use Refactor to rename the **category** attribute that is in the `PARENT_CATEGORY_ID` relationship to **parent**. Right-click the **category** attribute and select **Refactor > Rename**. When asked to rename the accessors, click **Yes**.

When client applications access this attribute, they will use the notation `category.parent`, which better describes the content and purpose of this attribute.

14) Examine the accessors to see that the names have changed. The accessor methods should look like the following:

```
public Category getParent() {  
    return parent;  
}  
  
public void setParent(Category category) {  
    this.parent = category;  
}
```

15) Notice also that the `addCategory` and `removeCategory` methods have also been refactored to reflect the change to the method name.

16) The constructor is also changed to reflect the new attribute name. The constructor should now look like the following:

```
public Category(String categoryDescription,  
                Long categoryId,  
                String categoryLockedFlag,  
                String categoryName,  
                Category category) {  
    this.categoryDescription = categoryDescription;  
    this.categoryId = categoryId;
```

### Practice 8-3: Customizing Entity Beans (continued)

```
        this.categoryLockedFlag = categoryLockedFlag;
        this.categoryName = categoryName;
        this.parent = category;
    }
```

- 17) Because you refactored category to parent, you need to change the OneToMany mappedBy from category to parent. The code should now look like the following:

```
@OneToMany(mappedBy = "parent")
private List<Category> categoryList;
```

- 18) Refactor the following attributes:

```
private List<Category> categoryList; to
private List<Category> children;
```

This will make client code more understandable because to access child categories now, the access will be category.children.

- 19) The refactor will also modify the get, set, addCategory, and removeCategory methods to reflect the new name. The methods should look like the following:

```
public List<Category> getChildren() {
    return children;
}

public void setChildren(List<Category> categoryList) {
    this.children = categoryList;
}

public Category addCategory(Category category) {
    getChildren().add(category);
    category.setParent(this);
    return category;
}

public Category removeCategory(Category category) {
    getChildren().remove(category);
    category.setParent(null);
    return category;
}
```

- 20) Right-click anywhere in the editor and select **Make**.

### ***Practice 8-4: Creating and Testing a Session EJB***

In this practice, you create a Session EJB to provide client access to the entity beans you just created.

- 1) Invoke the **New Gallery** on the Model project
- 2) Select **Business Tier > EJB > Session Bean**.
- 3) In the Create Session Bean wizard:
  - a) Set the EJB Name to **SessionEJB08**.
  - b) Ensure that **Generate Session Façade Methods** is selected.
  - c) The Entity Implementation should be **JPA Entities**.
  - d) On the Session Façade page, select all the methods *except* those that have `findAllByRange` and `RemovePerson`.
  - e) Accept the Bean class name, but change the package to `oracle.services` such as `oracle.services.SessionEJB08Bean`
  - f) Implement both **Remote** and **Local** Interfaces.
  - g) Click **Finish** to create the Session EJB.
- 4) Open `SessionEJB08Bean.java` in the visual editor and examine the generated methods.

Now that you have a Session EJB, you can create a simple Java client to test your model.

- 5) Right-click `SessionEJB08Bean.java` in the Application Navigator.
- 6) Select **New Sample Java Client**.
- 7) Accept the default settings and click **OK** to create the java class.
- 8) The class opens in the visual editor.
- 9) Examine the contents:
  - a) Notice that each of the queries is called with a set of `System.out.println` statements to return the row values. Those are nested in for loops that will display all the rows in the entity.
  - b) Notice the `getInitialContext` method, which provides the connection details to the WLS server.
- 10) Test the `SessionEJB08` and entity beans by performing the following steps:
  - a) Right-click `SessionEJB08Bean.java` and select **Run**.
  - b) After you see the message indicating that the application has been deployed successfully to the Server Instance Default Server, right-click `SessionEJB08Client.java` and select **Run**. You will see a list of the selected data in the log window.

### ***Practice 8-4: Creating and Testing a Session EJB (continued)***

- 11) Stop the Application Server instance (Run > Terminate > Default Server).
- 12) You can close and remove the Application from the IDE when you are done.

## Practices for Lesson 9

As you design and develop an application, you will discover the need for queries specific to your application. For example, you may want a query for finding a person by ID and by Name. You may also need to find a product by ID or Name. In this practice, you create and test several Named Queries.



## Practice 9-1: Adding Named Queries to Person

In this practice, you add two Named Queries to the Person Entity, and test them.

- 1) In JDeveloper, open `Application_09.jws`.
- 2) Open `Person.java` in the visual editor.
- 3) Add a Named Query named `Person.findById` that selects a row based on `Id`.
  - a) Within the `NamedQueries` annotation and after the first `NamedQuery`, add a `NamedQuery` named `Person.findById`.
  - b) The query should select from `Person` where `o.id` is equal to an `:id` parameter.
  - c) The code should look like the following:

```
@NamedQuery(name = "Person.findById",  
            query = "select o from Person o where o.id = :id")
```

- 4) Add another `NamedQuery` named `Person.findByName` that selects the `Person` by `lastName`. The completed code should look like the following:

```
@NamedQueries({  
    @NamedQuery(name = "Person.findAll",  
                query = "select o from Person o"),  
    @NamedQuery(name = "Person.findById",  
                query = "select o from Person o  
                        where o.id = :id"),  
    @NamedQuery(name = "Person.findByName",  
                query = "select o from Person o  
                        where o.lastName like :lastName")  
})
```

- 5) Add the new `NamedQueries` to the `SessionEJB`.
  - a) Right-click `SessionEJB09Bean` and select **Edit Session Façade**.
  - b) Expand **Person** in `Façade Options`.
  - c) Select `Person.findById` and `Person.findByName`. (Leave the original queries selected.)
  - d) Click **OK** to add them to the `Façade`.
- 6) Create a new `Sample Java Test Client`. Accept the default name.
- 7) JDeveloper opens the new test client in the visual editor. Notice that there are two red marks or indicators in the margin at the right of the editor. Click the first one and the editor highlights the line of code where there is a problem.
- 8) Notice that the generated code has a notation where the query is expecting a parameter.
- 9) Add a parameter to the `queryPersonFindById` query. You can use any valid `PersonId` (100–129).

### ***Practice 9-1: Adding Named Queries to Person (continued)***

10) Add a String parameter to `queryPersonFindByLastName`. You can use a wildcard such as “A%” to return all Persons whose last name begins with A.

11) Test the queries by running `SessionEJB09Bean`, and then the new Java client.

**Hint:** It will be easier to see your test results if you comment out the other queries in the test client.

12) If you used 106 and “A%”, the results should look like the following: (results truncated for space)

```
id = 106
firstName = Valli
lastName = Pataballa
email = VPATABAL
phoneNumber = 256.555.0130
id = 105
firstName = David
lastName = Austin
email = DAUSTIN
phoneNumber = 401.555.0129
id = 207
firstName = Jose
lastName = Armstrong
email = JARMSTRONG
phoneNumber = 818.555.0116
```

## Practice 9-2: Adding Named Queries to Product

In this practice, you add two named queries to Product. In your application, you need to query a Product based on the Id and the name.

- 1) Open `Product.java` in the visual editor.
- 2) Add a Named Query named `Product.findById` that selects a row based on Id.
  - a) Within the `NamedQueries` annotation and after the first `NamedQuery`, add a `NamedQuery` named `Product.findById`.
  - b) The query should select from `Product` where the `o.id` is equal to an `:id` parameter.
  - c) The code should look like the following:

```
@NamedQuery(name = "Product.findById",  
            query = "select o from Product o where o.id = :id")
```

- 3) Add another `NamedQuery` that selects the Product by name. The completed code should look like the following:

```
@NamedQueries({  
    @NamedQuery(name = "Product.findAll",  
                query = "select o from Product o"),  
    @NamedQuery(name = "Product.findById",  
                query = "select o from Product o  
                        where o.productId = :id"),  
    @NamedQuery(name = "Product.findByName",  
                query = "select o from Product o  
                        where o.productName like :name")  
})
```

- a) Add the new `NamedQueries` to the `SessionBeanEJB`.
  - 4) Create a new **Sample Java Test Client**. Accept the default name.
  - 5) JDeveloper opens the new test client in the visual editor.
  - 6) Add a parameter to the `queryProductFindById` query. You can use any valid `ProductId` (1–20).
  - 7) Add a String parameter to `queryProductFindByName`. You can use a wildcard such as `"I%"` to return all Products that have a name beginning with `"I."`
  - 8) Test the queries by running `SessionEJB09Bean`, and then the new java client.
- Hint:** It will be easier to see your test results if you comment out the other queries in the test client.
- 9) If you used 1 and `"I%"`, the results should look like the following:

```
additionalInfo = HDMI (High-Definition Multimedia  
Interface) is a lossless, ...  
costPrice = 1499.99  
description = The TH-42PX60U 42-inch Diagonal Plasma HDTV  
gives you deep blacks, ...
```

## ***Practice 9-2: Adding Named Queries to Product (continued)***

```
externalUrl = images/1.jpg
listPrice = 1999.99
minPrice = 1599.99
productId = 1
productName = Plasma HD Television
shippingClassCode = CLASS3
supplierId = 112
warrantyPeriodMonths = 6
category = oracle.model.Category@179d854
additionalInfo = null
costPrice = 35.0
description = The Portable Audio System for iPod ...
externalUrl = images/15.jpg
listPrice = 89.99
minPrice = 55.99
productId = 15
productName = Ipod Speakers
shippingClassCode = CLASS2
supplierId = 106
warrantyPeriodMonths = 6
category = oracle.model.Category@3a835d
additionalInfo = null
costPrice = 200.0
description = Apple iPod - Continuing its tradition ...
externalUrl = images/17.jpg
listPrice = 339.99
minPrice = 299.95
productId = 17
productName = Ipod Video 80Gb
shippingClassCode = CLASS2
...
```

### ***Practice 9-3: Adding Named Queries to Category***

In the application that you are building, you will need a couple of specialized queries on Category. The application needs to be able to find all the root categories. Root categories are defined as those categories that do not have a parent category. You also need a query that returns only those leaf categories, which do have a parent category. These two queries will be used to populate a tree control in the UI.

- 1) Open `Category.java` in the visual editor.
- 2) Add a NamedQuery called `Category.findRoot`, which queries all categories where the parent is null.
- 3) Add a NamedQuery called `Category.findLeaf` which queries only the categories that have parent categories.
- 4) The completed code should look like the following:

```
@NamedQueries({
    @NamedQuery(name = "Category.findAll",
        query = "select o from Category o"),
    @NamedQuery(name = "Category.findRoot",
        query="select o from Category o
            where o.parent is null"),
    @NamedQuery(name = "Category.findLeaf",
        query="select o from Category o
            where o.parent is not null")
})
```

- 5) Add the two new NamedQueries to the Session Facade (`SessionEJB09Bean`).
- 6) Create a **Sample Java Test Client**. (Accept the default name.)
- 7) Comment out all test code except the two new Category queries.
- 8) Run the test. (Remember to run `SessionEJB09Bean` first.)
- 9) The results should look something like the following:

```
categoryDescription = Office Supplies
categoryId = 2
...
parent = null
children = {IndirectList: not instantiated}
categoryDescription = Consumer Electronics
categoryId = 3
...
parent = null
children = {IndirectList: not instantiated}
categoryDescription = Books, Music, and Movies
categoryId = 1
...
parent = null
children = {IndirectList: not instantiated}
categoryDescription = Books
categoryId = 8
```

### ***Practice 9-3: Adding Named Queries to Category (continued)***

```
...
parent = oracle.model.Category@1198ff2
children = {IndirectList: not instantiated}
categoryDescription = DVDs
categoryId = 9
...
parent = oracle.model.Category@1198ff2
children = {IndirectList: not instantiated}
```

10) Notice that the parent in the first query is null.

11) Notice that the parent in the second query returns

`oracle.model.Category@nnnn`. That is because the query returns only a reference to the parent category object.

## Practices for Lesson 10

This practice set enables you to create and deploy a Web service application to Oracle WebLogic Server. You use Oracle JDeveloper 11g IDE as the development tool for developing the Web service applications with the top-down Web service development approach.

In this practice set, you create a credit card validation Web service. It is a simple validation Web service that validates a credit card by using a credit card type and credit card number. The XML artifacts (XSD and WSDL files) have been provided to you. You have to use these artifacts to generate and implement a Web service method. The method takes two parameters (credit card type and credit card number) and validates it with specific business logic, and returns a string value to the caller function. You perform the following set of tasks in this practice set:

- Creating a Web service application by using the top-down Web service development approach
- Testing the Web service application by using JDeveloper

## ***Practice 10-1: Creating the `ValidateCreditCardService` Web Service by Using the Top-Down Web Service Development Approach***

Your task is to create a Web service application to validate a credit card by using the top-down approach. The Web service takes the credit card type and credit card number, and validates it against a simple logic (no database implementation). After validating, it returns an appropriate string message. Perform the following steps:

- 1) Open `Application_10.jws`.
- 2) In the Model project, you see the following XML artifacts (in the Application Sources folder):
  - `request.xsd`
  - `response.xsd`
  - `WSDLDocument.wsdl`
- 3) Edit the `request.xsd` schema in the Source view of the code editor. The schema defines the input parameter of the Web service operation. Edit the schema with the following modification:
  - a) Set the name property of the complex type to `input`. The complex type contains two XML elements. Set the element's name and type attributes as:
    - For the first element: `name="arg0"`, `type="xsd:string"`
    - For the second element: `name="arg1"`, `type="xsd:int"`
  - b) Use the `input` complex type to create an XSD element. Set the name attribute of the XSD element to `request`.

```
<?xml version="1.0" encoding="windows-1252" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:tns="http://www.example.org"
            targetNamespace="http://www.example.org"
            elementFormDefault="qualified">
  <xsd:complexType name="input">
    <xsd:sequence>
      <xsd:element name="arg0" type="xsd:string"/>
      <xsd:element name="arg1" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="request" type="tns:input"/>
</xsd:schema>
```

- c) Save the modification.



## ***Practice 10-1: Creating the ValidateCreditCardService Web Service by Using the Top-Down Web Service Development Approach (continued)***

- 4) Modify the `WSDLDocument.wsdl` file in the Source view of the code editor. The file defines each aspect of the Web service operation. `WSDLDocument.wsdl` incorporates the two XML schemas, one of that you have modified earlier.
  - a) Import the `request.xsd` schema by specifying the namespace property of the `<import>` element as `"http:// www.example.org"`, and the `schemaLocation` property as `"request.xsd"`.

```
...  
  
<wsdl:types>  
    <xsd:schema>  
        <xsd:import namespace="http://www.example.org"  
                    schemaLocation="request.xsd"/>  
    ...
```

- b) Set the first `<message>` element's name property to `"validateCCInput"`.

```
...  
  
<wsdl:types>  
    <xsd:schema>  
        <xsd:import namespace="http://www.example.org"  
                    schemaLocation="request.xsd"/>  
        <xsd:import namespace="http://www.example.org"  
                    schemaLocation="response.xsd"/>  
    </xsd:schema>  
</wsdl:types>  
<wsdl:message name="validateCCInput">  
    <wsdl:part name="parameters" element="tns:request"/>  
</wsdl:message>  
<wsdl:message name="validateCCOutput">  
    <wsdl:part name="parameters" element="tns:response"/>  
</wsdl:message>  
...
```

- c) Set the `<portType>` element's name property to `"ValidateCreditCardService"`.
  - d) Set the `<operation>` element's name property to `"validateCC"`.

```
...  
  
<wsdl:message name="validateCCOutput">
```

## **Practice 10-1: Creating the ValidateCreditCardService Web Service by Using the Top-Down Web Service Development Approach (continued)**

```
        <wsdl:part name="parameters" element="tns:response"/>
    </wsdl:message>
    <wsdl:portType name="ValidateCreditCardService">
        <wsdl:operation name="validateCC">
            <wsdl:input message="tns:validateCCInput"
                xmlns:ns1="http://www.w3.org/2006/05/addressing/wsdl"
                                   ns1:Action="" />
            <wsdl:output message="tns:validateCCOutput"
                xmlns:ns1="http://www.w3.org/2006/05/addressing/wsdl"
                                   ns1:Action="" />
        </wsdl:operation>
    </wsdl:portType>
    ...
```

- e) Modify the `<soap12:address>` element's location property to `http://localhost:7101/contextroot/ValidateCreditCardServiceSoap12HttpPort`.

```
...
<wsdl:service name="ValidateCreditCardService">
    <wsdl:port name="ValidateCreditCardServiceSoap12HttpPort"
        binding="tns:ValidateCreditCardServiceSoapHttp">
        <soap12:address location="http://localhost:7101/
            contextroot/ValidateCreditCardServiceSoap12HttpPort"/>
    </wsdl:port>
</wsdl:service>
...
```

- 5) Because you will be deploying the Web service from JDeveloper, you need to change the deployment setting for the application's Java EE context root in JDeveloper. Execute the following steps to perform the task:
- In the Applications Navigator, right-click the Model project and select **Project Properties**.
  - Select **Java EE Application** in the left pane. In the Java EE Application pane, change the value of the **Java EE Web Context Root** to `contextroot`.
  - Click **OK**.
- 6) Save the modifications in the WSDL file.
- 7) Create the Web Service artifacts by using the "Create Java Web Service from WSDL Wizard." Create a Web service called `ValidateCreditCardService` in the `org.demo.business` package by using the JDeveloper wizard. Use the following details to accomplish this task:

**Practice 10-1: Creating the ValidateCreditCardService Web Service by Using the Top-Down Web Service Development Approach (continued)**

Step	Screen/Page Description	Choices or Values
a.	Applications Navigator	Right-click <b>Application_10 &gt; Model</b> . Select <b>New</b> .
b.	New Gallery	Categories: <b>Business Tier &gt; Web Services</b> Items: <b>Java Web Service from WSDL</b> Click <b>OK</b> .
c.	Create Java Web Service from WSDL - Step 1 of 6	Click <b>Next</b> .
d.	Create Java Web Service from WSDL - Step 2 of 6	WSDL Document URL: <LAB_HOME>/Application_10/Model/src/WSDLDocument.wsdl Click <b>Next</b> .
e.	Create Java Web Service from WSDL - Step 3 of 6	Package Name: org.demo.business Click <b>Finish</b> .

- 8) Expand **Model > Application Sources > org.demo.business**, and open `ValidateCreditCardServiceImpl.java`. You see the `validateCC()` method being defined and not implemented. Add the following code (in bold) in the `validateCC()` method to implement the Web service business logic.

```
...

public class ValidateCreditCardServiceImpl {
    public ValidateCreditCardServiceImpl() {
    }

    @javax.jws.soap.SOAPBinding(parameterStyle =
        javax.jws.soap.SOAPBinding.ParameterStyle.BARE)
    @WebMethod
    @WebResult(name = "response", targetNamespace =
        "http://www.example.org", partName = "parameters")
    public Output validateCC(@WebParam(name = "request",
        partName = "parameters",
        targetNamespace = "http://www.example.org")
        Input parameters) {
        Output res = new Output();
        if(parameters.getArg0().equalsIgnoreCase("AMEX") &&
            parameters.getArg1()==123456789)

            res.setArg0("Valid credit card");
        else
    }
}
```

***Practice 10-1: Creating the ValidateCreditCardService Web Service by Using the Top-Down Web Service Development Approach (continued)***

```
        res.setArg0("Invalid credit card type or  
                    credit card number");  
    return res;  
}  
}  
...
```

- 9) Save and compile the Model project.

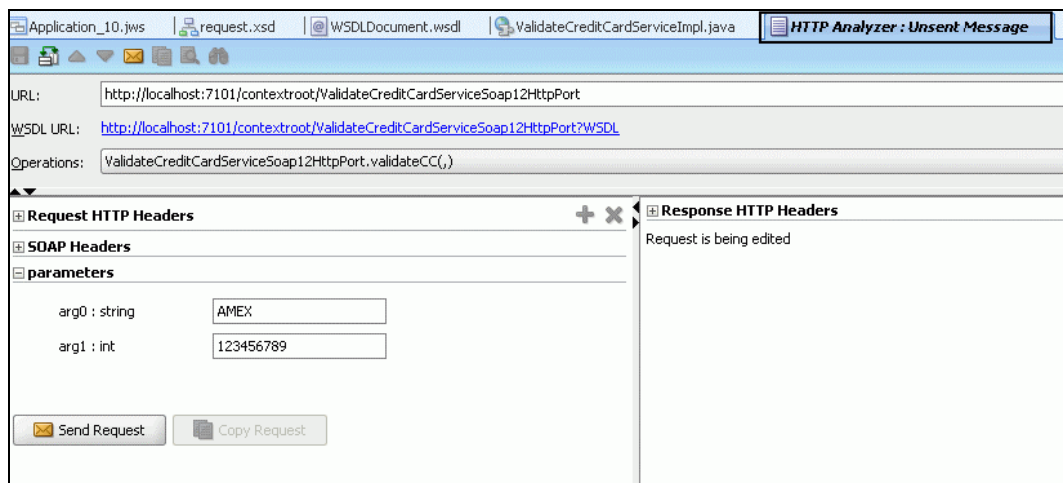
## Practice 10-2: Testing the Web Service by using JDeveloper

In this practice, you test the Web service that you created in your previous practice by using Oracle JDeveloper 11g.

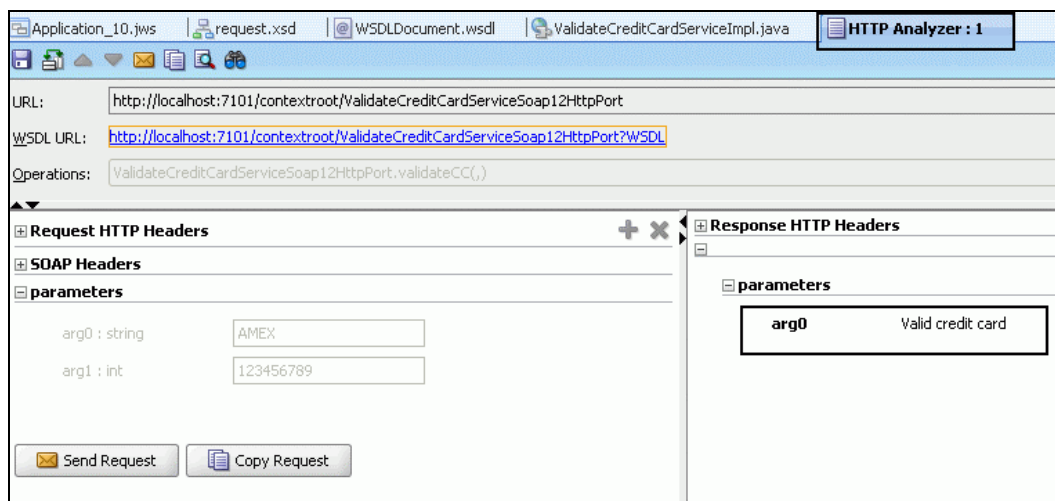
- 1) Right-click `org.demo.business > ValidateCreditCardService` and select **Test Web Service**.
- 2) You see the HTTP Analyzer: Unsent Message pane being displayed. Add the following values to the respective parameters and click the **Send Request** button.

arg0:string – AMEX

arg1:int – 123456789



- 3) In the right pane, you see the value of the arg0 response parameter being displayed as “Valid credit card.”



- 4) Click **Copy Request** in the left pane and modify the parameters to something like VISA and 12345678989, and then click Send Request. You see “Invalid credit card type or credit card number” in the response pane.
- 5) Close the application and remove it from the IDE when you are done.

---

## Practices for Lesson 11

---

In an earlier practice, you completed the FOD Storefront model with entities for Products and Categories. Next you move on to building the user interface for your application. The first part, building the UI, is to create a managed bean that will serve as the interface to the Session Façade. This behavior of this bean is similar to the Sample Java Test Client you built in a previous practice. After you build the managed bean, you create a page that shows a tree of Categories and a list of Products.

## Practice 11-1: Creating the ProductBrowsing Managed Bean

In this practice, you create a managed bean that provides an interface between the client UI and the Session Façade.

- 1) Open `Application_11.jws`.
- 2) In the **UI** project, create a new Java class named `ProductBrowsingBean`, in the `oracle.ui.backing` package. Make it `public` with a default constructor. The code should look something like the following:

```
package oracle.ui.backing;

public class ProductBrowsingBean {
    public ProductBrowsingBean() {
    }
}
```

The first thing you need in the bean is a method that will find and return the **SessionBean**. Remember that the `SessionBean` is the object that you use to get access to all the underlying entity beans and methods.

- 3) Add the following code to the bean:

```
public SessionEJB11Local getSessionBean()
    throws NamingException {
    InitialContext ic = new InitialContext();
    Object lookupObject =
        ic.lookup("java:comp/env/ejb/local/SessionEJB11");
    return (SessionEJB11Local) lookupObject;
}
```

- 4) Import the following two classes:

```
javax.naming.InitialContext
javax.naming.NamingException
```

- 5) You will see that `SessionEJB11Local` is not found in the current active project because it is defined in the Model project. Add a Model project dependency to this project:
  - a) Open the **UI** project properties.
  - b) Select **Dependencies**.
  - c) Click the **Edit** icon (pencil).
  - d) Expand **Model.jpr**, select **Build Output**, and click **OK**.
  - e) While you have the project properties open, add the following library in **Libraries and Classpath**.

```
Trinidad Runtime 11
```

- f) Click **OK**.
- g) Select **JSP Tag Libraries** and add:

```
ADF Faces Components 11
```

## ***Practice 11-1: Creating the ProductBrowsing Managed Bean (continued)***

- h) Click **OK** and close the Project Properties.
- 6) Press and hold Alt + Enter to add the import for `oracle.services.SessionEJB11Local`.
- 7) Before your code can lookup the session bean, it must be registered with `Web.xml`.
  - a) Open **Web Content** > **WEB-INF** > `web.xml`.
  - b) On the **Overview** page, click **References**.
  - c) Expand **EJB References**
  - d) Click the EJB References **Plus** icon at the right of the window.
  - e) Complete the entry as follows:

Field Name	Value
EJB Name	<code>ejb/local/SessionEJB11</code>
Interface Type	Local
EJB Type	Session
Home Interface	(leave blank)
Local/Remote Interface	<code>oracle.services.SessionEJB11Local</code>
EJB Link	<code>SessionEJB11</code>

- f) Save your work and close `web.xml`.

The page you are going to build later in this practice contains a tree of Categories. The root of this tree is based on the `NamedQuery` you created earlier that returns all Categories that do not have a parent. The Leaf of the tree will be from the children `OneToMany` relationship within a Category.

The UI you will be building uses a `TreeModel` object as the source of data for the tree component. The `TreeModel` object has a method that accepts the root objects and traverses the `OneToMany` relationship to get their children.

In the next few steps, you create the `TreeModel` object and write the methods to populate it.

- 8) Open `ProductBrowsingBean.java`.
- 9) Create a private variable `treeModel` of type `TreeModel` and import `org.apache.myfaces.trinidad.model.TreeModel`.
- 10) Right-click the variable and select **Generate Accessors**, choose only the **get** method and click **OK**.
- 11) In this `getTreeModel` method, you want to check if the `treeModel` is null and if it is, call the methods (which you will code next) to build the `treeModel`. The method should look something like:

```
public TreeModel getTreeModel() throws NamingException {  
    if (treeModel == null) {
```



## Practice 11-1: Creating the ProductBrowsing Managed Bean (continued)

```
}  
    return treeModel;  
}
```

- 12) If the `treeModel` is null, you need to create a `List<Category>` and set it to the results of a call to the `NamedQuery queryCategoryFindRoot` in the `SessionBean`. The code should look something like:

```
public TreeModel getTreeModel() throws NamingException {  
    if (treeModel == null) {  
        List<Category> categories =  
            getSessionBean().queryCategoryFindRoot();  
    }  
    return treeModel;  
}
```

- 13) Import `java.util.List` and `oracle.model.Category`.
- 14) Because `getSessionBean` throws a `NamingException`, this method needs to throw it also. Click the **Light bulb** icon in the left column in the code editor and choose **Add Exceptions to Method Header**.
- 15) Add the code to set `treeModel` to the results of the query using `ChildPropertyTreeModel(root, relationship)`. Remember the relationship of the `Category` to its children is named `children`. The completed method should look something like:

```
public TreeModel getTreeModel() throws NamingException {  
    if (treeModel == null) {  
        List<Category> categories =  
            getSessionBean().queryCategoryFindRoot();  
        this.treeModel = new  
            ChildPropertyTreeModel(categories, "children");  
    }  
    return treeModel;  
}
```

- 16) Import `org.apache.myfaces.trinidad.model.ChildPropertyTreeModel`.
- 17) You now have a bean that will return the object required for a `Tree` component on a page. Before your page can use the bean, you have to register it with `faces.config.xml`.
- a) Open `faces-config.xml` in the visual editor.
  - b) Add `ProductBrowsingBean` as a session scope managed bean.
    - i) Click the **Overview** tab.
    - ii) Select **Managed Bean** if it is not already selected.

### ***Practice 11-1: Creating the ProductBrowsing Managed Bean (continued)***

- iii) Click the Plus sign in the upper-right corner of the window.
  - iv) In the Create Managed Bean dialog box, specify `productBrowsingBean` as the Bean Name.
  - v) Set the class name to `ProductBrowsingBean`.
  - vi) Set the package name to `oracle.ui.backing`.
  - vii) Set the Scope to `session`.
  - viii) Deselect **Generate Class If It Does Not Exist**.
  - ix) Click **OK** to register the bean.
  - x) Save your work.
- 18) The `productBrowsingBean` is now registered as a managed bean and will be available to your pages.

## Practice 11-2: Creating a JSF Page

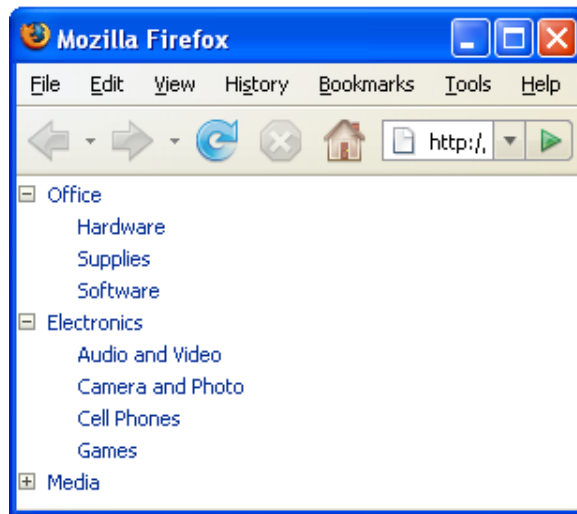
In this practice, you create a JSF page that uses objects in the `productBrowsingBean` as sources of data.

- 1) Open `faces-config.xml`.
- 2) Create a JSF page:
  - a) Click the **Diagram** tab.
  - b) Drag a **JSF Page** from the Component Palette to the diagram.
  - c) Name the component **Main**. Notice that the name shows as `/Main` on the page.
  - d) Double-click **Main**. This opens the Create Page dialog box.
  - e) Accept the default name, which is **Main.jspx**.
  - f) If it is not selected, select **Create as XML Document**.
  - g) Expand **Page Implementation** and choose to **Automatically Expose UI Components in an Existing Bean**, and select **productBrowsingBean**.  
By selecting the bean, JDeveloper will add variables and get and set methods to the bean for each component you add to your page. You probably will not need all of them, but it makes it easy to access values and components later.
  - h) Click **OK** to create the page.
- 3) Add a Tree component to the page and populate it from the bean.
  - a) In the Component Palette, select **ADF Faces** from the drop-down list at the top of the Palette.
  - b) Drag a **Tree** component to the page and enter the following values:

Field	Value
Id	CategoryTree
Value	<code>{productBrowsingBean.treeModel}</code>
Var	Category
  - c) Click **OK** to add the component to the page.
  - d) Expand **af:tree – categoryTree > Tree facets**, in the Structure window.
  - e) Drag an **Output Text** component to **NodeStamp**.
  - f) In the Properties Inspector, set the **Value** property to:  
`{category.categoryName}`
- 4) You are now ready to run and test your page:
  - a) Rebuild both the **UI** and **Model** projects. (Right-click the project and select **Rebuild**)
  - b) Run the **SessionEJB11Bean** from the **Model** project. (Remember that the session bean is in the `oracle.services` package.)
  - c) Click anywhere on your page and select **Run**.

## ***Practice 11-2: Creating a JSF Page (continued)***

5) Your running page should look something like:

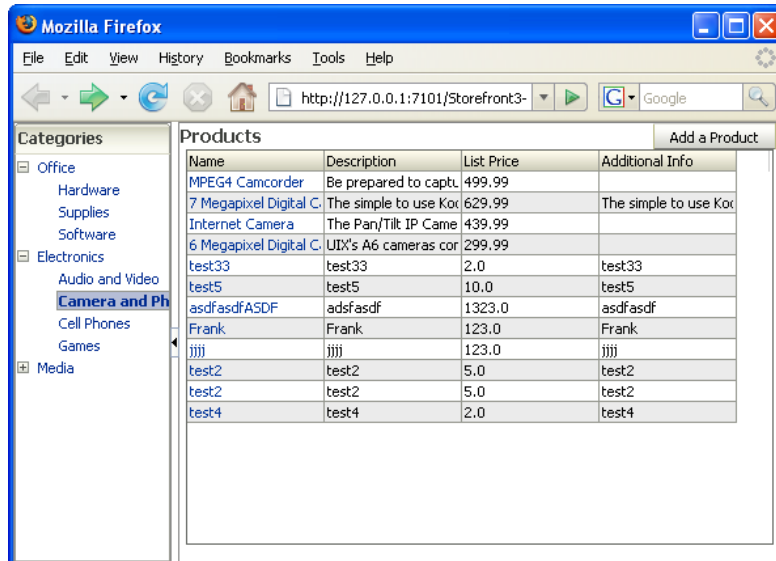


- 6) The page uses the treeModel component, which uses the SessionEJBBean methods to return a list of Categories and their children. For now, you just display the tree structure. In a later practice, you connect the results of the product table to the selected category.
- 7) When you are done, close the browser.

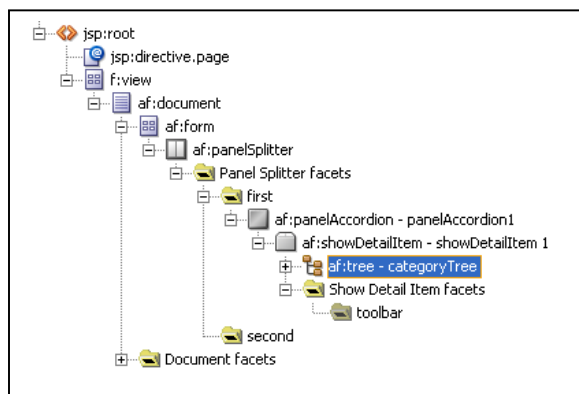
### Practice 11-3: Adding Products to the Page

In this practice, you add some layout components to the Main page, and then add a Table component that displays Products. You will also add the access methods to the managed bean.

The completed Main page will look something like:

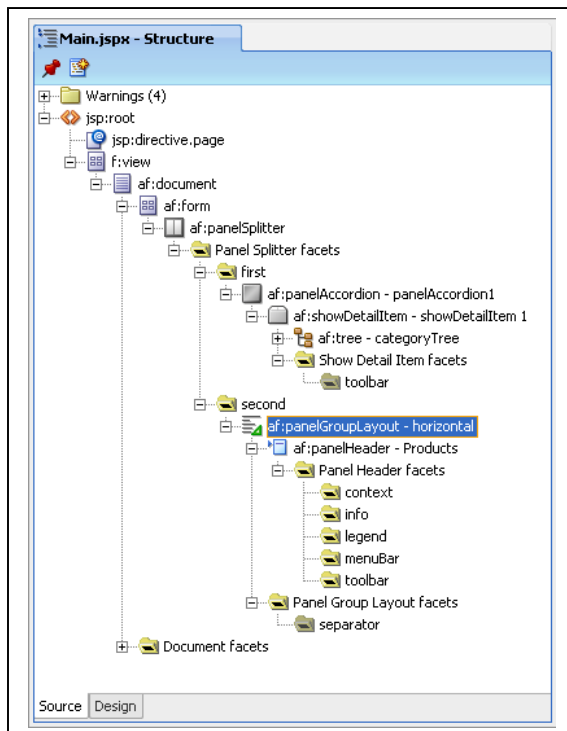


- 1) Open `Main.jspx` in the visual editor.
- 2) Drag a **Layout > Panel Splitter** to `af:form` in the Structure window. The Panel Splitter is what creates the two sections on the page: one for the Category Tree and the other for Products.
- 3) In the Properties Inspector, set the **Splitter Position** to 120.
- 4) Set the **Style > Box > Width** to 100 Percent.
- 5) Still in the Structure window, expand the Panel Splitter facets to show the **first** and **second** facets.
- 6) Add a **Panel Accordion** to the **first** facet.
- 7) Expand `af:PanelAccordion` and drag the `af:tree - categoryTree` component to the `af:showDetailItem` in the Structure window.
- 8) The Structure should look something like:



### ***Practice 11-3: Adding Products to the Page (continued)***

- 9) Select `af:showDetailItem` and set the **Text** property to **Categories**.
- 10) Add the **Panel Group Layout** component to the **second** facet.
- 11) Set the **Layout property** to **horizontal**.
- 12) Set the **Style > Box > Width** to **100 Percent**.
- 13) Add a Panel Header to the `af:panelGroupLayout`.
- 14) Set the **Text** property to **Products**.
- 15) The Structure window should look something like:



Before you can add a table to display Products, you need to add the method to the `productBrowsingBean`. For now, you will add a method that returns all the Products. In a later practice, you will change it to return only the Products for a selected Category.

- 16) Open `ProductBrowsingBean.java`.
- 17) You should first notice that there have been a number of variables and methods added to the class. These are added as you create components on the page because the bean is registered as an automatic backing bean.
- 18) Add a method called `getAllProducts` that returns a `List` and throws `NamingException`.

### ***Practice 11-3: Adding Products to the Page (continued)***

19) Use the `queryProductFindAll()` method on the Session bean to populate a `List<Product>` object.

20) Return the results.

21) You will need to import `oracle.model.Product`.

22) The code should look something like:

```
public List<Product> getAllProducts() throws NamingException{
    List<Product> products =
        this.getSessionBean().queryProductFindAll();
    return products;
}
```

23) Save your work.

Now that you have the method that returns Products, you can add the table to the page.

24) Open `Main.jspx`.

25) Add a **Table** component to `af:panelHeader - Products` in the Structure window.

26) In the **Create ADF Faces Table** dialog box, select **Bind Data Now**.

27) Click **Browse** for Table Data Collection.

28) Select `productBrowsingBean.allProducts` from the JSF Managed Beans list, and click **OK**.

29) Set the list of columns in the table to include only the following columns in this order:

**Product Name**

**Description**

**List Price**

**Additional Info**

30) Click **OK**.

31) Set the Id to `productsTable`.

32) Set **Appearance > Column Stretching** to `Last`.

33) Set **the Style > Box > Width** to `100 Percent`.

34) Test the page. (Remember that the Session bean has to be running.)

### Practice 11-3: Adding Products to the Page (continued)

35) The page should look something like:

showDetailItem 1		Products			
		Product Name	Description	List Price	Additional Info
<div>Office</div> <div>Hardware</div> <div>Supplies</div> <div>Software</div> <div>Electronics</div> <div>Audio and Video</div> <div>Camera and Photo</div> <div>Cell Phones</div> <div>Games</div> <div>Media</div> <div>Books</div> <div>DVDs</div> <div>Periodicals</div> <div>Music</div>		Zune 30Gb	Zune is here. Designe	225.99	The Digital Media Player reinvented. With
		RAZR Cellular Phone	Thin is definitely in. A	259.99	The Moto Razzr V3 is expertly crafted to de
		Muvo Personal MP3 F	The Creative Nomad	99.99	The Creative NOMAD MuVo? combines cut
		Bluetooth Adaptor	SyNET's TBW-101UB I	19.99	
		Plasma HD Television	The TH-42PX60U 42-	1999.99	HDMI (High-Definition Multimedia Interface
		PlayStation 2 Video G	Whether you're a die	199.95	MODEL- 97064 VENDOR- SONY PLAYSTAT
		Treo 650 Phone/PDA	The unlocked PalmOr	0.99	Get All You Need, All In One Treo 650 PDA
		Treo 700w Phone/PD	The Palm® Treo® 700	399.99	
		Tungsten E PDA	The sleek and powerl	195.99	
		XBox Video Game Sys	This is the original XB	159.99	
		XBox 360 Video Game	Xbox 360 sets a new	299.99	Includes: Xbox 360 Console, 1 Wireless Xt
		Playstation Portable	The era of no-compr	199.99	The awe and excitement that has made Sc
		Nintendo DS	Lighter, brighter and	129.99	
		Bluetooth Phone Hea	Streamlined and soph	49.99	
		Ipod Speakers	The Portable Audio S	89.99	
		Creative Zen Vision V	The Creative Zen Visi	389.99	
		Ipod Video 80Gb	Apple iPod - Continui	339.99	
		Ipod Shuffle 1Gb	The Apple 1 GB Shuff	99.99	

Remember that you have not yet coordinated the two areas on the page. The Category tree shows all the Categories and children and the Products table displays all the Products.

36) When you are done, close the browser. You can also Close and remove the application from the IDE.

37) Stop the deployed application.



---

## Practices for Lesson 12

---

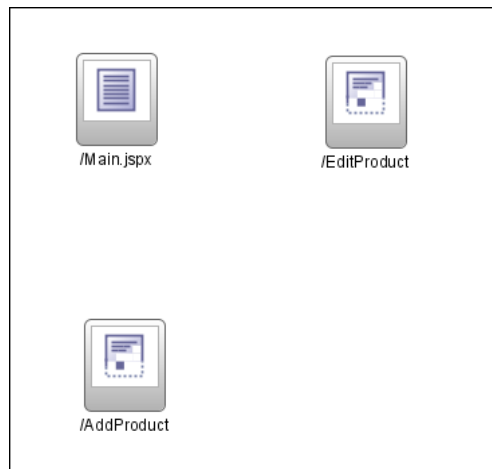
In this practice, you add a couple of pages to the application. One page will be used to edit a product and the other will be used to add a product.

You will add navigation cases to the pages and navigation elements to each page so that the user can move easily between pages. You will create only the page and navigation for the Edit page but you will complete the Add page. In the next practice, you add the event mechanisms to complete the Edit page.

## Practice 12-1: Creating the Edit and Add Pages

In this practice, you create both the EditProduct and AddProduct pages. You also add the Navigation cases to the application page flow.

- 1) Open Application\_12.
- 2) Open faces-config.xml in the UI project.
- 3) Add two JSF pages to the diagram. Name one AddProduct and the other EditProduct.
- 4) Orient the new objects to look something like this:



- 5) Double-click the EditProduct page to create an empty JSF page. Accept the default name, create it as an XML Document and **do not automatically** expose the UI components in a Managed Bean (in Page Implementation).
- 6) Go back to the faces-config.xml diagram.
- 7) Double-click the AddProduct page to create an empty JSF page. Accept the default name, create it as an XML Document. But this time, select the ProductBrowsingBean as the **Managed Bean**.

**IMPORTANT NOTE:** If ProductBrowsingBean is not selectable, choose **Do Not Automatically Expose UI Components in a Managed Bean**. Then edit the source code of addProduct.jspx. Add the following line (in bold) near the bottom of the XML, just before the closing `</jsp:root>` tag:

```
</f:view>
<!--oracle-jdev-comment:auto-binding-backing-bean-
      name:productBrowsingBean-->
</jsp:root>
```

Now that you have created the pages, you can add the Navigation Cases that control the page navigation.

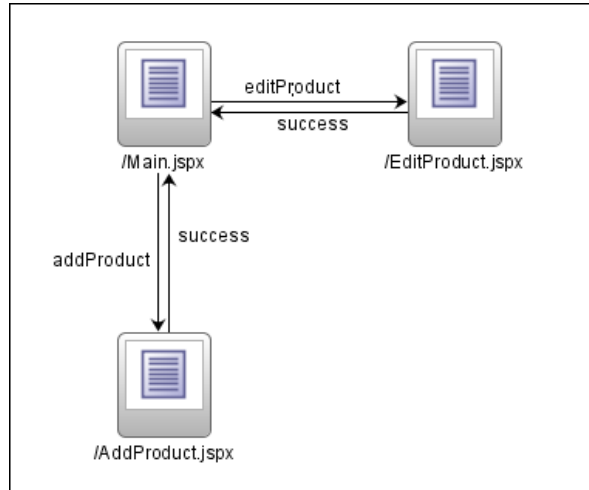
- 8) Open the faces-config.xml diagram.

## Practice 12-1: Creating the Edit and Add Pages (continued)

9) Add the following Navigation Cases:

From Page	To Page	Name
/Main.jspx	/EditProduct.jspx	editProduct
/Main.jspx	/AddProduct.jspx	addProduct
/EditProduct	/Main.jspx	success
/AddProduct	/Main.jspx	success

10) The diagram should now look something like:



Now that the navigation is in place, you can start working on building the AddProduct page.

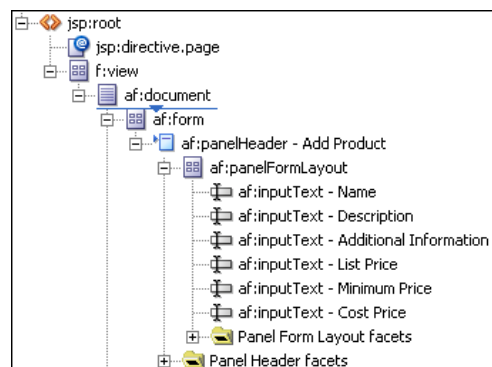
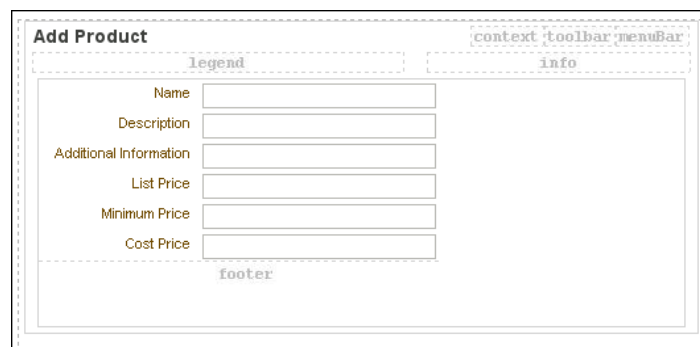
## Practice 12-2: Building the AddProduct Page

In this practice, you add the data components and the necessary managed bean code to complete the AddProduct page. The page can be accessed from a button on the Main page, which you also add.

- 1) Open `AddProduct.jspx` and go to the Design tab.
- 2) Add the ADF Faces layout components to the page.
  - a) Drag a **Panel Header** to the page and set **Text** to Add Product.
  - b) Drag a **Panel Form Layout** to the `af:panelHeader`. (You can drag it to the Structure window.)
- 3) Next, add **Input Text** components to the `af:panelFormLayout` as follows:

Id	Label
npName	Name
npDescription	Description
npAdditionalInfo	Additional Information
npListPrice	List Price
npMinPrice	Minimum Price
npCostPrice	Cost Price

- 4) The page and Structure window should look something like:



- 5) Right-click anywhere on the page and select **Run** to run the page and see how it looks.

## ***Practice 12-2: Building the AddProduct Page (continued)***

6) The page should look something like:

Add Product	
Name	<input type="text"/>
Description	<input type="text"/>
Additional Information	<input type="text"/>
List Price	<input type="text"/>
Minimum Price	<input type="text"/>
Cost Price	<input type="text"/>

### Practice 12-3: Creating the Add Product Functionality

Now that you have a page skeleton with fields, you add the code and buttons to the page that make the Add Product work.

The way the page works is that once a user navigates to the AddProduct page, he or she enters values in the six fields, and then click an Add Product button which calls a method to persist the data and navigate back to the Main page.

- 1) Still on the **AddProduct** page, add an **ADF Faces Toolbar** to the **Panel Form Layout facets > footer**.
- 2) Add an **ADF Faces Toolbar Button** to the toolbar.
- 3) Change the **Text** to Save and the **Id** to addProduct.
- 4) Double-click the Save button. In the **Bind Action Property** dialog box, ensure that the Managed Bean is productBrowsingBean, change the Method to addProduct, and then click **OK**. The editor opens productBrowsingBean at the new method.

In the next few steps, you add code that uses the values stored in the managed bean that correspond to each of the fields in the form (npName, npDescription, and so on) to create a new product. The code then persists that new row.

Because the page you are building contains a subset of all the columns in the table, you will set default values for a number of the columns.

- 5) Create default values in the addProduct () method for the following columns:

Type	Attribute Name	Value
String	url	"images/17.jpg"
String	shippingClassCode	"CLASS1"
Long	supplierId	123L
Long	warrantyMonths	12L
String	status	Null
Category	category	Null

- 6) Add code that uses the Product constructor to create a new Product object named newProduct. The values for the Product constructor arguments come from either the default values you just added or the values from the page. Remember the field values are stored in the managed bean, so they are accessible by name (npName, npDescription, and so on). You can look at the Product constructor in the Model project to see what arguments are used. However, you can also use the following code.

```
Product newProduct = new Product(  
    (String)this.getNpAdditionalInfo().getValue()  
    ,category  
    ,Double.parseDouble(this.getNpCostPrice()  
                        .getValue().toString())  
    ,getNpDescription().getValue().toString()  
    ,url  
    ,Double.parseDouble(this.getNpListPrice()  
                        .getValue().toString())
```

### **Practice 12-3: Creating the Add Product Functionality (continued)**

```
        ,Double.parseDouble(getNpMinPrice()  
                            .getValue().toString())  
        ,getNpName().getValue().toString()  
        ,shippingClassCode  
        ,supplierId  
        ,warrantyMonths);
```

- 7) Now that you have a new Product object, you can add a try-catch block to persist the new Product.
- a) In the try block, call `persistEntity` on the SessionBean passing `newProduct` as an argument. This is where the object is persisted to the database.
  - b) Next set the status variable to success.
  - c) In the catch block, set the status to fail.
  - d) Next change the return statement to return status.

That should complete the code for the AddProduct page. The completed method should look something like:

```
public String addProduct() {  
    String url = "images/17.jpg";  
    String shippingClassCode = "CLASS1";  
    Long supplierId = 123L;  
    Long warrantyMonths = 12L;  
    String status = null;  
    Category category = null;  
  
    Product newProduct = new Product(  
        (String)getNpAdditionalInfo().getValue()  
        , category  
        , Double.parseDouble(getNpCostPrice().getValue().toString())  
        , getNpDescription().getValue().toString()  
        , url  
        , Double.parseDouble(getNpListPrice().getValue().toString())  
        , Double.parseDouble(getNpMinPrice().getValue().toString())  
        , getNpName().getValue().toString()  
        , shippingClassCode  
        , supplierId  
        , warrantyMonths  
    );  
  
    try {  
        getSessionBean().persistEntity(newProduct);  
        status = "success";  
    } catch (Exception ex) {  
        ex.printStackTrace();  
        status = "fail";  
    } finally {  
    }  
    return status;  
}
```

- 8) Compile and save your work.

### ***Practice 12-3: Creating the Add Product Functionality (continued)***

Next you need to add a button to the Main page that will navigate to the AddProduct page.

- 9) Open `Main.jspx`.
  - 10) Add a **Button** to the toolbar facet of the **Products Panel Header**.
  - 11) Set the **Id** to `addProductButton` and the Text to `Add a Product`.
  - 12) Set the Action to `addProduct`. (This is the Navigation Case that references the `AddProduct.jspx` page.)
- Now you can test the page to see that you can add a Product.
- 13) Run `Main.jspx` from the `faces.config.xml` diagram.
  - 14) Click **Add a Product**.
  - 15) On the Add Product page, enter any values you want. Use values that you will be able to recognize when you go back to the main page. Also, use valid values for the number fields.
  - 16) When you have finished adding values, click **Save**.
  - 17) You will now be back on the Main page. Use the scroll bars to scroll down to find the row you just added.
  - 18) Click Add Product again and notice that the values you entered for the new row are still in the form. That is because the form fields are persisted in the managed bean.
  - 19) When you are done, close the browser.

In the next few steps, you create a method that resets those fields. You can then create a Cancel button and method that calls the reset method and returns to the Main page.

- 20) In the `productBrowsingBean`, create a `resetAddProductFields()` method that calls `resetValue()` on each of the fields on the page. The completed code should look something like:

```
private void resetAddProductFields() {  
    getNpName().resetValue();  
    getNpDescription().resetValue();  
    getNpAdditionalInfo().resetValue();  
    getNpListPrice().resetValue();  
    getNpMinPrice().resetValue();  
    getNpCostPrice().resetValue();  
}
```

- 21) The first use of this method is to reset the values after a successful persist of the values. Add a call to this method to the `addProduct()` method, just after `persistEntity(newProduct)`.
- 22) Now add a Toolbar button to the **AddProducts** page which calls `resetAddProductsFields()` and returns **success**. This will clear the fields and return to the Main page.



### ***Practice 12-3: Creating the Add Product Functionality (continued)***

- a) Add a **Toolbar button** to the right of the Save button. (It may be easier to drag it to the Structure window.)
- b) Set the Id to cancelButton and Text to Cancel.
- c) Double-click the button to create the cancelButton\_action method in the productBrowsingBean.
- d) Add code to this method to call resetAddProductsFields() and return success.
- e) The code should look something like:

```
public String cancelButton_action() {  
    resetAddProductFields();  
    return "success";  
}
```

23) Compile, save, and test the page.

- a) Add another Product and save it.
- b) Start to add another Product and click Cancel instead of Save.

24) When you are done, close the browser.

25) You may also close the application and remove it from the IDE.

26) Stop the running application.

## Practices for Lesson 13

The application you have built so far has not accounted for selecting a category in the Category tree or the coordinated display of Products. In the completed application, the user clicks a category in the tree and the products within that category are displayed in the Products section of the page. From the Products section, the user can choose to either add a product or can click a product and edit it. To add a product, the product should be added to the currently selected Category. This is one part of the application that you did not add in the previous practice.

## Practice 13-1: Adding the Category Tree Selection Event

In this practice, you create a `SelectionListener` for the Category Tree. This listener will determine which category the user has selected, and then set a class variable to the selected category.

You also modify the Products table to get its data from a new method that retrieves the products for the selected category.

- 1) Open `Application_13.jws`.
- 2) Open `Main.jspx`.
- 3) Select the **af:tree** component in either the Structure window or the Design editor.
- 4) In the Properties Inspector, set **Behavior > RowSelection** to **single**.
- 5) Set **SelectionListener** to `categoryTreeSelectionListener`. This also adds the method to `productsBrowsingBean`.
- 6) Open `productBrowsingBean.java`.
- 7) Add a private class variable to hold the selected Category. Name it `selectedCategory` and create a default get method using Generate Accessors. The variable declaration should look like the following:

```
private Category selectedCategory;
```

- 8) Locate the `categoryTreeSelectionListener()` method you just created. (It will probably be at the bottom of the class.)
- 9) The technique for retrieving the selected row from the tree is not difficult, but it requires using methods that belong to a `UITree` component.
- 10) The first thing you need to do is get the current instance of the `UITree`. Remember the bean has get and set methods for all of the components on the page. The code is:

```
public void categoryTreeSelectionListener
                               (SelectionEvent selectionEvent) {
    // Add event code here...
    UITree treeTable = getCategoryTree();
}
```

- 11) Import `org.apache.myfaces.trinidad.component.UITree`.
- 12) Because of the native behavior of the component, you need to create an `Iterator` of `selectedRowKeys` from the `treeTable`. Note that even though our UI supports only one row selection, the component supports multiple selections; therefore, we have to use the iterator.

```
public void categoryTreeSelectionListener
                               (SelectionEvent selectionEvent) {
    // Add event code here...
    UITree treeTable = getCategoryTree();
    Iterator selection = treeTable.getSelectedRowKeys().iterator();
}
```

- 13) Import `java.util.Iterator`.

### ***Practice 13-1: Adding the Category Tree Selection Event (continued)***

14) Next you loop through the iterator (knowing there should only be one row). You get the selection Object, and use it as a rowKey.

You then call `setRowKey(rowKey)` on the `treeTable`. After that is set, you set `this.selectedCategory` to `treeTable.getRowData()` cast to a `Category` object. The code should look something like:

```
public void categoryTreeSelectionListener
        (SelectionEvent selectionEvent) {
    // Add event code here...
    UIXTree treeTable = getCategoryTree();
    Iterator selection = treeTable.getSelectedRowKeys().iterator();
    for (;selection.hasNext();) {
        Object rowKey = selection.next();
        System.out.println("Here is the rowKey:  " + rowKey);
        treeTable.setRowKey(rowKey);
        this.selectedCategory = (Category)treeTable.getRowData();
    }
}
```

With this listener code, you can now know what category the user selects. With that information, you then populate the products table with only the related products.

In the next few steps, you create a method that returns a list of products for the selected category.

15) Create a method that returns a `List<Product>` named `getProductsForSelectedCategory`. The method should throw a `NamingException`.

```
public List<Product> getProductsForSelectedCategory()
        throws NamingException {
}
```

16) In the method, check whether `this.selectedCategory` is not null and if it is not, return the results of a call to `getProductList()` in the selected category. If it is null, return `Collection.emptyList()`.

```
public List<Product> getProductsForSelectedCategory()
        throws NamingException {
    if (this.selectedCategory != null) {
        return this.selectedCategory.getProductList();
    } else {
        return Collections.emptyList();
    }
}
```

17) Import `java.util.Collections`.

18) Compile and save your work.

Now that you have a method that returns the related categories, you just need to modify the products table to use it.

## **Practice 13-1: Adding the Category Tree Selection Event (continued)**

19) Open `Main.jspx`.

20) Select the `productsTable` and change the `Value` attribute to use `productsForSelectedCategory`.

**Note:** Use the Expression Builder to select this method.

21) Set the **Partial Trigger** (under **Behavior**) to use `categoryTree`.

a) Click the arrow to the right of the `PartialTrigger` property and select **Edit**.

b) In the Edit property dialog box, expand the **facet** (first) to find `tree - categoryTree`.

c) Shuttle it to the Selected pane and click **OK**.

22) Run and test the page. You will see that as you click a category, the products within that category are displayed.

The last thing you need to do is coordinate the Add Product page with the currently selected category. In other words, when the user adds a product, the product will be defaulted to the currently selected category. You also want to disable the Add Product button when there is no category selected. Because we want any new products to be within a selected category, you need to ensure that the user can only go to the Add Product page if there is a selected category.

23) To disable the button, you need to create a method in the managed bean named `isAddProductButtonDisabled` that returns a boolean:

```
public boolean isAddProductButtonDisabled () {  
    return true;  
}
```

24) You want to add code that returns true if `selectedCategory` is null or if the `selectedCategory` does not have children. The code looks like:

```
public boolean isAddProductButtonDisabled () {  
    return (selectedCategory == null ||  
            !selectedCategory.getChildren().isEmpty());  
}
```

25) Now that you have the method, bind the **Disabled** property to the method.

a) Click the **Add a Product** button in `Main.jspx`.

b) Click the **Expression Builder** on the **Behavior > Disabled** property.

c) Select `addProductButtonDisabled` in the Expression Builder. (You added it to the `productBrowsingBean`.)

d) Click **OK**.

e) Set the **Partial Trigger** property to `categoryTree` using Edit.

26) Run the main page and notice that the button is disabled until you select a leaf node category in the tree.

## **Practice 13-1: Adding the Category Tree Selection Event (continued)**

Now that the button only works when a category is selected, you must use that category on the Add Product page.

27) Open `productBrowsingBean.java`.

28) Find the `addProduct()` method.

29) Change the `category` variable definition from initializing to `null` to initializing to the `selectedCategory`.

Now that you are attaching a product to a specific category (it used to be `null`), you need to call the `addProduct()` on the `selectedCategory` before you persist the product.

30) Add the call to `selectedCategory.addProduct(newProduct)` to the try block just before the `persist`.

31) Run and test the application. Add several products to several different categories.

32) When you are done, close the browser.

Each of the fields on the AddProduct page is mandatory. The last step in creating the AddProduct page is to add validation to the fields.

33) Open `AddProduct.jspx`.

34) To apply the following validations from the Structure window, right-click **Insert Inside (field) > ADF Faces**. Add appropriate text for Hints and Messages.

Field	Validator	Rules
Name	Validate Length	Minimum 5, Maximum 30
Description	Validate Length	Minimum 5, Maximum 3000
Additional Information	Validate Length	Minimum 5, Maximum 3000

35) Set the **Behavior > Required** property to `True` for each of the fields.

36) Run and test the page (remember to start with `Main.jspx`).

- Add products to several categories.
- Test the validators by entering less than the minimum length.
- Test the maximum length in the Name field.
- Try to add a product with one or more of the price fields blank or out of range.

The Add Product page is now complete. In the next section, you add the Edit Page.

## Practice 13-2: Complete and Connect the Edit Product Page

In this practice, you complete and connect the Edit page to the main page.

Before you add fields to the Edit page, you need to add the elements to the Main page that will allow the user to select a row to edit. That selection gesture will also navigate to the Edit page.

- 1) Open `Main.jspx`.
- 2) Select the **Products** table and set the **Behavior > RowSelection** property to **single**.
- 3) Expand `af:column - Product Name`, right-click `af:outputText`, and choose **Surround With**.
- 4) Choose **Link** and click **OK**.
- 5) Remove the default value of the **Command Link Text**. (That should reset the displayed value to `ProductName`.)
- 6) Set the **Action** property to `editProduct`.
- 7) Add a method to `ProductBrowsingBean` that returns the row data from the selected row.
  - a) Open `ProductBrowsingBean.java`.
  - b) Add a public method that returns a `Product` object from the **ProductsTable** using the `getSelectedRowData()` method. The code should look something like:

```
public Product getSelectedProduct () {  
    return (Product) getProductsTable().getSelectedRowData();  
}
```

You will use this method as the source data for the Edit page fields.

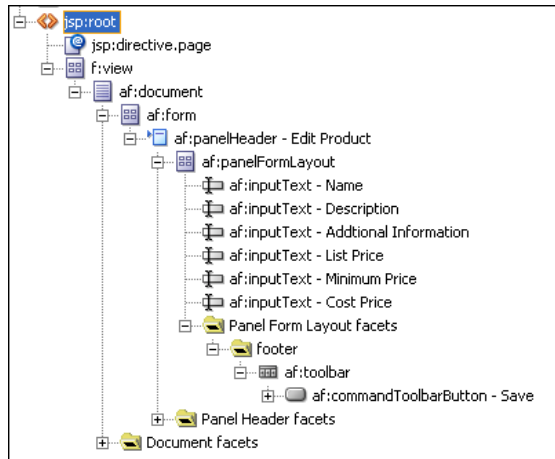
Next, you add the fields and buttons to the Edit page.

- 8) Open `EditProduct.jspx`.
- 9) Add a **Panel Header** and set the **Text** property to `Edit Product`.
- 10) Add a **Panel Form Layout**.
- 11) Add a **Toolbar** to the **Panel Form Layout footer** facet.
- 12) Add a **Toolbar Button** to the **Toolbar**. Set the **Text** to `Save`.
- 13) Add **Input Text** components to the `af:panelFormLayout` as follows:

Id	Label
<code>prName</code>	Name
<code>prDescription</code>	Description
<code>prAdditionalInfo</code>	Additional Information
<code>prListPrice</code>	List Price
<code>prMinPrice</code>	Minimum Price
<code>prCostPrice</code>	Cost Price

## Practice 13-2: Complete and Connect the Edit Product Page (continued)

14) The Structure window should look something like:



15) The page should look something like:

Edit Product		context toolbar menuBar
<div>legend info</div>		
Name	<input type="text"/>	
Description	<input type="text"/>	
Additional Information	<input type="text"/>	
List Price	<input type="text"/>	
Minimum Price	<input type="text"/>	
Cost Price	<input type="text"/>	
<div>Save</div>		

Next, you need to set the **Value** property of all the fields to get their data from `selectedProduct` (that is the method you added in the last section).

16) Select the **Name** field and set the Value property to:

```
#{productBrowsingBean.selectedProduct.productName}
```

17) You can use the Property Inspector or the Expression builder.

18) Set the rest of the fields to their corresponding attributes in `selectedProduct`.

The last step is to create a method that persists the changes and to connect the Save and Cancel buttons.

19) Open `productBrowsingBean.java`.

20) Add a public method named `updateProduct()` which returns a `String`.

21) Add a try-catch block.



## ***Practice 13-2: Complete and Connect the Edit Product Page (continued)***

- 22) In the try block, call `mergeEntity` on the `SessionBean` and pass it `this.getSelectedProduct()`. Create a `String` variable named `status`. Set `status` to "success" after the call to `mergeEntity`.
- 23) Return `status`.
- 24) Add `e.printStackTrace()` to the catch block followed by setting `status` to "success."
- 25) The completed code should look something like:

```
public String updateProduct() {
    String status = null;
    try {
        this.getSessionBean().mergeEntity(this.getSelectedProduct());
        status = "success";
    } catch (NamingException e) {
        e.printStackTrace();
        status = "success";
    }
    return status;
}
```

- 26) Set the Save button to call `updateProduct()`.
- a) Open `EditProduct.jspx`.
  - b) Double-click the **Save** button.
  - c) In the Bind Action Property dialog box, choose `producBrowsingBean` and `updateProduct` as the method.
  - d) Click **OK**.
- 27) Run `Main.jspx` to test your Edit page.
- a) Select a category and any product.
  - b) Make changes to the product and click **Save**.
  - c) Check whether the changes show in the Product table.
- 28) When you are done, close the browser.
- 29) You may also close the application and remove it from the IDE.

## Practices for Lesson 14

In this practice, you create a message-driven JMS application to send emails to notify the users.

You perform the following set of tasks in this practice set:

- Creating a JMS connection factory and a queue destination by using the WebLogic Server Administration Console
- Creating a message-driven bean that accepts messages sent to the JMS queue and sends emails based on those messages. You use Oracle JDeveloper as the development tool to develop the message bean
- Creating a business method in a session bean that sends messages to the JMS queue when an email message needs to be sent
- Creating a sample test client in JDeveloper to test the behavior of the confirmation email by creating a service request

## ***Practice 14-1: Creating a JMS Connection Factory and Queue Destination in WebLogic Server***

A JMS server implements the JMS infrastructure on a WebLogic server. Destinations (queues or topics) are targeted to a WebLogic server that has the JMS server configured. In this practice, you configure a JMS server, a JMS module, a Connection Factory, and a queue.

- 1) Open `Application14.jws`.
- 2) In the Menu, select **Run > Start Server Instance**.
- 3) Configure a WebLogic JMS Server by using the WebLogic Server Administration Console.
  - a) Open a browser and enter `http://127.0.0.1:7101/console`.
  - b) Log in to the WebLogic Server Administration Console using `weblogic/weblogic`.
  - c) On the Home page, click JMS Servers under the Services of Domain Configurations section.
  - d) Click **New** under the JMS Servers table to create a new JMS Server.
  - e) Specify the following properties and click Next:  
Name: `MyJMSServer`  
Persistent Store: `none`
  - f) Select `DefaultServer` as the target JMS server. Click **Finish**.
- 4) Configure a JMS module and a queue.
  - a) In the WebLogic Server Administration Console, navigate to **DefaultDomain > Services > Messaging > JMS Modules**.
  - b) Click **New** in the JMS Modules table and specify the following properties and click **Next**.  
Name: `MyJMSModule`  
Descriptor File Name: `MyJMSModule`
  - c) Select `DefaultServer` as the **target**. Click **Next**.
  - d) Select the **Would you like to add resources to this JMS system module** check box, and click **Finish**.
  - e) In the Settings for the `MyJMSModule` page, click the **Subdeployments** tab.
  - f) In the **Subdeployments** table, click **New**. Enter `MyJMSSubmodule` as the Subdeployment Name and click **Next**.
  - g) On the Targets page, select the `MyJMSServer` as the target under the **JMS Servers** table. Click **Finish**.
  - h) Click the **Configuration** tab.

## **Practice 14-1: Creating a JMS Connection Factory and Queue Destination in WebLogic Server (continued)**

- i) On the Settings for MyJMSModule page, under the Summary of Resources table, click **New** to configure a new connection factory for the JMS module.
- j) On the Create a New JMS System Module Resource page, under Choose the type of resource you want to create, select **Connection Factory**, and then click **Next**.
- k) In **Connection Factory Properties**, specify the following parameters and click **Next**.

Name: demoConnectionFactory

JNDI name: weblogic.jms.demoConnectionFactory

- l) Select **Advanced Targeting**. Select MyJMSSubmodule from the subdeployments list. Click **Finish**.
- m) On the Settings for MyJMSModule page, under the Summary of Resources table, click **New** to configure a new JMS queue for the JMS module.
- n) On the Create a New JMS System Module Resource page, under Choose the type of resource you want to create, select **Queue**, and then click **Next**.
- o) In **JMS Destination Properties**, specify the following parameters and click **Next**.

Name: demoQueue

JNDI name: weblogic.jms.demoQueue

Template: None

- p) Select MyJMSSubmodule from the subdeployments list. Click **Finish**.
- 5) Confirm the registration of weblogic.jms.demoConnectionFactory and weblogic.jms.demoQueue in the WebLogic Server's JNDI tree.
- a) Navigate to the Home page. On the Home page, under Environment, click Servers to get to the Summary of Servers. In the table of servers, click DefaultServer.
  - b) On the **Settings for DefaultServer** page, click the **View JNDI Tree** link.
  - c) The JNDI Tree opens in another window or tab. In that tab, notice the JNDI entries of weblogic.jms.demoConnectionFactory and weblogic.jms.demoQueue.

**Note:** Expand **DefaultServer > weblogic > jms** in the JNDI Tree Structure to view the connection factory and the queue.

## Practice 14-2: Creating a Message-Driven Bean to Send Emails

In this practice, you create a message-driven bean that accepts messages from the JMS resource `jms/demoQueue` and sends out email messages to an SMTP server using the Java Email API.

- 1) In the **Model** project, create a message-driven bean called `SendMailMessage` in the `org.demo.business` package using the JDeveloper wizard. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Applications Navigator	Right-click <b>Application_14</b> > <b>Model</b> . Select <b>New</b> .
b.	New Gallery	Categories: <b>Business Tier</b> > <b>EJB</b> Items: <b>Message-Driven Bean</b> Click <b>OK</b> .
c.	Create Message-Driven Bean - Step 2 of 4	EJB Name: <code>SendMailMessage</code> Transaction Type: Container (verify) Mapped Name: <code>weblogic.jms.demoQueue</code> Click <b>Next</b> .
e.	Create Message-Driven Bean - Step 3 of 4	Bean Class: <code>org.demo.business.SendMailMessageBean</code> Click <b>Finish</b> .

- 2) Modify the `@MessageDriven` annotation to specify the use of the JMS resource `jms/demoQueue` by including an `activationConfig` that has three properties:

```
connectionFactoryJNDIName = weblogic.jms.demoConnectionFactory
destinationName = weblogic.jms.demoQueue
destinationType = javax.jms.Queue
```

**Note:** Ensure that the following classes are imported:

```
javax.ejb.MessageDriven
javax.ejb.ActivationConfigProperty
```

```
import javax.ejb.ActivationConfigProperty;
...
@MessageDriven(
    mappedName = "weblogic.jms.demoQueue",
    name = "SendMailMessageBean",
    activationConfig = {
        @ActivationConfigProperty(
            propertyName = "connectionFactoryJndiName",
            propertyValue = "weblogic.jms.demoConnectionFactory"),
        @ActivationConfigProperty(
            propertyName = "destinationName",
```

## Practice 14-2: Creating a Message-Driven Bean to Send Emails (continued)

```
        propertyValue = "demoQueue"),  
  
        @ActivationConfigProperty(  
            propertyName = "destinationType",  
            propertyValue = "javax.jms.Queue")  
    })  
  
    public class SendMailMessageBean implements MessageListener  
    {
```

- 3) In the `onMessage()` method, declare a new variable called `prop` whose type is `java.util.Properties`. With the `Properties.put()` method, add the `mail.smtp.host` property name with a value of `srdemo.org` as a name-value pair property to the `prop` object, and get a `javax.mail.Session` using that property.

**Hint:** Import the required packages.

```
import javax.mail.Session;  
import java.util.Properties;  
...  
public class SendMailMessageBean implements MessageListener {  
    public void onMessage(Message message) {  
        Properties props = new Properties();  
        props.put("mail.smtp.host", "srdemo.org");  
        Session session = Session.getInstance(props);  
        ...  
    }  
}
```

**Note:** If there are any errors in importing the `javax.mail.*` package, add the Java EE 1.5 library to the Model project. Execute the following set of tasks to add the library:

- i. Right-click Model and select Project Properties from the shortcut menu.
  - ii. In the Project Properties window, select Libraries and Classpath in the left pane of the window. You can see classpath entries and the libraries description of the default project libraries in the right pane.
  - iii. Click the Add Library button. Select **Java EE 1.5** from the Add Library window and click **OK**. The library is added to the Model project.
- 4) In a try-catch block, create the `from`, `to`, `subject`, and `content` String variables and use them to store corresponding properties from the message parameter. Use these variables and the current date to construct a `javax.mail.Message` using the `MimeMessage` implementation class. Send the message using `javax.mail.Transport`.

**Note:** Ensure that the required classes are imported from the `javax.mail` and `javax.mail.internet` packages.

```
import javax.mail.Transport;  
import javax.mail.internet.InternetAddress;  
import javax.mail.internet.MimeMessage;
```

## ***Practice 14-2: Creating a Message-Driven Bean to Send Emails (continued)***

```
...

public void onMessage(Message message) {
    Properties props = new Properties();
    props.put("mail.smtp.host", "srdemo.org");
    Session session = Session.getInstance(props);
    try {
        String from = message.getStringProperty("from");
        String to = message.getStringProperty("to");
        String subject = message.getStringProperty("subject");
        String content = message.getStringProperty("content");
        javax.mail.Message msg = new MimeMessage(session);
        msg.setFrom(new InternetAddress(from));
        InternetAddress[] address = {new InternetAddress(to)};
        msg.setRecipients(
            javax.mail.Message.RecipientType.TO, address);
        msg.setSubject(subject);
        msg.setSentDate(new java.util.Date());
        msg.setContent(content, "text/html");
        Transport.send(msg);
    }
    catch (Throwable ex) { ex.printStackTrace(); }
}
```

- 5) Save and compile the message-driven bean.

### **Practice 14-3: Creating a Session Method for Sending Email Messages**

In this practice, you add a general-purpose `sendMailMessage` method to the `ServiceRequestSessionBean` bean that is used to send JMS messages to the `weblogic.jms.demoQueue` JNDI resource for processing by the message-driven bean created in the preceding task.

- 1) Edit `ServiceRequestSessionBean.java` and add a `SendMailMessage()` method that accepts four `String` parameters: `from`, `to`, `subject`, and `content`.

```
...
public void sendMailMessage(String from, String to,
String subject, String content) {
}
...
```

- 2) In a try-catch block, use JNDI to look up the name, `weblogic.jms.demoConnectionFactory`, and use it to create and start a connection.

**Note:** Be sure to import the following interfaces: `javax.jms.Connection`, `javax.jms.ConnectionFactory`, and `javax.naming.InitialContext`.

```
...
public void sendMailMessage(String from, String to,
String subject, String content) {
try {
System.out.println("Looking up CF");
ConnectionFactory connectionFactory =
    (ConnectionFactory) new InitialContext()
        .lookup("weblogic.jms.demoConnectionFactory");
Connection connection = connectionFactory.createConnection();
connection.start();
}
catch (Throwable ex) { ex.printStackTrace(); }
}
...
```

- 3) Use the connection to start a JMS Session. Then perform a JNDI lookup for the `weblogic.jms.demoQueue` name to create a JMS Destination; typecast the result of the JNDI lookup as a `Queue`. Finally, use the JMS Session and JMS Destination to create a JMS `MessageProducer`.

**Note:** Be sure to import the following interfaces: `javax.jms.Session`, `javax.jms.Destination`, and `javax.jms.MessageProducer`.

```
...
public void sendMailMessage(String from, String to,
String subject, String content) {
try {
...
connection.start();
System.out.println("Starting Queue Session");
}
```



### **Practice 14-3: Creating a Session Method for Sending Email Messages (continued)**

```
Session queueSession = connection.createSession(false,
                                                Session.AUTO_ACKNOWLEDGE);
Destination queue = (Queue) new InitialContext()
                    .lookup("weblogic.jms.demoQueue");
MessageProducer publisher =
                    queueSession.createProducer(queue);
}
...
```

- 4) Create a JMS Message with the JMS Session. Call the `setJMSType()` method of the Message instance to set its parameter string to MailMessage, and set String properties for from, to, subject, and content using the corresponding method parameters.

**Note:** Be sure to import the `javax.jms.Message` interface.

```
...
public void sendMailMessage(String from, String to,
String subject, String content) {
    try {
        ...
        MessageProducer publisher =
                                queueSession.createProducer(queue);
        Message message = queueSession.createMessage();
        message.setJMSType("MailMessage");
        message.setStringProperty("from", from);
        message.setStringProperty("to", to);
        message.setStringProperty("subject", subject);
        message.setStringProperty("content", content);
    }
    ...
}
```

- 5) Send the message with the MessageProducer, and then close the MessageProducer, the Session, and the Connection.

```
...
public void sendMailMessage(String from, String to,
String subject, String content) {
    try {
        ...
        Message message = queueSession.createMessage();
        message.setJMSType("MailMessage");
        message.setStringProperty("from", from);
        message.setStringProperty("to", to);
        message.setStringProperty("subject", subject);
        message.setStringProperty("content", content);
        publisher.send(message);
        System.out.println("Message Sent to JMS Queue");
        publisher.close();
        queueSession.close();
        connection.close();
    }
}
```

### ***Practice 14-3: Creating a Session Method for Sending Email Messages (continued)***

```
}  
...
```

- 6) Modify the local and remote interfaces for the Session Facade to include the signature for the `sendMailMessage` method. In `ServiceRequestSession.java` and `ServiceRequestSessionLocal.java`, add the following code:

```
...  
void sendMailMessage(String from, String to, String subject,  
String content);  
...
```

- 7) Save and compile the Model project.

## ***Practice 14-4: Creating a Sample Test Client in JDeveloper***

In this practice, you create a sample test client in JDeveloper to create a service request and test the `SendMessage()` method of the `ServiceRequestSessionBean.java`. After the application sends an email confirmation, you use an email client to verify receipt of the confirmation email.

- 1) Generate a sample Java client to test the `ServiceRequestSessionBean`.
  - a) In the Application Navigator, right-click `ServiceRequestSessionBean.java` and select the **New Sample Java Client** option from the shortcut menu.
  - b) In the Create Sample Java Client window, specify the following details and click **OK**.

Client Project: Model.jpr

Client Class Name:

`org.demo.client.ServiceRequestSessionClient`

Application Server Connection: IntegratedWLSConnection

- 2) Add and implement the `confirmByEmail()` method in the `ServiceRequestSessionClient.java`. The method creates the email message and invokes the `SendMessage()` method.

```
import org.demo.business.ServiceRequestSession;

...

public class ServiceRequestSessionClient {

    ...

    public static void confirmByEmail(ServiceRequestSession message) {

        System.out.println("**** Confirm By Email requested for Service
                               Request Id: \"SKING11296\"");

        try {
            String from = "support@srdemo.org";
            String to = "steve.king@srdemo.org";
            String subject = "Notification: OrderID #SKING11296 Created";
            String msgText = "<html>" +
                            "<body>" +
                            "<h2>OrderID #SKING11296 Created</h2>" +
                            "<p>Dear Steve,</p>" +
                            "<p>Thanks for submitting the order. Your
                                ordered has been entered into our system, and
                                will be processed in a couple of days.</p>" +
                            "<p>Yours sincerely,<br>Support</p>" +
                            "</body></html>";

            System.out.println("**** before sendMessage ");
            message.sendMessage(from, to, subject, msgText);
        }
        catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

### ***Practice 14-4: Creating a Sample Test Client in JDeveloper (continued)***

```
...  
    }
```

- 3) Modify the `main()` method of the `ServiceRequestSessionClient` class to invoke the `confirmByEmail()` method.

```
...  
  
public static void main(String[] args) {  
    try {  
        final Context context = getInitialContext();  
        ServiceRequestSession serviceRequestSession =  
            (ServiceRequestSession) context.lookup("ServiceRequestSessionEJB#org.  
                                                  demo.business.ServiceRequestSession");  
        confirmByEmail(serviceRequestSession);  
        System.out.println("Mail Sent....");  
    } catch (Exception ex) {  
        ex.printStackTrace();  
    }  
}  
...
```

- 4) Save and compile the Model project

## Practice 14-5: Testing the Confirmation Email

In this practice, you execute `ServiceRequestSessionBean` on integrated WebLogic server, and test the bean by using the sample `ServiceRequestSessionClient` Java client. You also use an email client to verify receipt of the confirmation email.

- 1) Execute the `ServiceRequestSessionBean` on the integrated WebLogic server.
  - a) In the Application navigator pane, right-click `ServiceRequestSessionBean.java` and select the **Run** option from the shortcut menu.
  - b) Observe the Message:Log pane for the successful deployment of the bean to the WebLogic Server.
- 2) Execute the sample Java client.
  - a) In the Application navigator pane, right-click `ServiceRequestSessionClient.java` and select the **Run** option from the shortcut menu.
  - b) Observe the Log pane for the successful execution of the client.

**Note:** Click the Running:DefaultServer tab in the Log pane to verify the successful delivery of the message to the client. You can see the following message on the successful execution of the application:

```
Looking up CF
Starting Queue Session
Message Sent to JMS Queue
```

- 3) Start Outlook Express application window, and view the email messages for Steve King. Use the information in the following table to perform this task or use the screenshots after the table as a guide:

Step	Screen/Page Description	Choices or Values
a.	Outlook Express - Main Identity	Click the Send/Recv icon on the toolbar.
b.	Login - soademo.org	User Name: <code>steve.king</code> Password: <code>oracle</code> Click <b>OK</b> .
c.	Outlook Express - Main Identity	In the Email section, confirm that there is an unread mail message.

## Practices for Lesson 15

In this practice, you open and modify a simple JSF application that enables book records to be inserted, updated, and deleted from the database. The JSF application uses a session-scoped backing bean to interact with an instance of the `BooksFacadeEJBBean` stateful session bean, which executes the persistence and transactional logic.

You perform the following set of tasks in this practice set:

- Executing the application to observe the default container-managed transaction rules applied, and modifying the transaction attributes
- Modifying the `BooksFacadeEJBBean` class to use an EJB client's transactional context. It enables the bean's methods to be invoked by a client's in its own transactional context.

## Practice 15-1: Working with Container-Managed Transactions

In this practice, you open the `Application_15.jws` workspace to examine the container-managed transaction annotations, and run the application with the default container settings.

- 1) Open `Application_15.jws`.
- 2) In the Model project, open the `BooksFacadeEJBBean.java` source in the `org.demo.business` package.
- 3) In `BooksFacadeEJBBean.java`, in the line after the `@Stateful (name = "BooksFacadeEJB", mappedName = "BooksFacadeSessionEJB")` annotation text, add the `@TransactionManagement` annotation with a value of `TransactionManagementType.CONTAINER` as its parameter, and the `@TransactionAttribute` annotation with its parameter set to `TransactionAttributeType.REQUIRED`. Import the required packages to support the annotations. Compile the class.

```
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
...
@Stateful(name = "BooksFacadeEJB", mappedName =
            "BooksFacadeSessionEJB")
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.REQUIRED)

public class BooksFacadeEJBBean implements BooksFacadeEJB,
ProductsFacadeEJBLocal {
    ...
}
```

- 4) Save and compile `BooksFacadeEJBBean`.
- 5) Execute the application.

In the ViewController project, run `ListBooks.jsp`.

In the Add Product window, use the Product ID, Name, and Description fields to insert, update, and delete product data in/from the BOOKS database table by clicking the appropriate button.

- a) Insert a new product with the following details:

Book ID: 200

Book Name: Pride and Prejudice

Genre: Romantic comedy, Novel of manners.

Click **Insert**.

Verify that the new row is visible in the Book List when the page is updated.

- b) The newly inserted record should be visible in the HTML under the Book List heading.

### ***Practice 15-1: Working with Container-Managed Transactions (continued)***

A new record is always visible to the session that inserted it. Check whether the new book has been committed—that is, it is visible to other users. This can be done by executing the following `SELECT` statement in a SQL Worksheet window:

```
SELECT * FROM BOOKS
```

To access the SQL Worksheet, right-click the `fod` database connection on the JDeveloper Database Navigator tabbed page and select **SQL Worksheet**.

- 6) Close the browser window and terminate the Integrated Oracle WebLogic Server in JDeveloper.

**Hint:** In JDeveloper, terminate the WebLogic Server by selecting `Run > Terminate > DefaultServer`.



## Practice 15-2: Working with EJB Client's Transactional Context

In this practice, you execute the session bean's methods within a JSF client's transactional context. The JSF client starts a transaction and calls the session bean's methods within its transactional context. You use

`javax.transaction.UserTransaction`, which is part of the Java Transaction API (JTA) to demarcate the boundaries of a transaction explicitly.

- 1) Open the `BooksFacadeEJBBean.java` source in the `org.demo.business` package.
- 2) In `BooksFacadeEJBBean.java`, change the `@TransactionAttribute` annotation with its parameter set to `TransactionAttributeType.SUPPORTS`. The `SUPPORTS` attribute implies that the bean method inherits the transactional context of the caller. Compile the class.

```
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
...
@Stateful(name = "BooksFacadeEJB", mappedName =
            "BooksFacadeSessionEJB")
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.SUPPORTS)

public class BooksFacadeEJBBean implements BooksFacadeEJB,
ProductsFacadeEJBLocal {
    ...
}
```

- 3) Save and compile `BooksFacadeEJBBean`.
- 4) In the `ViewController` project, open the `ListBooks.java` source in the `org.demo.view.backing` package. In `ListBooks.java` add the following code that implements the JTA transactional APIs to enable the JSF client to start and end a transactional context, and call the `BooksFacadeEJBBean` session bean.
  - a) Declare a private `UserTransaction` instance variable named `userTx`. You need to include an import statement for `javax.transaction.UserTransaction`.

```
...

private List<Books> books;
private BooksFacadeEJB BooksFacade;
private HtmlDataTable dataTable1;
private UserTransaction userTx;
...
```

## **Practice 15-2: Working with EJB Client's Transactional Context (continued)**

- b) Add the code (in bold) in the constructor of the `ListBooks` class to use JNDI to return an `Object` reference to the `UserTransaction` object.

```
...

public ListBooks() {
    InitialContext context = null;
    System.out.println("ListBooks()");
    try {
        context = new InitialContext();
        BooksFacade =

(BooksFacadeEJB) context.lookup("BooksFacadeSessionEJB#org.demo
.business.BooksFacadeEJB");
        userTx =
(UserTransaction) context.lookup("javax.transaction.UserTransac
tion");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

...

```

- c) Add the code (in bold) to start and commit/rollback a transaction for the **`insertBook()`** method, which in turn calls the session bean's `findBookById()` and `persistEntity()` methods.

```
...

public String insertBook() {
    try{
        userTx.begin();
    }
    catch(Exception e){
        addMessage(e.getMessage(), true);
    }
    Books prod = new Books();
    Long bkId =
        Long.parseLong(bookId.getValue().toString());
    prod.setBookId(bkId);
    prod.setName(bookName.getValue().toString());
    prod.setGenre(genre.getValue().toString());

    List<Books> book = null;

```

## Practice 15-2: Working with EJB Client's Transactional Context (continued)

```
        book = BooksFacade.findBookById(bkId);
        if (book.size() == 1){
            try {
                userTx.rollback();
                addMessage("Book with ID " +
                    prod.getBookId() + " already exist", false);
            }
            catch (Exception e){
                addMessage(e.getMessage(), true);
            }
        }
        else{
            try{
                System.out.println ("In insertBook()
                                     method.....");
                BooksFacade.persistEntity(prod);
                bookId.setValue(prod.getBookId());
                userTx.commit();
                addMessage("Book " + prod.getBookId()
                    + " inserted", false);
            } catch (Exception e) {
                addMessage(e.getMessage(), true);
            }
        }

        return "success";
    }
    ...
```

- d) Add the code (in bold) to start and commit/rollback a transaction for the **updateBook()** method, which in turn calls the session bean's `findBookById()` and `mergeEntity()` methods.

```
...
    public String updateBook() {
        List<Books> prods = null;
        Long bookId = (Long)this.bookId.getValue();
        System.out.println("updateBook() bookId: " + bookId);
        try {
            userTx.begin();
            prods = BooksFacade.findBookById(bookId);
            if (prods.size() == 1) {
                Books prod = prods.get(0);
                prod.setName((String)bookName.getValue());
                prod.setGenre((String)genre.getValue());
                BooksFacade.mergeEntity(prod);
                addMessage("Book " + prod.getBookId() + "
                    updated", false);
                userTx.commit();
            }
        }
    }
```

## Practice 15-2: Working with EJB Client's Transactional Context (continued)

```
        }
        else {
            addMessage("Book " + bookId + "
                                not Found", false);
            userTx.rollback();
        }
    }
    catch (Exception e) {
        addMessage(e.getMessage(), true);
    }
    return "success";
}
...
```

- e) Add the code (in bold) to start and commit/rollback a transaction for the **deleteBook()** method, which in turn calls the session bean's **findBookById()** and **mergeEntity()** methods.

```
...

public String deleteBook() {
    List<Books> prods = null;
    Long bookId = (Long)this.bookId.getValue();
    System.out.println("deleteBook() bookId: " + bookId);

    try {
        userTx.begin();
        prods = BooksFacade.findBookById(bookId);
        if (prods.size() == 1) {
            Books prod = prods.get(0);
            BooksFacade.removeBooks(prod);
            addMessage("Book " + prod.getBookId() + "
                                Deleted", false);
            userTx.commit();
        }
        else {
            addMessage("Book " + bookId + " not found",
                                false);
            userTx.rollback();
        }
    } catch (Exception e) {
        addMessage(e.getMessage(), true);
    }
    return "success";
}
...
```

## ***Practice 15-2: Working with EJB Client's Transactional Context (continued)***

- f) Save and compile the JSF backing bean.
- 5) Run ListBooks.jsp.  
  
In the Add Product window, use the Product ID, Name, and Description fields to insert, update, and delete product data in/from the BOOKS database table by clicking the appropriate button.
  - a) Insert a new product with the following details and an existing Book ID:  
  
Book ID: 200  
Book Name: Pride and Prejudice  
Genre: Romantic comedy, Novel of manners.  
  
Click **Insert**.
  - b) Verify that the book is not inserted and you see an appropriate error message on the Web page.
- 6) When you are done, close the browser.
- 7) You can also remove the application from the IDE.

## Practices for Lesson 16

In this practice, you secure a Web application by using JAAS in Oracle WebLogic Server. You perform the following set of tasks in this practice set:

- Enabling form-based login authentication for the Web application using the provided login and error JSP pages
- Restricting access to portions of the Web application by declaring security roles and URL-based security constraints in the application deployment descriptors
- Creating users and groups in WebLogic Server by using the WebLogic Server Administration Console
- Testing the login authentication and authorization functionality of the Web application

## ***Practice 16-1: Enabling Form-based Login Authentication***

In this practice, you enable form-based login authentication for a Web application. You modify the `web.xml` deployment descriptor file by creating the necessary login tags using a JDeveloper interface.

- 1) Open `Application_16.jws`.
- 2) Expand **ViewController > Web Content > WEB-INF** node, right-click `web.xml` select **Properties**.
- 3) In the Web Application Deployment Descriptor window, modify the Login Configuration properties to specify use of form-based authentication using the `Login.jsp` and `Error.jsp` pages in the `infrastructure` folder:
  - a) In the left pane of the window, scroll down and select **Login Configuration**.
  - b) In the right pane, select **Form-Based Authentication** and enter the following values into the Login Page and Error Page fields:

`Login Page: /infrastructure/Login.jsp`  
`Error Page: /infrastructure/Errors.jsp`
  - c) Click **OK**.
  - d) Double-click `web.xml` and click the Source tab to see how those changes are applied to the deployment descriptor properties.

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/infrastructure/Login.jsp</form-login-page>
    <form-error-page>/infrastructure/Errors.jsp</form-error-page>
  </form-login-config>
</login-config>
```

## Practice 16-2: Creating Security Roles and URL Constraints in the Deployment Descriptors

In this practice, you restrict access to portions of the course application by defining JAAS security roles and URL-based security constraints that authorize access based on those security roles.

- 1) Edit the `web.xml` deployment and add the following logical security groups: `myadmin` and `myuser`. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Applications Navigator	Right-click <code>web.xml</code> . Select <b>Properties</b> .
b.	Web Application Deployment Descriptor	On the left pane of the screen select <b>Security Roles</b> .
c.	Security Roles	Click <b>Add</b> .
d.	Create Security Roles	Security Role Name: <code>myadmin</code> Click <b>OK</b> .
e.	Security Roles	Click <b>Add</b> .
f.	Create Security Roles	Security Role Name: <code>myuser</code> Click <b>OK</b> .
g.	Security Roles	Click <b>OK</b> .

JDeveloper added the following XML elements to the `web.xml` deployment descriptor:

```
...
<security-role>
  <role-name>myadmin</role-name>
</security-role>
<security-role>
  <role-name>myuser</role-name>
</security-role>
...
```

- 2) Edit the `web.xml` deployment and add a security constraint called `AdminPages` that allows the `myadmin` security role to access URL matching the pattern: `/admin/edit.jsp`. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Applications Navigator	Right-click <code>web.xml</code> . Select <b>Properties</b> .
b.	Web Application Deployment Descriptor	On the left pane select <b>Security Constraints</b> . Click <b>New</b> (located under the left pane).



## Practice 16-2: Creating Security Roles and URL Constraints in the Deployment Descriptors (continued)

c.	Constraint	Click <b>Add</b> (next to Web Resource Collections field).
d.	Create Web Resource Collections	Web Resource name: AdminPages Click <b>OK</b> .
e.	Constraint	Click <b>Add</b> (next to next to URL Patterns field).
f.	URL Pattern	URL Pattern: /admin/edit.jsp Click <b>OK</b> .
g.	Constraint	Click the <b>HTTP Methods</b> tab. Select <b>Choose</b> , and subsequently the GET, POST check box.
h.	Constraint	Click the <b>Authorization</b> tab. Select the “myadmin” check box. Click <b>OK</b> .

JDeveloper should have added the following XML elements to the web.xml deployment descriptor:

```
...
<security-constraint>
  <web-resource-collection>
    <web-resource-name>AdminPages</web-resource-name>
    <url-pattern>/admin/edit.jsp</url-pattern>
    <http-method>GET</http-method>
  <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>myadmin</role-name>
  </auth-constraint>
</security-constraint>
...
```

- 3) Edit the web.xml deployment and add another security constraint called AllStaff that allows both the myadmin and myuser security roles to access URL matching the pattern: /common/\*. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Applications Navigator	Right-click web.xml. Select <b>Properties</b> .
b.	Web Application Deployment Descriptor	Select <b>Security Constraints</b> . Click <b>New</b> (located under the left pane).
c.	Constraint	Click <b>Add</b> (next to Web Resource Collections field).
d.	Create Web Resource Collections	Web Resource name: AllStaff Click <b>OK</b> .

## Practice 16-2: Creating Security Roles and URL Constraints in the Deployment Descriptors (continued)

e.	Constraint	Click <b>Add</b> (next to next to URL Patterns field).
f.	URL Pattern	URL Pattern: /common/* Click <b>OK</b> .
g.	Constraint	Click the <b>HTTP Methods</b> tab. Select <b>Choose</b> , and subsequently the GET, POST check box.
h.	Constraint	Click the <b>Authorization</b> tab. Select the “myadmin” and “myuser” check boxes. Click <b>OK</b> .

JDeveloper should have added the following XML elements to the web.xml deployment descriptor:

```
...
<security-constraint>
  <web-resource-collection>
    <web-resource-name>AllStaff</web-resource-name>
    <url-pattern>/common/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>myadmin</role-name>
    <role-name>myuser</role-name>
  </auth-constraint>
</security-constraint>
...
```

- 4) Edit the web.xml deployment and add a <welcome-file-list> element. Use the following details to accomplish this task:

Step	Screen/Page Description	Choices or Values
a.	Applications Navigator	Right-click web.xml. Select <b>Properties</b> .
b.	Web Application Deployment Descriptor	Select <b>Welcome File Lists</b> . Click <b>New</b> (located under the left pane).
c.	WelcomeFileList0	Click <b>Add</b> .
d.	Create Welcome File	Welcome File: /index.jsp Click <b>OK</b> .
e.	Web Application Deployment Descriptor	Click <b>OK</b> .

## Practice 16-2: Creating Security Roles and URL Constraints in the Deployment Descriptors (continued)

- 5) Save the `web.xml` file.
- 6) Create the `weblogic.xml` deployment descriptor. In this file, you map the security role names to users and groups. Use the following details to create the `weblogic.xml` deployment descriptor.

Step	Screen/Page Description	Choices or Values
a.	Applications Navigator	Right-click <code>ViewController</code> Select <b>New</b> .
b.	New Gallery	Categories: <b>Deployment Descriptors</b> Items: <b>WebLogic Deployment Descriptor</b> Click <b>OK</b>
c.	Create WebLogic Deployment Descriptor – Step 1 of 4	Select <b>weblogic.xml</b> . Click <b>Next</b>
d.	Create WebLogic Deployment Descriptor – Step 2 of 4	<b>Deployment Descriptor Version: 10.3</b> Click <b>Finish</b> .

- 7) Edit the `weblogic.xml` deployment descriptor to map the `myadmin` and `myuser` security role defined in the `<security-role>` tag in the `web.xml` file.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<weblogic-web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-
web-app.xsd" xmlns="http://www.bea.com/ns/weblogic/weblogic-
web-app">
  <security-role-assignment>
    <role-name>myadmin</role-name>
    <principal-name>chen</principal-name>
    <principal-name>mike</principal-name>
  </security-role-assignment>
  <security-role-assignment>
    <role-name>myuser</role-name>
    <principal-name>joe</principal-name>
  </security-role-assignment>
</weblogic-web-app>
```

- 8) Save the `weblogic.xml` file.

### ***Practice 16-3: Creating the Users and Groups in WebLogic Server***

In this practice, you define the users and groups that will have access to the URL resource. In the `weblogic.xml` file, the `<role-name>` tag defines `myadmin` as the group that has access to the `edit.jsp` file and defines the user `mike` as a member of that group. Therefore, use the WebLogic Server Administration Console to define the `myadmin` group, define user `mike`, and add `mike` to the `myadmin` group. You can also define other users and add them to the group and they will also have access to the protected WebLogic resource.

- 1) Create the `myadmin` and `myuser` group.
  - a) Start the default server in JDeveloper, and open a browser and enter `http://127.0.0.1:7101/console`.
  - b) Log in to the Console using `weblogic/weblogic`.
  - c) In the **Domain Structure** pane, click **Security Realms**.
  - d) On the Summary of Security Realms page, click **myrealm**.
  - e) On the “Settings for myrealm” page, click the Users and Groups > Groups tab.
  - f) Click **New**.
  - g) Enter `myadmin` in the Name field and click **OK**. The `myadmin` group is added to the security realm.
  - h) Execute the same sequence of steps to add the `myuser` group.
- 2) Create the users: `mike`, `chen`, and `joe`.
  - a) In the Domain Structure pane, Click Security Realms.
  - b) On the Summary of Security Realms page, click **myrealm**.
  - c) On the “Settings for myrealm” page, click the Users and Groups > Users tab.
  - d) Click **New**.
  - e) Enter `mike` in the Name field, `welcome1` in the Password and Confirm Password fields, and click **OK**. The user `mike` is added to the security realm.
  - f) Execute the same sequence of steps to add users `chen` and `joe`.
- 3) Add the users to their respective groups: `mike` and `chen` to `myadmin`, and `joe` to the `myuser` group.
  - a) On the “Settings for myrealm” page, click the Users and Groups > Users tab.
  - b) Click the new user, `mike`.
  - c) On the “Settings for mike” page, click the Groups tab.
  - d) Add the `myadmin` parent group from the Available column to the Chosen column.

***Practice 16-3: Creating the Users and Groups in WebLogic Server (continued)***

- e) Click Save.
- f) Execute the same sequence of steps to add the user chen and joe to the myuser groups.

## ***Practice 16-4: Testing the Security Implementation***

In this practice, you deploy the Web application to Oracle WebLogic Server and test its security implementation.

- 1) Run `index.jsp`.

**Note:** You might need to activate the changes in the WebLogic Server by clicking the **Activate Changes** button in WebLogic Server Administration Console. The Activate Changes button is visible when you click the **View changes and restarts** link in the Change Center pane of the administration console.

- 2) Enter the username as `mike` and password as `welcome1`. Click **Sign On**.
- 3) Click the link for the shopping cart page on the Security Login Example page. You see the contents of the shopping cart page.
- 4) Click the **Back** button of the Internet browser window. Click the **Configure Background** link in the Security Login Example page. Mike has the authorization to access the administrative page.

Try executing the same sequence of steps, and log in to the Web application by providing `joe` as the username and `welcome1` as the password. Joe has the privilege to access the shopping cart page, but will get the “Error 403—Forbidden” message when he clicks the **Configure Background** link on the Security Login Example page.