Developing Applications for the Java EE 6 Platform

Activity Guide - NetBeans

D65269GC20
Edition 2.0
June 2011
D73429

ORACLE

Author

Paromita Dutta

Technical Contributors and Reviewers

Tom Mc Ginn, Matthieu Heimer, Greg Stachnick, Moises Lejter

This book was published using: Oracle Tutor

Table of Contents

Practices for Lesson 1: Placing the Java™ EE Model in Context

Chapter 1

Practices for Lesson 1: Overview

Practices Overview

In these practices, you will categorize Java EE services, describe the Java EE platform layers, and explore the existing Java SE **BrokerTool** project.

Practice 1-1: Categorizing Java EE Services

Overview

In this practice, you complete a matching activity to check your understanding of the Java EE service categories.

Tasks

1. Place each Java EE service in the appropriate Java EE service category in the following table:
   - Persistence, Scalability, Naming, Threading
   - Remote Object Communication, Connector, Load Balancing, Failover
   - Security, Life-cycle Services, Transaction, Messaging

No.	Service Category	Java EE Services
1	Deployment-based services	
2	API-based Services	
3	Inherent services	
4	Vendor-specific functionality	

Practice 1-2: Describing the Java EE Platform Layers

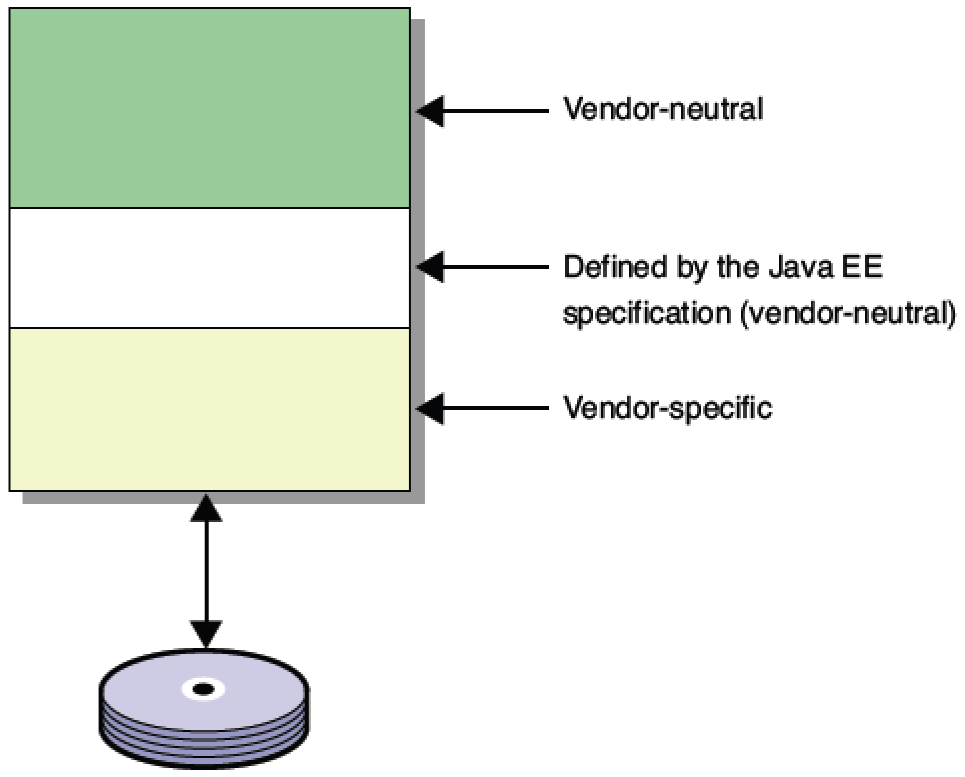
Overview

In this practice, you complete a matching activity to check your understanding of the layers in the Java EE platform.

Tasks

Match the number of each term with the corresponding layer shown in the figure:

a. Databases and other back-end services
b. API layer
c. Service layer
d. Component layer



Practice 1-3: Examining the Java SE Broker Tool Application

Overview

In this practice, you examine the existing **BrokerTool_SE** project. The **BrokerTool_SE** project is a Java Platform, Standard Edition (Java SE) application.

ABC StockTrading is an established stock trading company that manages portfolios for a small set of clientele. ABC StockTrading had an intern develop a prototype Java application to manage their clientele. You have been hired as a Java developer by ABC to further develop the prototype as part of a study to modernize their software by leveraging the power of the Java EE platform.

Assumptions

This practice assumes that NetBeans is installed and the **BrokerTool_SE** project is present on your system.

Tasks

1. Launch NetBeans.
   a. Double-click the desktop shortcut to start NetBeans.
   b. Ensure that the Projects, Files, and Services window are open.
   c. Close the Start Page.
2. Open the BrokerTool_SE project, by selecting Open Project from the File menu.
   - Project Location: `D:\labs\netbeans\projects` folder
   - Project Name: `BrokerTool_SE`
   - Open as Main project: Selected
3. Build the BrokerTool_SE project.
   a. Switch to the Projects window.
   b. Select the BrokerTool_SE node, right-click and select Build from the contextual menu.
4. Run the BrokerTool_SE project.
   a. Select the BrokerTool_SE node, right-click and select Run from the contextual menu.
   b. Using the All Customers tab, view the list of customers. Write down several Customer IDs.
   c. Click the Customer Details tab. Using a Customer ID that you wrote down, complete the Customer Identity field and click the Get Customer button.
   d. Try the other buttons.
   e. Quit the BrokerTool_SE application.

Practice Solutions

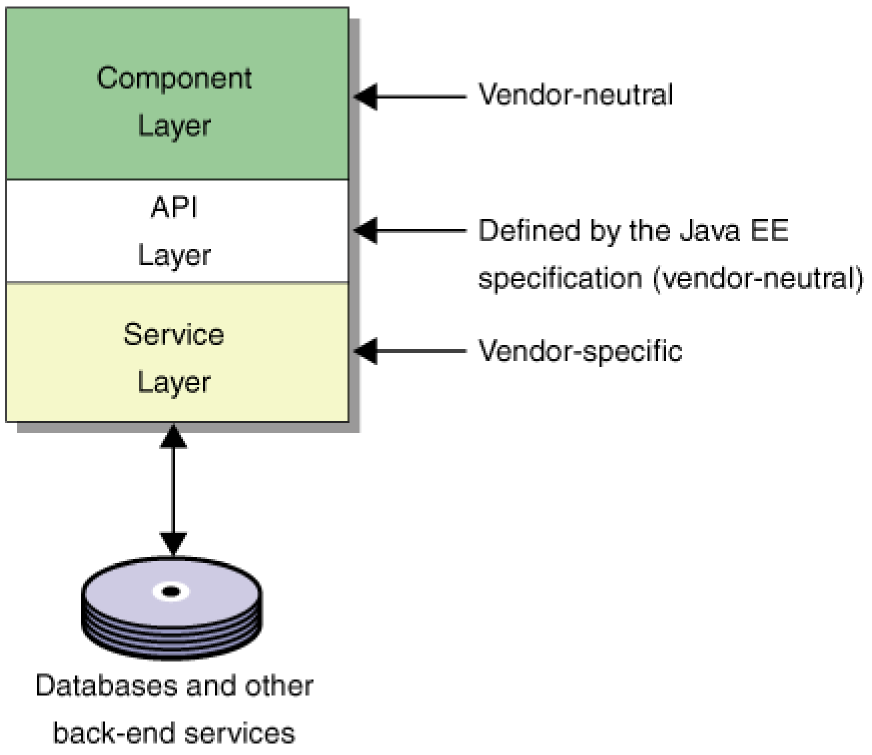
Solution for Practice 1

Compare your answers with the service category and Java EE services shown in the following table:

No.	Service Category	Java EE Services
1	Deployment-based services	Persistence, Transaction, Security
2	API-based Services	Naming, Messaging, Connector
3	Inherent services	Life-cycle services, Threading, Remote object communication
4	Vendor-specific functionality	Scalability, Load balancing, Failover

Solution for Practice 2

Compare your answers to the number and description of the layers of Figure given below


Component Layer
API Layer
Service Layer
Vendor-neutral
Defined by the Java EE specification (vendor-neutral)
Vendor-specific
Databases and other back-end services

Practices for Lesson 2: Java EE Component Model and Development Steps

Chapter 2

Practices for Lesson 2: Overview

Practices Overview

In these practices, you will describe Java EE roles and responsibilities and list the options for packaging applications.

Practice 2-1: Java EE Component Model and Development Steps

Overview

In this practice, you complete a matching activity to check your understanding of the Java EE roles and responsibilities.

Tasks

Match the two columns in the following table to identify the roles that best match each responsibility.

Roles	Responsibility
(1) Application component provider	(1) Resolves references to external resources, and configures the run-time environment of the application
(2) Application assembler	(2) Is the vendor of the application server
(3) Deployer	(3) Maintains and monitors the application server environment
(4) System Administrator	(4) Implements development, packaging, assembly, and deployment tools
(5) Tool provider	(5) Develops EJB components and web components
(6) product provider	(6) Resolves cross-references between components

Practice 2-2: Describing Options for Packaging Applications

Overview

In this practice, you complete a matching activity to check your understanding of the options for packaging applications.

Tasks

For each of the following figures, identify the type of archive file that best describes the packaging option. The archive files include:

- Enterprise archive (EAR) file
- Web archive (WAR) file
- EJB component Java Archive (JAR) file

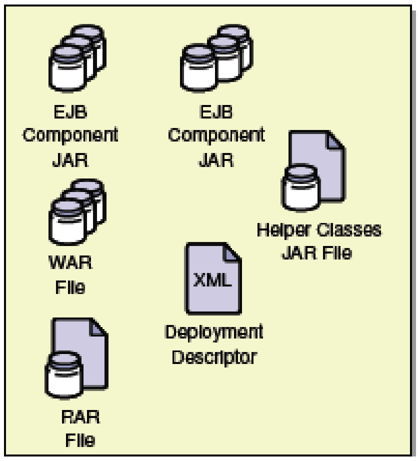


Figure 1

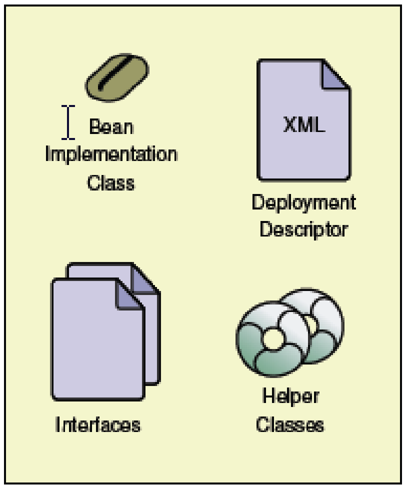


Figure 2

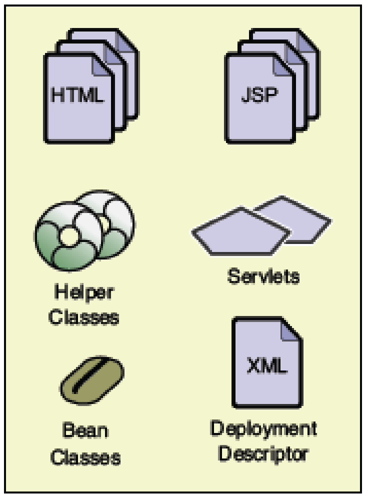


Figure 3

Practice Solutions

Use the following solutions to check your answers to the practices in this lab.

Solutions for Practice 1

This table shows the answers for the Java EE roles and responsibilities matching activity:

No	Role	Responsibility
3	Deployer	(1) Resolves references to external resources, and configures the run-time environment of the application
6	Product Provider	(2) Is the vendor of the application server
4	System Administrator	(3) Maintains and monitors the application server environment
5	Tool Provider	(4) Implements development, packaging, assembly, and deployment tools
1	Application Component Provider	(5) Develops EJB components and web components
2	Application Assembler	(6) Resolves cross-references between components

Solutions for Practice 2

Figure 1 – EAR File
Figure 2 – JAR File
Figure 3 – WAR File

Practices for Lesson 3: Web Component Model

Chapter 3

Practices for Lesson 3: Overview

Practices Overview

In these practices, you will:

- Create a basic JavaServer Pages™ (JSP™) component
- Configure, deploy, and test a web module
- Create a basic servlet
- List the ways that JSP components and servlets fit into the web component model

Practice 3-1: Creating a Basic JSP Component

Overview

In this practice, you create, deploy, and run a web application project with a basic JSP component.

Assumptions

This practice assumes that the application server is installed and configured in NetBeans

Tasks

1. Add GlassFish Server 3.1 to the NetBeans IDE.
   a. In NetBeans, from the menu select *Window* > *Services*.
   b. Right-click *Servers* and click *Add Server...*
   c. Select *GlassFish Server 3*.
   d. Change the name to *GlassFish Server 3.1* to make it clear that you are using this version.
   e. Click *Next*
   f. Browse to *D:\glassfish3* and click Next.
   g. Make sure that *Register Local Domain* is selected, and the domain name is *domain1.*
   h. Click *Finish.*
2. Develop a basic JSP page.
   a. Ensure that NetBeans is running.
   b. In NetBeans, from the menu select *File* > *New Project*.
   c. Under Categories, click Java *Web*.
   d. Under Project, click *Web Application*.
   e. Click the *Next* button.
   f. The new Web Application project should have the following characteristics:
      - Project Name: `SampleWebApplication`
      - Project Location: `D:\Labs\netbeans\projects`
      - Use Dedicated Folder for Storing Libraries: (deselected)
      - Set as Main Project: (selected)
   g. Click the *Next* button.
   h. The "Server and Settings" dialog box should have the following information:
      - Server: GlassFish Server 3.1
      - Java EE Version: Java EE 6 Web
      - Context Path: `/SampleWebApplication`
   i. Click the *Finish* button.
   j. An `index.jsp` file is created for you automatically. You can place any static HTML in a JSP page. Experiment with adding Java code to the JSP page.
   k. The following is an example of what you might enter:
      ```
      <%= new java.util.Date() %>
      ```

3. Deploy and test the sample application.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **SampleWebApplication** Web project by right-clicking the project folder in NetBeans and selecting *Deploy*.
   b. Test the application by right-clicking the project folder and selecting *Run*, or by pointing a web browser at:
   `http://localhost:8080/SampleWebApplication/index.jsp`

Practice 3-2: Troubleshooting a Web Application

Overview

In this practice, you introduce an error into the SampleWebApplication project. Then, redeploy the application and review the errors generated by your change to the code.

Assumptions

This practice assumes that the previous exercise has been completed.

Tasks

1. Create a faulty web component.
    a. Ensure that the `SampleWebApplication` project is open in NetBeans.
    b. Modify the `index.jsp` web component to produce an exception upon execution. Enter the following into your `index.jsp` page:

```
<%
    Object o = null;
    o.toString();
%>
```

2. Deploy and test the faulty application.
    a. Save any modified files. If Deploy on Save is not enabled, deploy the **SampleWebApplication** Web project manually.
    b. Test the application by selecting *Run* or pointing a web browser at:

```
http://localhost:8080/SampleWebApplication/index.jsp
```

    c. Notice the error message displayed in your web browser.

3. View the Application Server error messages.
    a. Bring up the Application Server log in the IDE.
        1) In the Services window, expand the Servers node.
        2) Right-click the GlassFish Server 3.1 node and select **View Server Log**.
        3) The log will be displayed in a *GlassFish Server 3.1* tab in the output pane.
    b. Find where the web application causes an exception to be generated. The line should be similar to:

```
Servlet.service() for servlet jsp threw exception
java.lang.NullPointerException
```

When looking for errors in the log file it may be helpful to clear the log to ensure that you are only viewing recent errors.

4. Remove the error placed in the `index.jsp` file during Task 1 and redeploy the application.

Practice 3-3: Creating a Basic Servlet Component

Overview

In this practice, you create a basic servlet component in a web application.

Assumptions

This practice assumes that the application server is installed and running along with NetBeans.

Tasks

1. Create a servlet
   a. Ensure that the **SampleWebApplication** project is open in **NetBeans.**
   b. Right-click the **SampleWebApplication** project select New > Other.
   c. Select Web from Categories.
   d. Select Servlet from File Types.
   e. Enter for the following information for the servlet:
      - Class Name: **BasicServlet**
      - Project**: SampleWebApplication**
      - Location: **Source Packages**
      - Package: **`test`**
   f. Click *Next*. Do NOT check the box to *Add information to deployment descriptor*. By leaving the box deselected you are instructing the IDE to add configuration annotations. This eliminates the need for the `web.xml` configuration file.
   g. Click *Finish*.
   h. Modify the `processRequest` method of the `BasicServlet` servlet to display a dynamically generated message.
      - Remove the comments surrounding the `out.println()` statements.
      - Modify your servlet using the following code as a guide:

```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {

        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet BasicServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet BasicServlet at " +
request.getContextPath() + "</h1>");
        out.println("Generated at: " + new java.util.Date());
```

```
            out.println("</body>");
            out.println("</html>");

        } finally {
            out.close();
        }
    }
```

i. Show the HTTP methods hidden below the `processRequest` method. Verify that both the `doGet` and `doPost` methods call the `processRequest` method.

j. Ensure that the `BasicServlet` class has a `@WebServlet` annotation at the class level. The annotation should have a `urlPatterns` attribute that specifies the URLs for the servlet. It should look like:

```
@WebServlet(name="BasicServlet", urlPatterns={"/BasicServlet"})
public class BasicServlet extends HttpServlet {
```

k. The `urlPatterns` indicate URLs that are relative to the context root for the application. Thus, you can access the servlet using the `http://localhost:8080/SampleWebApplication/BasicServlet` URL in your web browser.

2. Deploy and test the application.

   a. Save any modified files. If Deploy on Save is not enabled, deploy the **SampleWebApplication** Web project manually.

   b. Test the application by selecting *Run* or by entering the following URL in a browser: `http://localhost:8080/SampleWebApplication/BasicServlet`

   c. You should see a dynamically generated web page. This indicates that the servlet has been successfully invoked.

Practice 3-4: Describing Web Components

Overview

In this practice, you complete a fill-in-the-blank activity to check your understanding of web components.

Task

Fill in the blanks of the following sentences with the missing word or words:

1. At runtime, JSP components are essentially just ______________.
2. JSP components and servlets are packaged into a web application, along with any static content that is required. The web application is deployed in a ___________ file.
3. ______________________ are useful for generating presentation, particularly HTML and XML. _____________, on the other hand, are useful for processing form data, computation, and collecting data for rendering.
4. The web container calls the _________________ method once for each incoming request.
5. Because HTTP is _______________, the server cannot ordinarily distinguish between successive requests from the same browser and a single request from different browsers.
6. The two most common HTTP request types that are used with servlets are ____________ and ___________.
7. In the HTTP model, a client sends a ____________ to a server and receives a ________________ from the server

Practice Solutions

This section contains the practice solutions.

Solutions for Practice 1, 2, and 3

You can find solutions for the practices in this lab in the following directory:
`D:\labs\netbeans\solutions\WebComponents`

Solution for Practice 4: Describing Web Components

Compare your fill-in-the-blank responses with the following answers:

1. At runtime, JSP components are essentially just *servlets*.
2. JSP components and servlets are packaged into a web application, along with any static content that is required. The web application is deployed in a *WAR* file.
3. *JSP components* are useful for generating presentation, particularly HTML and XML. *Servlets*, on the other hand, are useful for processing form data, computation, and collecting data for rendering.
4. The web container calls the *`service()`* method once for each incoming request.
5. Because HTTP is *stateless*, the server cannot ordinarily distinguish between successive requests from the same browser and a single request from different browsers.
6. The two most common HTTP request types that are used with servlets are *`GET`* and *`POST`*.
7. In the HTTP model, a client sends a *request* to a server and receives a *response* from the server.

Practices for Lesson 4: Developing Servlets

Chapter 4

Practices for Lesson 4: Overview

Practices Overview

In these practices, you will:

- Create a Java EE web application project in NetBeans
- Create servlets to dynamically process form data
- Describe controller components

Practice 4-1: Developing the BrokerTool Web Application

Overview

In this practice, you create the BrokerTool Web Application, copy provided Java classes and create the `CustomerDetails` servlet the default index page.

Assumptions

This practice assumes that the application server is installed and configured in NetBeans.

Tasks

1. Create the BrokerTool project.
   a. In NetBeans, from the menu select *File* then *New Project*.
   b. Under Categories, click Java *Web*.
   c. Under Project, click *Web Application*.
   d. Click the *Next* button.
   e. The new Web Application project should have the following characteristics:
      - Project Name: **`BrokerTool`**
      - Project Location: `D:\Labs\netbeans\projects`
      - Use Dedicated Folder for Storing Libraries: (deselected)
      - Set as Main Project: (selected)
   f. Click the *Next* button.
   g. The "Server and Settings" dialog box should have the following information:
      - Server: GlassFish Server 3.1
      - Java EE Version: Java EE 6 Web
      - Enable Contexts and Dependency Injection (deselected)
      - Context Path: `/BrokerTool`
   h. Click the *Finish* button.
   i. An `index.jsp` file is created for you automatically. You will not use the `index.jsp` file in this project, delete it.
2. Copy the BrokerTool SE classes.
   a. Right-click the **BrokerTool** project icon.
   b. Select *New* and then *Other*
   c. Select *Java* from Categories
   d. Select *Java Package* from File Types
   e. Enter `trader` for the package name.
   f. Click the *Finish* button.

g. Copy classes from the **BrokerTool_SE** project to the `trader` package in the **BrokerTool** project.
   1) Open the **BrokerTool_SE** project.
   2) Expand Source Packages and then trader.
   3) To copy a class, right-click the class you want to copy in the **BrokerTool_SE** `trader` package and select *Copy*. Next, right-click the target *trader* package in the **BrokerTool** project and select *Paste* and then *Refactor Copy*. A confirmation dialog box appears. Choose *Refactor*. The copy of the class is complete.

   Copy the following classes:

   `BrokerException.java`

   `BrokerModel.java`

   `BrokerModelImpl.java`

   `Customer.java`

   `CustomerShare.java`

   `Stock.java`

h. Close the **BrokerTool_SE** project.

3. Copy the `CusomerDetails` servlet
   a. Right-click the **BrokerTool** project icon.
   b. Select *New* then *Java Package*.
   c. Enter `trader.web` for the package name. Click Finish.
   d. Enable the Favorites tab if it is not displayed already. From the *Window* menu select *Favorites*.
   e. Add the `D:\Labs\netbeans\resources` directory to the *Favorites* window if it is not already present. Right-click an empty area in the Favorites window and select *Add to Favorites*. In the “file chooser” dialog box find and select the `D:\Labs\netbeans\resources` directory.

   You can also add the `D:\Labs\netbeans\solutions` directory to the *Favorites* window for easy access to lab solution files. The solutions are also complete projects that can be opened in NetBeans.

   f. In the *Favorites* window, copy the `CustomerDetails.java` file from *resources* > *brokertool* to the clipboard.
   g. In the *Projects* window, paste the `CustomerDetails.java` file into the `trader.web` package of the **BrokerTool** project.
   h. View the source code for the `CustomerDetails` servlet. When accessed with a web browser this servlet will present an empty HTML form. The `CustomerDetails` servlet is designed to function as a MVC view. You will implement the corresponding MVC controller later in this lesson.

4. Set the **Broker Tool** default home page
   a. Right-click the **BrokerTool** project icon.
   b. Select *New* and then *Other.*
   c. Select *Web* from Categories.
   d. Select Standard Deployment Descriptor (web.xml) from File Types.

e. Click Next and Finish. The `web.xml` deployment descriptor should be created and opened.

f. Switch to the *Pages* view of the `web.xml` deployment descriptor.

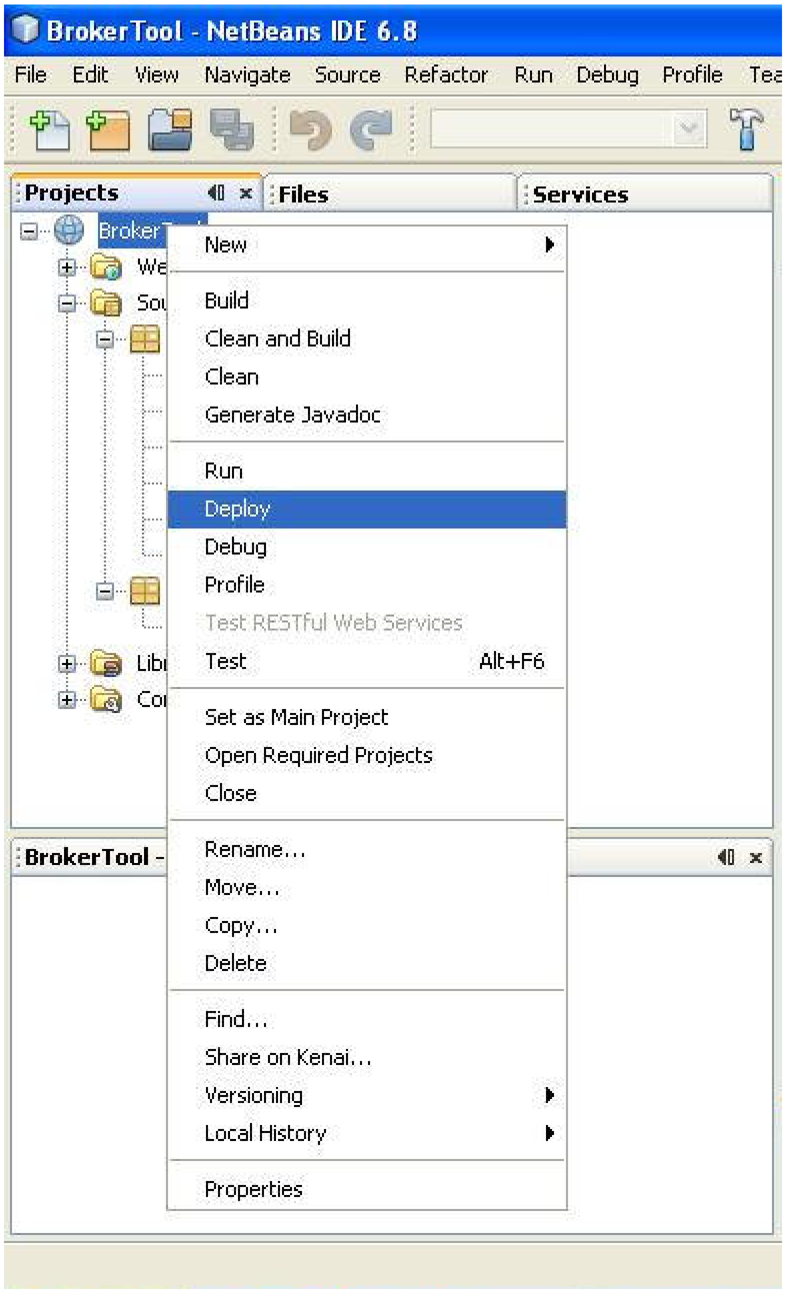
g. Enter a value of **CustomerDetails** for *Welcome Files*.

h. Switch to the *XML* view to see the changes to the deployment descriptor. You should see:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>CustomerDetails</welcome-file>
    </welcome-file-list>
</web-app>
```

5. Configure, deploy, and test the application.

a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually as shown in the following screenshot.



b. Run the application by pointing a browser at:
`http://localhost:8080/BrokerTool/`

c. You should see an empty customer details form. At this point none of the functionality is implemented.

Practice 4-2: Implementing Controller Components

Overview

In this practice, you create the `CustomerController` and `PortfolioController` servlet.

Assumptions

This practice assumes that the application server is installed and the previous practice has been completed.

Tasks

1. Create the `CustomerController` servlet.
   a. Create a new servlet in the **BrokerTool** project. In the project window, right-click BrokerTool and select New > Servlet.
   b. In the "Name and Location" dialog box, enter the following information:
      - Class Name: `CustomerController`
      - Location: Source Packages
      - Package: `trader.web`
   c. Click the *Finish* button.
   d. Update the `@WebServlet` annotation and the `urlPatterns` attribute make the servlet available at two URLs:
      - `/CustomerController`
      - `/AllCustomers`
   e. Add an import statement for the `trader` package.
   f. Import `javax.servlet.RequestDispatcher`.
   g. Use the following steps to code the `processRequest` method:
      1) Remove all code from the `processRequest` method.
      2) Retrieve the singleton instance of the `BrokerModelImpl` class by adding the following line:

```
BrokerModel model = BrokerModelImpl.getInstance();
```

      3) Use the `HttpServletRequest` object to get the path used to invoke the servlet:

```
String path = request.getServletPath();
```

      4) If the path was `/CustomerController`, perform the following steps:
         a) Retrieve the request form parameter values for `customerIdentity`, `customerName`, `customerAddress` and `submit`. Assign the values to string variables `id`, `name`, `address`, and `submit`.
         b) Use the value of the `submit` variable to determine whether one of the submit buttons in the `CustomerDetails` servlet was clicked to invoke this servlet, perform the following actions if so:

c) If the *Get Customer* submit button was clicked, use the `model` to look up the customer with the `customerIdentity` request parameter. Store the customer as a request attribute named `customer` (case sensitive).

d) If the *Update Customer* submit button was clicked, use the `model` to update the customer with the ID of the `customerIdentity` request parameter to have the values of `customerName` and `customerAddress`. Retrieve and store the updated customer as a request attribute named `customer` (case sensitive).

e) If the *Add Customer* submit button was clicked, use the `model` to create a new customer with the `customerIdentity`, `customerName`, and `customerAddress` request parameters. Retrieve and store the new customer as a request attribute named `customer` (case sensitive).

f) If the *Delete Customer* submit button was clicked, use the `model` to delete the customer with the ID of the `customerIdentity` request parameter.

g) Use exception handling to deal with any errors that might occur when using the `model` variable. If exceptions occur, call the `Exception` class `getMessage` method and store the value in a request attribute named `message`.

h) Use a `RequestDispatcher` to forward the response to the `CustomerDetails` servlet.

```
RequestDispatcher dispatcher =
request.getRequestDispatcher("CustomerDetails");
dispatcher.forward(request, response);
```

5) If the path `/AllCustomers` was used to invoke this servlet, perform the following steps:

a) Use the `model` variable to retrieve an array of all customers.

b) Store the array of all customers as a request attribute named `customers`.

c) Use exception handling to deal with any errors that might occur when using the `model` variable. If exceptions occur, call the `Exception` class `getMessage` method and store the value in a request attribute named `message`.

d) Use a `RequestDispatcher` to forward the response to `AllCustomers.jsp`.

```
RequestDispatcher dispatcher =
request.getRequestDispatcher("AllCustomers.jsp");
dispatcher.forward(request, response);
```

2. Create the `PortfolioController` servlet that is designed to retrieve a customer's portfolio and forward that data to a `Portfolio.jsp` for display.

a. Create a new Servlet in the **BrokerTool** project.

b. In the "Name and Location" dialog box, enter the following information:

- Class Name: **`PortfolioController`**
- Location: Source Packages
- **Package: `trader.web`**

c. Click the *Finish* button.

d. Make sure that the `@WebServlet` annotation sets the URL of the `PortfolioController` servlet to `/PortfolioController`.

e. Add an import statement for the `trader` package.

f. Import `javax.servlet.RequestDispatcher`.

g. Implement the `processRequest` method as follows:

You can use the `PortfolioController` template located at
`D:\labs\netbeans\resources\brokertool\PortfolioController.java`

```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)throws ServletException,
IOException {

  String customerId = request.getParameter("customerIdentity");

  BrokerModel model = BrokerModelImpl.getInstance();

  try {
     CustomerShare[] shares =
model.getAllCustomerShares(customerId);
     Customer customer = model.getCustomer(customerId);
     request.setAttribute("shares", shares);
     request.setAttribute("customer", customer);
      } catch (BrokerException be) {
          request.setAttribute("message", be.getMessage());
      }
      RequestDispatcher dispatcher =
request.getRequestDispatcher("Portfolio.jsp");
        dispatcher.forward(request, response);
}
```

3. Configure, deploy, and test the application.

a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.

b. Run the application or test the servlet by entering the following URL in a browser: `http://localhost:8080/BrokerTool/`

c. You will see an empty customer details form. You should be able to enter a known customer identity, such as 111-11-1111, and retrieve information for that customer. Try all the buttons in the customer details page. Links such as *View Portfolio*, *All Customers*, and *Stocks* will not function yet. Fix any errors that occur when using the Customer Details form.

Practice 4-3: Describing Servlet Components

Overview

In this practice, you complete a fill-in-the-blank activity to check your understanding of web components.

Task

1. Fill in the blanks of the following sentences with the missing word or words:
   a. The _____________________________ package contains the HTTP- specific servlet classes.
   b. The object type _______________ is representative of the storage area that is provided by a Java EE web container for managing sessions in the web component model.
   c. The method signature of the initialization method inherited from `HttpServlet`, which is recommended for use is ______________.
   d. The __________________ method typically calls the `doGet` or `doPost` method.
   e. In the `WEB-INF` directory of a web application, a configuration file named ________________ is used to configure the application.
   f. In Java EE 6, the __________________ annotation can be used in place of the `init` method for a servlet.
   g. To read form data, the ____________________ method of an `HttpServletRequest` is used.
   h. __________________ class has a forward and include method used to invoke a servlet from within another servlet.
   i. In Java EE 6, the __________________ annotation can be used in-place of the deployment descriptor to specify a URL for a servlet.
   j. Every time a `request.getSession` method is called, the server attempts to send a ________________ to the client.

Practice Solutions

This section contains the practice solutions.

Solutions for Practice 1 and 2

You can find solutions for the practices in this lab in the following directory: `D:\labs\netbeans\solutions\Servlets.`

Solutions for Practice 3

Compare your fill-in-the-blank responses with the following answers:

a. The *`javax.servlet.http`* package contains the HTTP-specific servlet classes.

b. The object type *`HTTPSession`* is representative of the storage area that is provided by a Java EE web container for managing sessions in the web component model.

c. The method signature of the initialization method inherited from *`HttpServlet`*, which is recommended for use is `init()`.

d. The *`service(HttpServletRequest, HttpServletResponse)`* method typically calls the `doGet` or `doPost` method.

e. In the WEB-INF directory of a web application, a configuration file named *`web.xml`* is used to configure the application.

f. In Java EE 6, the *`@PostConstruct`* annotation can be used in place of the `init` method for a servlet.

g. To read form data, the *`getParameter("name")`* method of an `HttpServletRequest` is used.

h. *`RequestDispatcher`* class has a `forward` and `include` method used to invoke a servlet from within another servlet.

i. In Java EE 6, the *`@WebServlet`* annotation can be used in-place of the deployment descriptor to specify a URL for a servlet.

j. Every time the `request.getSession` method is called, the server attempts to send a *cookie* to the client.

.

Practices for Lesson 5: Developing with JavaServer Pages Technology

Chapter 5

Practices for Lesson 5: Overview

Practices Overview

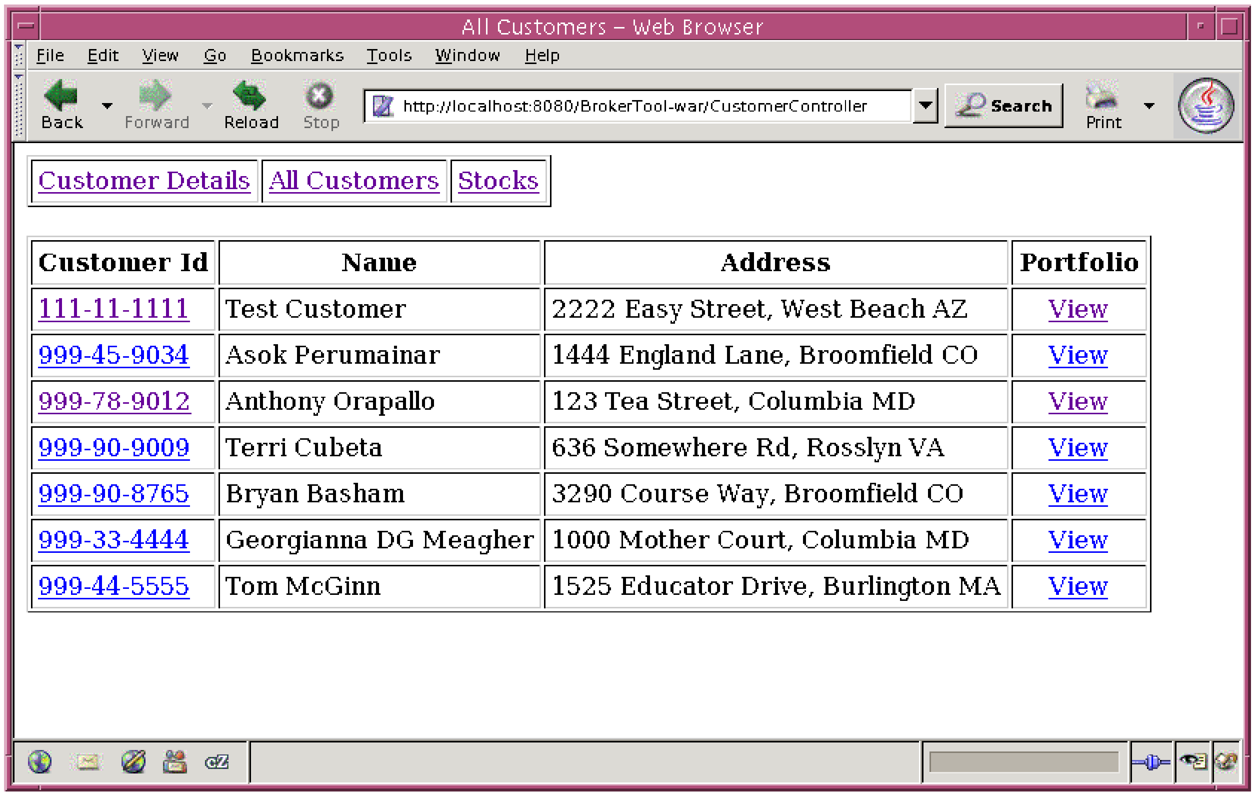
In these practices, you will:

- Create a JSP component
- Use JSP scriptlet tags
- Use JSTL tags
- Use the Expression Language (EL)

Practice 5-1: Creating the `AllCustomers.jsp` Component

Overview

In this practice, you will create the `AllCustomers.jsp` component that displays the identities, names, and addresses of all customers registered in the **BrokerTool** application as shown in the following figure.


All Customers – Web Browser
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://localhost:8080/BrokerTool-war/CustomerController
Search
Print
Customer Details | All Customers | Stocks

Customer Id	Name	Address	Portfolio
111-11-1111	Test Customer	2222 Easy Street, West Beach AZ	View
999-45-9034	Asok Perumainar	1444 England Lane, Broomfield CO	View
999-78-9012	Anthony Orapallo	123 Tea Street, Columbia MD	View
999-90-9009	Terri Cubeta	636 Somewhere Rd, Rosslyn VA	View
999-90-8765	Bryan Basham	3290 Course Way, Broomfield CO	View
999-33-4444	Georgianna DG Meagher	1000 Mother Court, Columbia MD	View
999-44-5555	Tom McGinn	1525 Educator Drive, Burlington MA	View

Assumptions

This practice assumes that the application server is installed and the previous lesson's **BrokerTool** practice has been completed.

Tasks

1. Create the `AllCustomers.jsp` component.
    a. On the Projects window, right-click the **BrokerTool** icon.
    b. Select *New* then *Other*
    c. Under Categories, click *Web*.
    d. Under File Types, click *JSP*.
    e. Enter the following information in the "Name and Location" dialog box:
        - File Name: `AllCustomers`
        - Location: Web Pages

- Folder: (empty)
- Options: JSP File (Standard Syntax)

f. Click the *Finish* button.

g. Modify the `AllCustomers.jsp` component to ensure the following behavior:

1) Add an import directive to import the classes in the `trader` package.
2) Make the page title *All Customers*.
3) Create a table based navigational menu along the top that includes links to:

```
<a href='CustomerDetails'>Customer Details</a>
<a href='AllCustomers'>All Customers</a>
<a href='Stocks.xhtml'>Stocks</a>
```

4) Create a table with the following headers: `Customer Id`, `Name`, `Address`, and `Portfolio`.
5) Using a scriptlet tag, create a `customers` array variable of type `Customer[]`. Retrieve the array data from `customers` attribute stored in the request scope and assign it to the `customers` array. This attribute was created by the `CustomerController`.
6) Create a `for` loop to iterate through all the customers. For each iteration, display a table row with a customer's ID, name, address, and a link to view the customer's portfolio. Use JSP expression tags to display the data. The following is a sample portfolio link:

```
<a href='PortfolioController?customerIdentity=<%=
customers[i].getId() %>'>View</a>
```

7) Close the table.
8) Using scriptlet tags, display the message stored in the request scope under the attribute name `message`.

```
<%
    String message =
              (String)request.getAttribute("message");
    if(message != null) {
        out.println("<font color='red'>" + message +
                    "</font>");
    }
%>
```

2. Configure deploy and test the application.

a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.

b. Run the application. Test your JSP by entering the following URL in a browser:
`http://localhost:8080/BrokerTool/AllCustomers`

c. You should see a table of all customers. Fix any errors that occur.

Note: The View links in the All Customers page and the View Portfolio link in the Customer Details page will fail. You will fix this in the next practice.

Practice 5-2: Creating the `Portfolio.jsp` Component

Overview

In this practice, you create the `Portfolio.jsp` component, which displays the symbols and quantities of stocks owed by the customer most recently selected in the `CustomerDetails` page.

Assumptions

This exercise assumes that the application server is installed and the previous **BrokerTool** exercise has been completed.

Tasks

1. Create `Portfolio.jsp` component in BrokerTool.
   a. In the Projects window, select BrokerTool, right-click and select *New* > *JSP*. Enter the following information in the "Name and Location" dialog box:
      - JSP File Name: **Portfolio**
      - Location: **Web Pages**
      - Folder: **(empty)**
      - Options: **JSP File (Standard Syntax)**

      Click the Finish button.
   b. Add a page import directive to import the classes in the `trader` package.
   c. Add a directive to include the Java Standard Tag library, for example:
      `<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>`
   d. Give the page a title of *Portfolio*.
   e. Create a table based navigational menu along the top that includes links to:

```
<a href='CustomerDetails'>Customer Details</a>
<a href='AllCustomers'>All Customers</a>
<a href='Stocks.xhtml'>Stocks</a>
```

   f. Create a JSTL `<c:choose>` tag. Inside the `<c:choose>` create a `<c:when test="${...}">` tag and a `<c:otherwise>` tag.
   g. The test in the `<c:when test="${...}">` should use the EL to test if there is a message stored in the `requestScope. ${requestScope.message == null}`
      1) If there is no message :
         a) Display a message with the customer's name:

```
${requestScope.customer.name}'s Stocks
     <br />
```

         b) Create a table with Stock Symbol and Quantity as headers.

c) Use JSTL and EL to display the `CustomerShare[]` array stored in the request scope by the `PortfolioController`.

```
<c:forEach var="share" items="${requestScope.shares}">
    <tr>
        <td>${share.stockSymbol}</td>
        <td>${share.quantity}</td>
    </tr>
</c:forEach>
```

2) If there is a message, display it inside the `<c:otherwise>` tags using the EL to read the string stored under the attribute name of the message in the request scope.

2. Configure, deploy, and test the application.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.
   b. Run the application. Test your JSP by entering the following URL in a browser:
      `http://localhost:8080/BrokerTool/`

   Get the details of a customer with shares. You can read `BrokerModelImpl.java` to find a customer with shares or use 123-45-6789. After getting the details for a customer, use the *View* link under *Portfolio* column to view the customer's portfolio.

   You should see a table of all shares for a customer. Fix any errors that occur.

Practice 5-3: Optional: Creating the `CustomerDetails.jsp` Component

Overview

This exercise is optional and provides fewer instructions in order to provide a challenge to more advanced students. Ask your instructor if you have time to complete this exercise.

In this practice, you create the `CustomerDetails.jsp` component that displays the identities, names, and addresses of all customers registered in the **BrokerTool** Application.

Assumptions

This practice assumes that the application server is installed and the previous **BrokerTool** practice has been completed.

Tasks

1. Create the `CustomerDetails.jsp` component.
   a. Create a new JSP component in the **BrokerTool** project. In the Projects window, right click BrokerTool and select *New > JSP.*
   b. Enter the following information in the "Name and Location" dialog box:
      - JSP File Name: **CustomerDetails**
      - Location: **Web Pages**
      - Folder: **(empty)**
      - Options: **JSP File (Standard Syntax**)

      Click the Finish button
   c. View the details of a customer by using the existing `CustomerDetails` servlet at `http://localhost:8080/BrokerTool/`.
   d. View the HTML output of the `CustomerDetails` servlet using your web browser.
   e. View the behavior of the `CustomerController` servlet.
   f. Using the information gathered in step d and e, code the functionality of `CustomerDetails.jsp`. Use JSTL and EL, and avoid scriptlets.
   g. Modify the `CustomerController` servlet to forward to the new JSP page.
   h. Modify the default welcome page in the `web.xml` deployment descriptor.
   i. In all JSP pages, modify any links that point to the `CustomerDetails` servlet to point to `CustomerDetails.jsp`
2. Configure, deploy, and test the application.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.
   b. Run the application. Test your JSP by entering the following URL in a browser:
      `http://localhost:8080/BrokerTool/`

Practice 5-4: Describing JavaServer Pages Components

Overview

In this practice, you answer the question or complete a fill-in-the-blank activity to check your understanding of JSP components.

Tasks

Answer the question or fill in the blanks of the following sentences with the missing word or words:

1. True or False: A JSP typically has fewer lines of Java code than HTML.
2. A typical `scriptlet` tag starts with a ______ and ends with a ______.
3. To import the classes in the `java.util` package in a JSP, you would add ________________________________ to the JSP.
4. In place of `scriptlet` code, a `jsp:useBean` tag in the form of ________________________________ could be used to locate a `Customer` object stored with the `HttpServletRequest.setAttribute("Customer", cust)` method.
5. Java EE has a pre-written set of custom tag libraries known as the _________.
6. The ____________________ is the name of the new JavaScript-like language that is executed during the server-side execution of a JSP.

Practice Solutions

This section contains the practice solutions.

Solutions for Practice 1, 2, and 3

You can find solutions for the practices in this lab in the following directory:
`D:\labs\netbeans\solutions\JSPs`

Solutions for Practice 4: Describing JavaServer Pages Component

Compare your responses with the following answers:

1. *True*: A JSP typically has fewer lines of Java code than HTML.
2. A typical `scriptlet` tag starts with a *<%* and ends with a *%>*.
3. To import the classes in the `java.util` package in a JSP you would add *<%@ page import="java.util.*" %>* to the JSP.
4. In place of scriptlet code, a `jsp:useBean` tag in the form of *<jsp:useBean id="Customer" scope="request" />* could be used to locate a `Customer` object stored with the `HttpServletRequest.setAttribute("Customer", cust)` method.
5. Java EE has a pre-written set of custom tag libraries known as the *JSTL*.
6. The *Expression Language (EL)* is the name of the new JavaScript-like language that is executed during the server-side execution of a JSP.

Practices for Lesson 6: Developing with JavaServer Faces™ Technology

Chapter 6

Practices for Lesson 6: Overview

Practices Overview

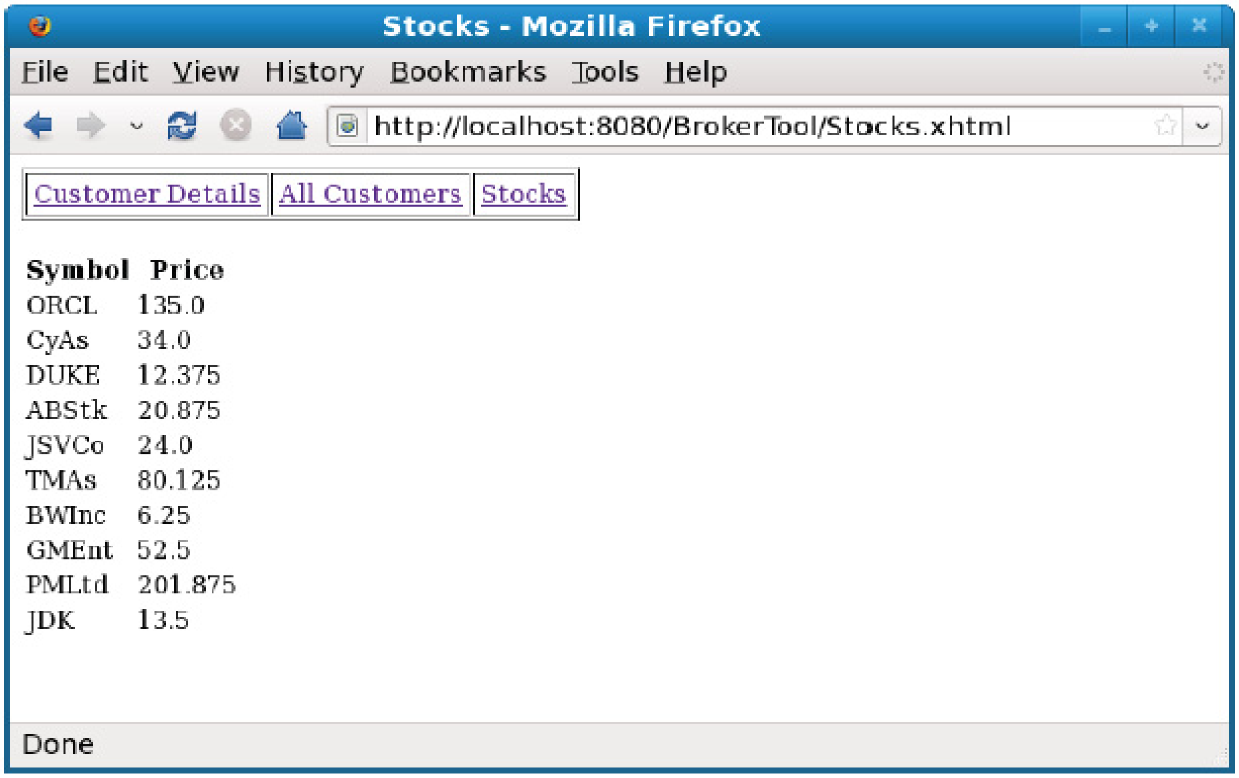
In these practices, you will:

- Enable the JSF framework in a Java EE application
- Create a JSF facelet page
- Create a JSF managed bean
- Use JSF tags
- Use the Expression Language (EL) with managed beans

Practice 6-1: Creating the `Stocks.xhtml` Component

Overview

In this practice, you create the `Stocks.xhtml` component that displays the names, and prices of all stocks in the **BrokerTool** Application as seen in the following figure:


Stocks - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/BrokerTool/Stocks.xhtml
Customer Details
All Customers
Stocks
Done

Symbol	Price
ORCL	135.0
CyAs	34.0
DUKE	12.375
ABStk	20.875
JSVCo	24.0
TMAs	80.125
BWInc	6.25
GMEnt	52.5
PMLtd	201.875
JDK	13.5

Assumptions

This practice assumes that the application server is installed and the previous **BrokerTool** exercise has been completed.

Tasks

1. Configure the `JSF` facelet servlet.
   a. Open the `web.xml` file
   b. Modify the deployment descriptor to add a new Servlet element. Navigate to the Sevlets tab and provide the following details:
      - Servlet Name: `Faces Servlet`
      - Servlet Class: `javax.faces.webapp.FacesServlet`
      - URL Pattern(s): `*.xhtml`

      Click OK. This creates a servlet entry in NetBeans. The configuration form for this servlet is displayed in the NetBeans.
   c. In the Faces Servlet form, set the Startup Order to 1.

   d. Navigate to XML tab to view source.
   e. The following XML `servlet` and `servlet-mapping` tags are included in the `web.xml` deployment descriptor:

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-
class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

2. Create the `StocksManagedBean` JSF component.
   a. In the Projects window, right-click the **BrokerTool** icon.
   b. Select *New* and then *Other*
   c. Under Categories, click *JavaServer Faces*
   d. Under File Types, click *JSF Managed Bean*
   e. Click Next.
   f. Enter the following values in the “Name and Location” dialog box.
      - Class Name: **`StocksManagedBean`**
      - Project: **`BrokerTool`**
      - Location: **Source Packages**
      - Package: **`trader.web`**
      - Name: **`stocks`**
      - Scope: **request**
   g. Click the *Finish* button.
   h. Add an instance variable of type `BrokerModel` in the `StocksManagedBean` class.
      `private BrokerModel model = BrokerModelImpl.getInstance();`
   i. Declare the following method:
      `public Stock[] getAllStocks() { }`
   j. Add any required import statements.
   k. Implement the `getAllStocks` method as follows:
      - Retrieve an array of all `Stock` objects using the `model` variable.
      - Catch any exceptions that occur. Return `null` if an Exception occurs.
3. Create the `Stocks.xhtml` facelet page.
   a. In the Projects window, right-click the **BrokerTool** icon.
   b. Select *New* and then *Other*
   c. Under Categories, click *JavaServer Faces*
   d. Under File Types, click *JSF Page*
   e. Click Next. Enter the following values in the “Name and Location” dialog box:

- File Name: `Stocks`
- Project: `BrokerTool`
- Location: **Web Pages**
- Folder: (empty)
- Options: **Facelets** (selected)

f. Click the *Finish* button.

g. Add the core JSF tags to the `Stocks.xhtml` page:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
```

h. Modify the `Stocks.xhtml` page as follows:

- Specify a value of *Stocks* for the page title.
- Delete the content of the existing body tag.

i. Create a table-based navigational menu along the top that includes links to:

```
<a href='CustomerDetails'>Customer Details</a>
<a href='AllCustomers'>All Customers</a>
<a href='Stocks.xhtml'>Stocks</a>
```

j. Use the `h:dataTable` tag to display the symbol and price of all stocks.

4. Configure, deploy, and test the application.

a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.

b. Run the application. Test your JSF page by entering the following URL in a browser:
`http://localhost:8080/BrokerTool/Stocks.xhtml`

You should see a table of all stocks. Fix any errors that occur.

Optional Practice 6-2: Implementing the JSF `CustomerDetails.xhtml` View

Overview

This practice is optional and provides fewer instructions in order to provide a challenge to more advanced students. Ask your instructor whether you have time to complete this exercise.

In this practice, you create the `CustomerDetails.xhtml` page that displays the identities, names, and addresses of all customers registered in the **BrokerTool** Application.

Assumptions

This practice assumes that the application server is installed and the previous **BrokerTool** practice has been completed.

Tasks

1. Create the `CustomerManagedBean` JSF component.
   a. Create a new JSF Managed Bean.
   b. Enter the following values in the "*Name and Location*" dialog box:
      - Class Name: `CustomerManagedBean`
      - Project: `BrokerTool`
      - Location: Source Packages
      - Package: `trader.web`
      - Name: `customerDetails`
      - Scope: `request`

   c. Click the *Finish* button.
   d. Add the following import statements to the class.

```
import trader.BrokerException;
import trader.BrokerModel;
import trader.BrokerModelImpl;
import trader.Customer;
```

   e. Add an instance variable of type `BrokerModel` in the `CustomerManagedBean` class.

```
private BrokerModel model = BrokerModelImpl.getInstance();
```

   f. Create the following variables along with getters and setters in the `CustomerManagedBean` class:

```
private String message = "";
private String customerId = "";
private String customerName = "";
private String customerAddress = "";
```

g. To automate the creation of getters and setters, right-click a variable name. Select *Refactor* and then *Encapsulate Fields*. This will automate the creation of the getters and setters.

h. Create controller methods as instructed below.

- All methods should use the variables from step f and the model variable from step e during execution.
- In the event of a `BrokerException`, the message variable should be set to the value of `Exception.getMessage()`.
- Return a `String` value of `CustomerDetails`.
- Add the following method signatures listed. Implement the functionality of the methods as indicated by the method name.

```
public String retrieveCustomer()
public String updateCustomer()
public String addCustomer()
public String deleteCustomer()
```

2. Create the `CustomerDetails.xhtml` facelet page

a. Create a new JSF page.

b. Enter the following values in the “Name and Location” dialog box:

- File Name: `CustomerDetails`
- Project: BrokerTool
- Location: Web Pages
- Folder: (empty)
- Options: Facelets (selected)

c. Click the *Finish* button.

d. Modify the `CustomerDetails.xhtml` page as follows:

- View the details of a customer by using the existing CustomerDetails servlet at `http://localhost:8080/BrokerTool/`.
- View the HTML output of the `CustomerDetails` servlet using your web browser. Use this as a guide for the structure of the facelet page.
- Specify a value of *Customer Details* for the page title.
- Remove any existing body content.
- Create a table-based navigational menu along the top that includes links to:

```
<a href=’CustomerDetails.xhtml’>Customer Details</a>
<a href='AllCustomers'>All Customers</a>
<a href='Stocks.xhtml'>Stocks</a>
```

- Use the `h:form, h:inputText`, and `h:commandButton` tags along with EL to implement the Customer Details form.

3. Configure, deploy, and test the application
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.
   b. Run the application. Test your JSP by pointing a browser at:
      `http://localhost:8080/BrokerTool/CustomerDetails.xhtml`

   The other views of the application still link to the previous implementation of the Customer Details view. Do not remove the older `CustomerDetails` implementation. Converting all the views and controllers to use JSF is beyond the scope of this course. The solution project for this exercise is an example of converting the entire application to JSF.

Practice Solutions

This section contains the practice solutions.

Solutions for Practices 1 and 2

You can find solutions for the practices in this lab in the following directory:

```
D:\labs\netbeans\solutions\JSF
```

Practices for Lesson 7: EJB Component Model

Chapter 7

Practices for Lesson 7: Overview

Practices Overview

The purpose of these practices is to learn the basic structure of an EJB module and how to deploy it. The existing **SampleWebApplication** is modified to become a simple EJB client. This EJB client makes use of EJB annotations to *find* an EJB for use.

In these practices, you will:

- Create and deploy a simple EJB component module
- Use annotations and dependency injection
- Run a simple EJB client application

Practice 7-1: Creating and Deploying a Simple EJB Application

Overview

In this practice, you create and deploy an EJB module and an EJB client.

Assumptions

This practice assumes that the application server is installed and configured in NetBeans and the `SampleWebApplication` practice has been completed.

Tasks

1. Create an EJB application module.
   a. From the NetBeans menu select *File* and then *New Project*.
   b. Select Java EE and then EJB Module.
   c. Click *Next*.
   d. Enter the following information in the "Name and Location" dialog box.
      - Project Name: `SampleEJBApplication`
      - Project Location: `D:\Labs\netbeans\projects`
      - Use Dedicated Folder for Storing Libraries: (deselected)
      - Set as Main Project: (deselected)
   e. Click *Next*.
   f. In the "Server and Settings" dialog box enter the following information:
      - Server: **GlassFish Server 3.1**
      - Java EE Version: **Java EE 6**
      - Enable Contexts and Dependency Injection: (deselected)
   g. Click *Finish*.
2. Create a basic session EJB
   a. Right-click the **SampleEJBApplication** project and select *New* and then *Other*.
   b. Select *Java EE* from Categories
   c. Select *Session Bean* from File Types
   d. In the "Name and Location" dialog box, enter the following information:
      - EJB Name: **BasicSession**
      - Location: **Source Packages**
      - Package: `test`
      - Session Type: **Stateless**
      - Create Interface: `Remote in Project: SampleEJBApplication`
   e. Click *Finish*.
3. Add a business method to the `BasicSession` EJB.
   a. Add a method signature in `BasicSessionRemote.java`.

```
String getMessage();
```

b. Add a business method to `BasicSession.java`. The method should be:

```
public String getMessage() {
    return "Hello EJB World";
}
```

c. Perform a *Clean and Build* of the project

4. Add the EJB project to the libraries of the `SampleWebApplication.`
   a. Load the **SampleWebApplication** project in NetBeans.
   b. In the Projects window, right-click the *Libraries* folder.
   c. Select *Add Project*.
   d. Navigate to and select the **SampleEJBApplication** project.
   e. Click *Add Project JAR Files*.
5. Use annotations in the `BasicServlet` to look up and use the `BasicSession` EJB.
   a. In **SampleWebApplication**, open `BasicServlet` to modify it.
   b. Import the `javax.ejb.*` classes. Most EJB annotations reside in this package.
   c. Add an annotated field to the `BasicServlet` class. An annotated field is typically a standard non-final instance variable. Within the class add:
   ```
   @EJB private BasicSessionRemote basicSessionBean;
   ```
   d. When the `BasicServlet` is instantiated, any annotated fields are automatically initialized before any of the servlet methods can execute. You can use the session bean by adding the following line in the `processRequest` method:
   ```
   out.println("Message: " + basicSessionBean.getMessage());
   ```
6. Configure, deploy, and test the application

   Do **NOT** deploy the **SampleEJBApplication** EJB module. Because the **SampleEJBApplication** is a library, it will be archived as a JAR file and placed inside the WAR for the **SampleWebApplication**. Java EE 6 application servers will deploy EJB components that exist within library JAR files of web archives.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **SampleWebApplication** WAR module manually.
   b. Test your modified servlet and new EJB by entering the following URL in a browser at:
   ```
   http://localhost:8080/SampleWebApplication/BasicServlet
   ```
   c. You should see the *Hello EJB World* message displayed.

Practice 7-2: Describe the EJB Component Model

Overview

In this practice, you complete a fill-in-the-blank activity to check your understanding of the EJB component model.

Tasks

1. Fill in the blanks of the following sentences with the missing word or words:
    1. The two types of EJB are _______________ and _______________ beans.
    2. Scheduling the execution of an EJB for a later time can be accomplished with the _______________________.
    3. True or False: An Enterprise Bean instance can have its methods directly invoked by a client.
    4. The three different access types to a Session EJB are ___________, ___________, and ___________.
    5. The two Java technologies that a client can use to gain a reference to a Session Bean interface are __________ and __________________.
    6. True or False: A session EJB always requires an interface.

Practice Solutions

This section contains the practice solutions.

Solutions for Practice 1

You can find solutions for the practices in this lab in the following directory:
`D:\labs\netbeans\solutions\EJBComponents`

Solution for Exercise 2: Describing the EJB Component Model

Compare your fill-in-the-blank responses with the following answers:

1. The two types of EJB are *session* and *message-driven* beans.
2. Scheduling the execution of an EJB for a later time can be accomplished with the *EJB Timer Service*.
3. *False(clients use stubs)*: An Enterprise Bean instance can have its methods directly invoked by a client.
4. The three different access types to a Session EJB are *Local Stub*, *Remote or Distributed Stub*, and *Web Service*.
5. The two Java technologies that a client can use to gain a reference to a Session Bean interface are *JNDI* and *Annotations or Dependency Injection*.
6. *False(Java EE 6 has a local no-interface session bean)*: A session EJB always requires an interface.

Practices for Lesson 8: Implementing EJB 3.1 Session Beans

Chapter 8

Practices for Lesson 8: Overview

Practices Overview

In these practices, you will learn how to create a singleton session bean as part of a web application project. A singleton EJB is used for in-memory persistence. These practices also demonstrate the essential features of the Java Naming and Directory Interface™ (J.N.D.I. or JNDI) API as an alternative way to find EJB components.

Upon completion of these practices, you will be able to:

- Code a session EJB component
- Create EJB references for web-tier clients
- Create a Java SE EJB client
- Describe session beans

Practice 8-1: Coding the EJB Component and Client

Overview

In this practice, you will code a session EJB component.

Assumptions

This practice assumes that the application server is installed and the previous **BrokerTool** practice has been completed.

Tasks

1. Ensure that the **BrokerTool** project is open in NetBeans. Modify the `BrokerModelImpl` class to be a local singleton session bean.

   Because the `BrokerModelImpl` class maintains all application data and changes to that data in memory, a client must have access to the same `BrokerModel` instance for every request to see any changes. Allowing the client to reuse the same `BrokerModel` instance can be achieved by making `BrokerModelImpl` a singleton session bean.

   a. The `BrokerModelImpl` currently implements the traditional singleton design pattern in such a way that it uses a private constructor. Session beans cannot have non-public constructors. Modify the `BrokerModelImpl` class so that it no longer implements the singleton design pattern.

   b. Remove the static model instance and make the constructor public.

```
//    private static BrokerModel instance = new
BrokerModelImpl();

//    public static BrokerModel getInstance() {
//        return instance;
//    }

    /** Creates a new instance of BrokerModelImpl */
    public BrokerModelImpl() {
```

   c. Add the following import statements:

```
import javax.ejb.Local;
import javax.ejb.Singleton;
```

   d. Add the annotations required to make the `BrokerModelImpl` class a singleton session bean with a local interface.

   `@Local @Singleton`

2. Modify the `BrokerModel` clients to use the `BrokerModelImpl` local session bean.

   All the servlet and JSF-managed bean classes that function as controllers in the web tier must be modified to be EJB clients.

   a. Expand the `trader.web` package in the **BrokerTool** project.

   b. Clean and Build the **BrokerTool** project to discover the classes that must be modified to use the new `BrokerModelImpl` singleton session bean.

   c. For all the classes that must be modified, perform the following actions:

1) Remove any lines of code that call `BrokerModelImpl.getInstance()`.
2) Remove any model local variable declarations.
3) Create an instance-level variable:

```
@EJB private BrokerModel model;
```

The `@EJB` annotation cannot be applied to local variables.

4) Add the needed import for the `@EJB` annotation.
5) If necessary, modify any methods that do not compile to use the new model variable.

Do not call `new` on the `BrokerModelImpl` class. Calling `new` on a session bean is valid Java syntax but it treats the bean class as a POJO. Any benefits of EJB technology such as the EJB life cycle, security handling, and container managed transactions are lost when calling `new` on an EJB bean class.

3. Configure, deploy, and test the application.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.
   b. Test your application by entering the following URL in a browser:
   `http://localhost:8080/BrokerTool/`

Practice 8-2: Optional: Creating a Java SE EJB Client That Uses JNDI

Overview

In this practice, you will code a session EJB component.

Assumptions

This practice assumes that the application server is installed and the previous SampleEJBApplication exercise has been completed.

Tasks

1. Create the `SampleEJBClient` project.
   a. From the NetBeans menu select *File* and then *New Project*.
   b. Select *Java* and then *Java Application*.
   c. Click *Next*.
   d. Enter the following information in the "Name and Location" dialog box.
      - Project Name: `SampleEJBClient`
      - Project Location: `D:\Labs\netbeans\projects`
      - Use Dedicated Folder for Storing Libraries: (deselected)
      - Create Main Class: `test.Main`
      - Set as Main Project: (selected)
   e. Click *Finish*.
2. Undeploy the `SampleWebApplication` module.

This task is required because the `SampleEJBApplication` will be deployed as a stand-alone EJB module and the `SampleWebApplication` contains a copy of the `SampleEJBApplication` as a library.

3. Open the `SampleEJBApplication` project.
   a. Open the `SampleEJBApplication` project created in Lesson 7, "EJB Component Model."
   b. Build the `SampleEJBApplication` project.
4. Configure the `SampleEJBClient` project libraries. Add the `gf-client.jar` library to the `SampleEJBClient` project. The `gf-client.jar` library allows JNDI lookups from a Java SE application to a GlassFish Server 3.
   a. In the projects window, expand `SampleEJBClient`. Right-click the Libraries node and select **Add JAR/Folder** from the context menu.
   b. Browse to `D:\glassfish3\glassfish\lib\`.
   c. Select `gf-client.jar` (Use the default selection Absolute path).
   d. Click the **Open** button.

5. Add the `SampleEJBApplication` as a library to the `SampleEJBClient` project. The `SampleEJBApplication` library provides the `BasicSessionRemote` interface to the EJB client.
   a. In the projects window, expand `SampleEJBClient`. Right-click the Libraries node and select **Add Project** from the context menu.
   b. Browse to `D:\Labs\Netbeans\projects`. Select `SampleEJBApplication`. Click the **Add Project Jar Files** button.
6. Code the EJB client class. In the `test.Main` class, make the following changes:
   a. In the `main` method, obtain a JNDI Context.
      ```
      Context ctx = new InitialContext();
      ```
   b. Use JNDI to perform a lookup of the `BasicSessionBean` EJB. The syntax of the JNDI name used is `java:global/<module-name>/<bean-name>`.
   c. Cast the result of the JNDI lookup to `BasicSessionRemote` and store it in a local variable. This is the EJB stub.
   d. Use the EJB stub to call the `getMessage` method of the `BasicSession` and print the result.
   e. Catch any exceptions that occur by using a `try/catch` block.
7. Test the EJB client application.
   a. Right-click the `SampleEJBClient` project and select Build from the context menu. Do not run the project yet. Correct any errors that occur.
   b. Deploy the `SampleEJBApplication` project.
   c. Run the `SampleEJBClient` project.

   If you perform a Clean and Build on `SampleEJBClient`, that will cause `SampleEJBApplication` to be built and undeployed. If you run `SampleEJBClient` without `SampleEJBApplication` deployed, you will receive an error message similar to the following:
   ```
   javax.naming.NamingException: Lookup failed for
   'java:global/SampleEJBApplication/BasicSession' in
   SerialContext,orb'sInitialHost=localhost,orb'sInitialPort=37 00
   [Root exception is javax.naming.NameNotFoundException:
   SampleEJBApplication]
   ```
   If you receive the following error message:
   ```
       javax.naming.NoInitialContextException: Need to specify class
       name in environment or system property, or as an applet
       parameter, or in an application resource file:
       java.naming.factory.initial
   ```
   This indicates that you have not added `gf-client.jar` to the SampleEJBClient project.

8. Additional troubleshooting tips:
   a. To determine what the GlassFish server is publishing as the lookup names of the EJB objects, right-click the GlassFish icon in the Services tab under Servers and select *View Server Log*.
   b. You should see messages like this:

```
INFO: Portable JNDI names for EJB BasicSession :
[java:global/SampleEJBApplication/BasicSession!test.BasicSession
Remote, java:global/SampleEJBApplication/BasicSession]
INFO: Glassfish-specific (Non-portable) JNDI names for EJB
BasicSession : [test.BasicSessionRemote#test.BasicSessionRemote,
test.BasicSessionRemote]
INFO: SampleEJBApplication was successfully deployed in 235
milliseconds.
```

   c. Make sure that the lookup matches either the portable or non-portable JNDI names published by GlassFish.
   d. (Optional) Try changing the value of the lookup to the non-portable JNDI names.

Practice 8-3: Describing Session Beans

Overview

In this practice, you complete a fill-in-the-blank activity to check your understanding of the EJB component model.

Tasks

Fill in the blanks of the following sentences with the missing word or words:

1. The three types of session beans are _______________, _______________ and _______________.
2. To declare a session EJB as a remote EJB, you can place the @Remote annotation on the _________________ or _________________.
3. For a session bean to access its environment including transaction status, the bean needs a reference to its _________________.
4. The two annotations that have meaning in a stateful session bean but not a stateless session bean are _________________ and _________________.

Practice Solutions

Use the following solutions to check your answers to the exercises in this lab.

Solutions for Practice 1 and 2

You can find solutions for the practices in this lab in the following directory:
`D:\labs\netbeans\solutions\SessionBeans\.`

Solutions for Practice 3

Compare your fill-in-the-blank responses with the following answers:

1. The three types of session beans are *singleton, stateful*, and *stateless*.
2. To declare a session EJB as a remote EJB, you can place the @Remote annotation on the *bean class* or *business interface*.
3. For a session bean to access its environment including transaction status, the bean needs a reference to its *SessionContext*.
4. The two annotations that have meaning in a stateful session bean but not a stateless session bean are *@PostActivate* and *@PrePassivate*.

Practices for Lesson 9: The Java Persistence API

Chapter 9

Practices for Lesson 9: Overview

Practices Overview

In these practices, you will:

- Create and configure a persistence unit
- Use the basic functionality of the Java Persistence API
- Describe the Java Persistence API

In this practice, you modify the broker application to use a database. Currently, the broker application stores all domain data in memory. You create and populate a database to hold customer, share, and stock data. The `BrokerModelImpl` session bean is modified to use the Java Persistence API and the `Customer`, `CustomerShare`, and `Stock` classes are turned into entity classes.

Practice 9-1: Create the Java Persistence API Version of the BrokerTool Project

Overview

In this practice, you will create a database, a persistence unit, and modify classes to use the Java Persistence API.

Assumptions

This practice assumes that the application server and Java DB database are installed, the application server is running, and the previous **BrokerTool** exercise was completed.

Tasks

1. Create the StockMarket database.
   a. Start the Java DB database server from NetBeans.
      1) Click the *Services* tab.
      2) Expand the *Databases* node.
      3) Right-click the *Java DB* icon.
      4) Select Start Server.

   If *Start Server* is grayed-out, this means that the database was already started by NetBeans.
   b. Right-click the *Java DB* icon and select *Create Database*.
   c. Enter the following information for the database:
      - Database Name: **`StockMarket`**
      - User Name: user
      - Password: user
   d. Click *OK*. This creates the database and adds a connection for the database under the Databases icon
   e. Connect to the newly created database by right-clicking the `jdbc:derby://localhost:1527/StockMarket` connection and selecting *Connect*.
   f. From within NetBeans, open the `StockMarket.sql` file provided in the `D:\Labs\netbeans\resources\brokertool` directory.
   g. On the StockMarket.sql tab, select `jdbc:derby://localhost:1527/StockMarket` as the connection.
   h. Click the *Run SQL* icon (or use the key sequence Ctrl + Shift + E) to execute the SQL statements.
2. Examine the contents of the database.
   a. In the Services window, expand the `jdbc:derby://localhost:1527/StockMarket` connection under the Databases node.
   b. Right-click the connection and select Refresh.
   c. Expand the USER schema. You see the nodes for the tables, views, and procedures. Expand the Tables node to see the CUSTOMER, SHARES, and STOCK tables.

d. Right-click the CUSTOMER table node and select View Data. A SQL command window opens and executes an SQL command to display the data in the table.

e. Repeat the previous step for the STOCK and the SHARES tables.

3. Create a persistence unit.

a. Right-click the **BrokerTool** project, select *New* and then *Other*.

b. In the "New File" dialog box, select the *Persistence* category and *Persistence Unit* as the file type.

c. Click *Next*.

d. Enter the following information in the "Provider and Database" dialog box:

- Persistence Unit Name: `BrokerToolPU`
- Persistence Provider: `EclipseLink(JPA.2.0)(Default)`
- Use Java Transaction APIs (selected)
- Table Generation Strategy: **None**
- Data Source: **New Data Source...**

e. In the popup Create Data Source window, enter the following:

- JNDI Name: `StockMarket`
- Database Connection: `jdbc:derby://localhost:1527/StockMarket [user on USER]`
- Click Ok

f. Click *Finish*.

4. Convert the `Customer.java` class to a Java Persistence API class. In the **BrokerTool** project, add the annotations required to convert Customer to a persistence class.

a. Import the `javax.persistence` package.

b. Add a `@Entity` annotation to the `Customer` class.

c. Map the `Customer` class to the CUSTOMER table with a `@Table` annotation.

d. Using field-based access, specify that the id field is the primary key value (`@Id`). The id field should map to the SSN database column using the `@Column` annotation.

e. The `name` field should map to the CUST_NAME database column.

f. The `addr` field should map to the ADDRESS database column.

Some databases are case-sensitive. You should use the correct case for column names when specifying the column mapping.

5. Convert the `CustomerShare` and `Stock` classes to use the Java Persistence API. Modify the two classes of the BrokerTool project.

a. Import the `javax.persistence` package.

b. Add no-arg constructors for each class.

c. Add any annotations necessary to make `CustomerShare` and `Stock` Entity classes.

d. Using the database table structure as information, add field-based persistence annotations to `CustomerShare.java` and `Stock.java`.

e. For the `CustomerShare` class, add the following annotations for the `id` field.

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
@Column(name = "ID")
```

6. Modify the `BrokerModelImpl.java` class of the **BrokerTool** project to use the Java Persistence API. Modify all methods to use the newly modified `Customer`, `CustomerShare`, and `Stock` entity classes.

a. Import the `javax.persistence` package.

b. Use dependency injection to obtain a reference to an `EntityManager` instance named `em`.

```
@PersistenceContext private EntityManager em;
```

c. `Modify the BrokerModelImpl` class so there are no more in-memory lists of domain objects. Perform the following changes:

1) Change `BrokerModelImpl` to a stateless session bean.

2) Remove the `customers`, `shares`, and `stocks` list instance variables.

3) Remove all code in the constructor.

NetBeans will identify a number of errors after the instance variables are removed. Use the next step to fix all the errors.

d. Use the following example methods as a starting point to modify **all** the `BrokerModelImpl` methods to use the Java Persistence API:

```
public Stock[] getAllStocks() throws BrokerException {
    Query query = em.createNativeQuery("SELECT * FROM STOCK",
Stock.class);
    List stocks = query.getResultList();
    return (Stock[]) stocks.toArray(new Stock[0]);
}

public Stock getStock(String symbol) throws BrokerException {
    Stock stock = em.find(Stock.class, symbol);
    if (stock == null) {
        throw new BrokerException("Stock : " + symbol + " not
found");
    } else {
        return stock;
    }
}

public void addStock(Stock stock) throws BrokerException {
    try {
        em.persist(stock);
    } catch (EntityExistsException exe) {
        throw new BrokerException("Duplicate Stock : " +
stock.getSymbol());
```

```
    }
}

public void updateStock(Stock stock) throws BrokerException {
    Stock s = em.find(Stock.class, stock.getSymbol());
    if (s == null) {
        throw new BrokerException("Stock : " + stock.getSymbol()
+ " not found");
    } else {
        em.merge(stock);
    }
}

public void deleteStock(Stock stock) throws BrokerException {
    String id = stock.getSymbol();
    stock = em.find(Stock.class, id);
    if (stock == null) {
        throw new BrokerException("Stock : " + stock.getSymbol()
+ " not found");
    } else {
        em.remove(stock);
    }
}
```

6. Configure, deploy, and test the application.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.
   b. Test your application by entering the following URL in a browser at:
      `http://localhost:8080/BrokerTool/`

Practice 9-2: Describing Java Persistence API

Overview

In this practice, you answer the question or complete a fill-in-the-blank activity to check your understanding of the Java Persistence API.

Tasks

Answer the question or fill in the blanks of the following sentences with the missing word or words:

1. True or False: The Java Persistence API requires an application server.
2. The fully qualified annotation used by classes to be marked as Entity classes is _________________.
3. Entity classes often function as data transfer objects (DTOs) and implement the _________________ interface.
4. True or False: An entity class can have either field based or property based access but not both.
5. The _________________ annotation is used to have an EntityManager injected in a managed component.
6. Every entity class must have a property or field that is annotated as the _________________.
7. When the transaction ends, the entity instance becomes _________________.

Practice Solutions

Use the following solutions to check your answers to the practices in this lab.

Solutions for Practice 1

You can find solutions for the practices in this lab in the following directory: `D:\Labs\netbeans\Solutions\Persistence\`.

Solutions for Practice 2

Compare your responses with the following answers:

1. *False*: The Java Persistence API requires an application server.
2. The fully qualified annotation used by classes to be marked as Entity classes is *javax.persistence.Entity*.
3. Entity classes often function as data transfer objects (DTOs) and implement the *java.io.Serializable* interface.
4. *False for JPA 2.0, True for JPA 1.0*: An entity class can have either field based or property based access but not both.
5. The *@PersistenceContext* annotation is used to have an EntityManager injected in a managed component.
6. Every entity class must have a property or field that is annotated as the *primary key*.
7. When the transaction ends, the entity instance becomes *detached*.

Practices for Lesson 10: Implementing a Transaction Policy

Chapter 10

Practices for Lesson 10: Overview

Practices Overview

In these practices, you will:

- Determine when rollbacks occur given a specific scenario
- Use the Java Persistence API versioning features to control optimistic locking

Practice 10-1: Determining When Rollbacks Occur

Overview

In this practice you become familiar with the effect of transaction attributes on the handling of transaction rollback when failures occur in various parts of an application.

Tasks

In each of the following scenarios, you are shown a call stack. That is, you are shown a sequence of nested method calls between servlets and EJB components. The indentation shows which methods are called in a particular transaction scope. For example:

```
methodA()
methodB()
methodC()
```

In this example, `methodA` calls `methodB` and then calls `methodC`. The calls to `methodB` and `methodC` are both in the same scope.

Each method call has been assigned a transaction attribute. The first method call is always on the Controller servlet, which does not use any Java Transaction API (JTA) calls. So, you can assume that when the first method call is made by the servlet on an EJB component, no transaction is in effect at that point.

Review the scenarios and answer the questions that follow.

Scenario 1

Consider the scenario of creating a new customer record and sending a notification to the customer by email. Suppose that you have the following call stack:

```
1. No transaction            Controller.addCustomer(...)
2.                           Customer cust = new Customer()
3. Required                  BankMgr.addCustomer(cust)
4.                           em.persist(cust)
5. RequiresNew               DBLogBean.writeStatusToLog()
6. NotSupported              BankMgr.sendNotificationMessage()
```

Answer the following questions about this scenario:

a. Which methods get rolled back if a *system* exception is thrown from the method on line 5, `writeStatusToLog`?

b. Which methods get rolled back if a *system* exception is thrown from the method on line 6, `sendNotificationMethod`?

c. If the transaction attribute for the method on line 6 were `Required`, rather than `NotSupported`, which methods would be rolled back if the method on line 6 failed?

Scenario 2

Consider the scenario of transferring money between two customer accounts. Suppose that you have the following call stack:

```
1.No transaction   Controller.transferMoney()
2.Required         BankMgr.transferMoney()
3.                 Customer cust1 = em.find(Customer.class, id1)
4.                 Customer cust2 = em.find(Customer.class, id2)
5.                 cust1.setBalance()
6.                 cust2.setBalance()
```

a. Which methods would be rolled back if the method started on line 2 threw a system exception *after* running lines 3, 4, 5, and 6 all successfully?

b. Which methods would be rolled back if the method on line 2 threw a `BankException` after running lines 3, 4, 5, and 6 all completed successfully?

c. Which lines would be rolled back if a call was made between lines 5 and 6 to `setRollbackOnly`?

Practice 10-2: Using the Versioning Features of the Persistence API to Control Optimistic Locking

Overview

In this practice, you will modify the **BrokerTool** application to use versioning.

Assumptions

This practice assumes that the application server and the Java DB database are installed, the application server is running, and the previous **BrokerTool** practice was completed.

Tasks

1. Demonstrate lost updates in the BrokerTool application.
   a. Deploy the **BrokerTool** web application if it is not deployed currently.
   b. Launch two web browsers that are referred to as Browser A and Browser B.
   c. In both Browser A and B, launch the **BrokerTool** application by using the URL `http://localhost:8080/BrokerTool/`.
   d. In both Browser A and B, retrieve the same customer's details.
   e. In Browser A, change the customer's name or address and click the Update button.
   f. Browser B does not know that the customer being displayed has been changed. In Browser B, change the name or address to something other than what was entered in Browser A, and click the Update button. The changes made in Browser B overwrite those made in Browser A. This is called a lost update
2. Modify the StockMarket Database to support versioning.
   a. Switch to the Services window. Start the Java DB database server if it is not already started.
   b. Connect to the StockMarket database using the `jdbc:derby://localhost:1527/StockMarket` connection.
   c. Recreate the database tables and populate them by executing the `VersioningStockMarket.sql` provided in the `D:\Labs\netbeans\resources\brokertool` directory.
   d. View the data stored in the `CUSTOMER,` `STOCK`, and `SHARES` tables.
3. Update the `BrokerLibrary` entity classes to support versioning.
   a. Modify `Customer.java`, `CustomerShare.java`, and `Stock.java` to support versioning by adding the following code to each class:

```
@Version
@Column(name = "VERSION")
private int version = 1;

public int getVersion() {
        return version;
}
```

   b. Add any required import statements.

c. Add a new multi-arg constructor in each entity class to receive all initialization data and an additional version value. An example constructor for `Customer` is provided as follows:

```
public Customer(String id, String name, String addr, int
version){
        this(id, name, addr);
        this.version = version;
}
```

4. Add a hidden version form input field to the `CustomerDetails` servlet

The steps listed below assume you are using the `CustomerDetails` servlet, if you preformed the optional `CustomerDetails.jsp` lab please modify the steps as needed.

a. Create a `version` variable for use in the form. It should be a local variable with a value that is obtained from the `Customer` object stored in the request scope. This can be done in the same way the name, ID, and address data is retrieved.

b. Modify the `CustomerDetails` servlet in the **BrokerTool** project to support a new hidden form element. Insert the hidden form element after the form element and before the table. Use the following code:

```
out.println("<input type='hidden' name='version' value='" +
version + "'/>");
```

5. Modify `CustomerController` to use the version value.

a. After retrieving all other submitted form data, add the following code to read the value of the hidden version form input:

```
int version = 1;
if(request.getParameter("version") != null) {
  version = Integer.parseInt(request.getParameter("version"));
}
```

b. Find any calls to `new Customer` in the `CustomerController` and modify them to pass the `version` value to the constructor created in Task 3, Step c.

6. Modify `BrokerModelImpl` to use versioning.

a. In `BrokerModelImpl`, any methods that invoke merge operations to update entity data can possibly cause an `OptimisticLockException`. `OptimisticLockException` is a subclass of `RuntimeException` and should be caught to avoid invalidation the `BrokerModelImpl` session bean in the web tier.

b. Handle all merge calls in a fashion similar to the following example:

```
try {
    em.merge(cust);
} catch(OptimisticLockException ole) {
    throw new BrokerException("Record for " + cust.getId() + "
has been modified since retrieval");
}
```

c. Add any required import statements.

7. Configure, deploy, and test the application

a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.

b. Launch two web browsers that are referred to as Browser A and Browser B.

c. In both browsers A and B, launch the **BrokerTool** application using the URL: `http://localhost:8080/BrokerTool/`.

d. In both browsers A and B, retrieve the same customer's details.

e. In Browser A, change the customer's name or address and click the Update button.

f. Browser B does not know that the customer being displayed has been changed. In Browser B, change the name or address to something other then what was input in Browser A, and click the Update button. The changes made in Browser B are no longer accepted because the customer's data has been modified by another client.

A production quality application would probably not store the version value in a hidden form field because a knowledgeable user could forge any version value. A better method would be to store the information in the web server by using an `HttpSession`.

Practice Solutions

Use the following information to verify your answers to the scenario questions.

Solution For Practice 1

Scenario 1

Consider the scenario of creating a new customer record and sending a notification to the customer by email. Suppose that you have the following call stack:

```
1. No transaction               Controller.addCustomer(...)
2.                              Customer cust = new Customer()
3. Required                     BankMgr.addCustomer(cust)
4.                              em.persist(cust)
5. RequiresNew                  DBLogBean.writeStatusToLog()
6. NotSupported                 BankMgr.sendNotificationMessage()
```

a. Which methods get rolled back if a *system* exception is thrown from method on line 5, `writeStatusToLog`?

   Only the method on line 5.

   The transaction that was initiated on entry to the method on line 3 is suspended on entry to the method on line 5. When the method on line 5 fails, its own transaction is rolled back. The original transaction is then resumed intact. The `BankMgr.addCustomer` method should catch an `EJBException`, and might choose to roll itself back if the logic dictates. This example demonstrates the use of the `RequiresNew` attribute to isolate non-critical code from a transaction.

b. Which methods get rolled back if a *system* exception is thrown from method on line 6, `sendNotificationMethod`?

   None.

   If there was any transaction in effect on entry to the method on line 6, it would be suspended. Consequently, anything that happens in the method on line 6 is isolated from the outer transaction. Like `RequiresNew`, the `NotSupported` attribute is often used to isolate non-critical logic from a critical transaction.

c. If the transaction attribute for line 6 were `Required`, rather than `NotSupported`, which methods would be rolled back if the method on line 6 failed?

   Only line 6.

   Line 3 and 6 are called by the method `Controller.addCustomer`. Because `Controller.addCustomer` has no transaction, lines 3 and 6 must be different transactions. As a result, a failure in the method on line 6 cannot affect line 3. Because line 5 is called from line 3, it is immune to a failure in line 6.

Scenario 2

Consider the scenario of transferring money between two customer accounts. Suppose that you have the following call stack:

```
1.No transaction   Controller.transferMoney()
2.Required         BankMgr.transferMoney()
3.                 Customer cust1 = em.find(Customer.class, id1)
4.                 Customer cust2 = em.find(Customer.class, id2)
5.                 cust1.setBalance()
6.                 cust2.setBalance()
```

a. Which methods would be rolled back if the method started on line 2 threw a system exception *after* running lines 3, 4, 5, and 6 all successfully?

   Lines 2-6.

   The transaction does not commit until the method on line 2 exits successfully. If a method throws a system exception, the container rolls back the transaction rather than committing it.

b. Which methods would be rolled back if the method on line 2 threw a `BankException` after running lines 3, 4, 5, and 6 all completed successfully?

   None.

   `BankException` is an application exception. Throwing it does not cause the container to roll back any transaction.

c. Which lines would be rolled back if a call was made between lines 5 and 6 to `setRollbackOnly`?

   Lines 2-6.

   Moreover, when `setRollbackOnly` is called, line 6 then proceeds to do its work, even though everything it does is rolled back later. When line 2 completes, the container examines the `rollbackOnly` flag and, because it is set, rolls back the transaction. The `setRollbackOnly` method not only affects work done up until the point it was called, it also affects the entire transaction.

Solution for Practice 2

You can find sample solutions for Practice 2 in this lab in the following directory:
`D:\Labs\student\solutions\Transactions\.`

Practices for Lesson 11: Developing Java EE Applications Using Messaging

Chapter 11

Practices for Lesson 11: Overview

Practices Overview

There are no practices associated with this lesson.

Practices for Lesson 12: Developing Message-Driven Beans

Chapter 12

Practices for Lesson 12: Overview

Practices Overview

In these practices, you write a message-driven bean that accepts messages that list the current price of stocks in the **BrokerTool** application. The contents of the message specify the stock ID and the current price of the stock. This simulates the way in which an EJB application could use messaging to interact with, for example, a legacy system. The message-driven bean interacts with the `BrokerModelImpl` session bean to carry out the actual update operation. The `BrokerModelImpl` bean, in turn, interacts with the Stock entity class to modify the database.

To test the message-driven bean, you need a source of messages. Consequently, part of this exercise is to deploy a JMS message producer that can send messages in the appropriate format.

You also need to configure a new queue and queue connection factory in the application server. The message-driven bean is then assigned to the JNDI name of the queue.

Practice 12-1: Implementing the Message-Driven Bean

Overview

In this practice, you will create a message-driven bean in the BrokerTool application and test it.

Assumptions

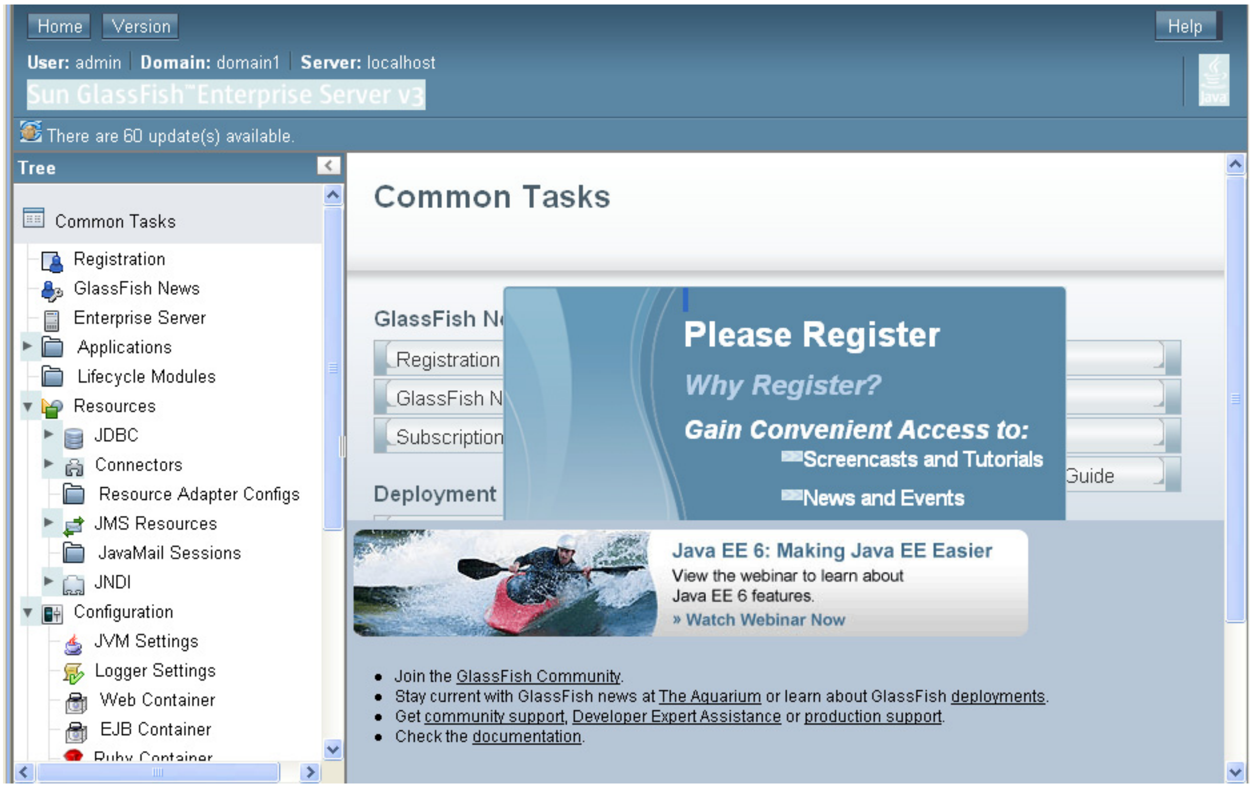
This practice assumes that the application server and the Java DB database are installed, the application server is running, and the previous **BrokerTool** exercise was completed.

Tasks

1. Create the managed resources.
   A JMS queue and connection factory must be created in the application server. NetBeans can be used to create these JMS administered objects. To create them, complete the following steps.
   a. Complete the following steps to create a JMS queue with a JNDI name of `jms/UpdateStock`.
      1) Right-click the **BrokerTool** project, select *New* and then *Other*.
      2) In the “New File” dialog box, select the *GlassFish* category and *JMS Resource* as the file type.
      3) Click *Next*.
      4) Enter the following information in the “General Attributes - JMS Resource” dialog box:
         - JNDI Name: `jms/UpdateStock`
         - Enabled: true
         - Admin Object Resource: `javax.jms.Queue`
      5) Click *Next*.
      6) In the “JMS Properties” dialog box, the *Name* property should have a value of **UpdateStock**. After setting the value, click the *Name* property to make NetBeans accept your value.
      7) Click *Finish*.
   b. Complete the following steps to create a JMS connection factory with a JNDI name of `jms/UpdateStockFactory`.
      1) Right-click the **BrokerTool** project, select *New* and then *Other*.
      2) In the “New File” dialog box, select the *GlassFish* category and *JMS Resource* as the file type.
      3) Click *Next*.
      4) Enter the following information in the “General Attributes - JMS Resource” dialog box.
         - JNDI Name: `jms/UpdateStockFactory`
         - Enabled: `true`
         - Connector Resource: javax.jms.QueueConnectionFactory
      5) Click *Finish*.

2. Copy the `StockMessageProducerBean` EJB
   a. The JMS message producer is provided to you in the form of a scheduled no-interface local session EJB.
   b. In the *Favorites* window, copy the `StockMessageProducerBean.java` file from *resources* > *brokertool* to the clipboard.
   c. Paste the `StockMessageProducerBean.java` file into the `trader` package of the **BrokerTool** project.
   d. View the source code for the `StockMessageProducerBean` EJB. The EJB tries to update the *ORCL* stock price once per minute.
3. Create the message-driven bean: Create a new message-driven bean in the **BrokerTool** project.
   a. Right-click the **BrokerTool** project, select *New* and then *Other*.
   b. In the "New File" dialog box, select the *Java EE* category and *Message-Driven Bean* as the file type.
   c. Click *Next*.
   d. Enter the following information in the "Name and Location" dialog box.
      - EJB Name: `UpdateStockBean`
      - Project: BrokerTool
      - Location: Source Packages
      - Package: `trader`
      - Project Destinations: `jms/UpdateStock`
   e. Click *Finish*.
   f. Add an annotated EJB reference variable for the `BrokerModelImpl` session bean to the `UpdateStockBean`.
      ```
      @EJB private BrokerModel model;
      ```
   g. The `onMessage` method should receive a `javax.jms.TextMessage` object containing a message in the format of *ORCL,200.75*. Parse this message by using the `String` class method `split(",")` and the `Double.parseDouble("")` method.
   h. Use the `model` reference obtained in step g to retrieve and update the current stock price. Catch any exceptions that occur and print their stack trace. Do not throw *any* exception from `onMessage`, because the container will try to deliver the message again.
4. Configure and deploy the application.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.
   b. After deployment, log in to the application server administrative interface.
      1. Switch to the Services window.
      2. Expand the Servers node.
      3. Right-click the GlassFish server node in the Services window and select **View Admin Console**.
      4. The URL **`http://localhost:4848`** opens in your web browser. The port 4848 is the default port specified during installation. Your administration port might be different.

5. Provide the administration user name and password (defaults are **`admin/adminadmin`**) in the welcome screen. Upon successful login, you should see the screen illustrated in the following figure:



- Join the GlassFish Community.
- Stay current with GlassFish news at The Aquarium or learn about GlassFish deployments.
- Get community support, Developer Expert Assistance or production support.
- Check the documentation.

c. Click *Resources* > *JMS Resources*. Examine the JMS resources created on the application server. You should see a JMS connection factory with the JNDI name `jms/UpdateStockFactory`. You should also see a JMS destination resource with the JNDI name `jms/UpdateStock`.

d. View the stock prices by entering the following URL in a browser:
`http://localhost:8080/BrokerTool/Stocks.xhtml`

e. You should see the current stock prices. Wait more than one minute and refresh the page. You should see a change in the `ORCL` stock price.

Practice 12-2: Describing Message-Driven Beans

Overview

In this practice, you answer questions and complete a fill-in-the-blank activity to check your understanding of message-driven beans.

Tasks

1. Answer the question or fill in the blanks of the following sentences with the missing word or words:
   a. Message-driven beans are: (pick one)
      1) Synchronous message consumers
      2) Asynchronous message consumers
   b. The _________________ method in a JMS MDB is called by the server when a message arrives.
   c. To send a message to a queue a JMS client would need to obtain a ____________________ and a ____________________ using either JNDI or dependence injection.
   d. True or False: A message-driven bean must have an `onMessage` method.

Exercise Solutions

Use the following solutions to check your answers to the exercises in this lab.

Solutions for Practice 1

You can find solutions for the exercises in this lab in the following directory:
`D:\labs\netbeans\solutions\MessageDriven.`

Solution for Practice2

Compare your responses with the following answers:

a. Message-driven beans are: (pick one)

   2) Asynchronous message consumers

b. The *onMessage(Message)* method in a JMS MDB is called by the server when a message arrives.

c. To send a message to a queue a JMS client would need to obtain a *javax.jms.QueueConnectionFactory* and a *javax.jms.Queue* using either JNDI or dependence injection.

d. *False* (only **JMS** MDBs must have an onMessage method): A message-driven bean must have an `onMessage` method.

Practices for Lesson 13: Web Service Model

Chapter 13

Practices for Lesson 13: Overview

Practices Overview

There are no practices associated with this lesson.

Practices for Lesson 14: Implementing Java EE Web Services with JAX-RS & JAX-WS

Chapter 14

Practices for Lesson 14: Overview

Practices Overview

In these practices, you create a JAX-WS web service, a JAX-RS web service, and a JAX-WS web service client to check the price of a stock. There are two methods to making a web service with JAX-WS, starting with a WSDL file, or starting with a Java program. You create a small Java class that the application server turns into a web service.

When the application server has created a web service from your Java code, a WSDL file will be available on the server. You use this WSDL file to generate client-side Java code for use in a test application.

Practice 1 and 2 focus on JAX-WS. Practice 3 implements the same functionality as Practice 1 but uses JAX-RS. There is no JAX-RS client exercise because JAX-RS does not include a client API.

Practice 14-1: Creating a JAX-WS Web Service

Overview

In this practice, you create a class, called `StockPrice` that functions as a web service. The `StockPrice` web service allows clients to retrieve the price of any stock in the **BrokerTool** application.

Assumptions

This practice assumes that the application server and the Java DB database are installed, the application server is running, and the previous **BrokerTool** practice was completed.

Tasks

1. Create the `StockPrice` web service in the **BrokerTool** project.
   a. Right-click the **BrokerTool** project.
   b. Select *New* and then *Other*.
   c. Select *Web Services* from Categories.
   d. Select *Web Service* from File Type.
   e. Click *Next*.
   f. In the "Name and Location" dialog box, enter the following information:
      - Web Service Name: **`StockPrice`**
      - Location: `Source Packages`
      - Package: `trader.web`
      - Create Web Service from Scratch (Selected)

   g. Click *Finish*.

   Note: You may receive error messages in the GlassFish Server console at this point. This is because you have not created a Web service end-point, which you will do next.

   h. Add the `getStockPrice` method to the `StockPrice` class:
      - Add a method with the following signature:
        ```
        public String getStockPrice(String symbol) { .. }
        ```
      - Add the `@WebMethod` annotation to the `getStockPrice(...)` method.
      - Add an annotated EJB reference variable for the `BrokerModelImpl` session bean to the `StockPrice` class.

        @EJB private BrokerModel model;
      - Add any needed import statements.
      - Use the BrokerModelImpl session bean to retrieve the current price of the requested stock and return its value as a String.
      - Wrap the code for the getStockPrice method in a try/catch block. Do not throw an exception from the `getStockPrice(...)` method. Complex data types, such as a `BrokerException`, returned or thrown from a web service method require JAXB bindings. Return the `String` *`Price` unavailable* when an `Exception` occurs.
2. Compile and deploy the BrokerTool application. Deploying the `StockPrice` web service generates several supporting classes and a WSDL on the application server.

   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application to the application server manually.

3. Test the `StockPrice` web service
   a. View the XML output generated in WSDL by entering the following URL in a web browser: `http://localhost:8080/BrokerTool/StockPriceService?WSDL`
   b. Test the `StockPrice` web service by entering the following URL in a web browser: `http://localhost:8080/BrokerTool/StockPriceService?Test`

Practice 14-2: Creating a Web Service Client

Overview

In this practice, you create a standard command line Java application that functions as a JAX-WS client. This is a small application designed to test the `StockPrice` web service.

Assumptions

This practice assumes that the application server and the Java DB database are installed, the application server is running, and the previous **BrokerTool** practice was completed.

Tasks

1. Create the web service port or proxy classes.
   a. Create a new Java Application Project named the **WebServiceTester** project by completing the following steps:
      1) Select *File* from the NetBeans menu, and then *New Project*.
      2) Select *Java* and then *Java Application.*
      3) Click the *Next* button.
      4) In the "Name and Location" dialog box, enter the following information.
         - Project Name: **`WebServiceTester`**
         - Project Location: `D:\Labs\netbeans\projects`
         - Use Dedicated Folder for Storing Libraries: (deselected)
         - Create Main Class: **`webservicetester.Main`**
         - Set as Main Project: **`(deselected)`**
      5) Click *Finish*.
   b. When developing a simple JAX-WS web service client, you use helper classes to perform all low-level SOAP and HTTP work. JAX-WS can generate these helper classes after analyzing a WSDL. To generate these classes, complete the following steps:
      1) Right-click the **WebServiceTester** project.
      2) Select *New* and then *Web Service Client*.
      3) Enter the following information for the web service client in the "WSDL and Client Location" dialog box:
         - WSDL URL: `http://localhost:8080/BrokerTool/StockPriceService?WSDL`
         - Package: `webservicetester`
         - Client Style: JAX-WS Style
      4) Click *Finish*.

   Notice that a *Generated Sources (jax-ws)* node has appeared in the **WebServiceTester** project. These class files are used to communicate with the remote web service.

2. Code the web service client application. Add the following to the `main` method in `webservicetester.Main`:

```
StockPriceService service = new StockPriceService();
StockPrice port = service.getStockPricePort();
System.out.println("Stock price is: " +
port.getStockPrice("ORCL"));
```

3. Compile and execute the WebServiceTester application.

   Compile and execute the **WebServiceTester** application. Because it is a standard command-line application, there is no need to deploy it. Correct any errors.

The application should display the current stock price on a NetBeans output tab.

Practice 14-3: Create a JAX-RS Web Service

Overview

In this practice, you create a class, called `StockResource`, which functions as a web service. The `StockResource` web service allows clients to retrieve the price of any stock in the **BrokerTool** application.

Assumptions

This practice assumes that the application server and the Java DB database are installed, the application server is running, and the previous **BrokerTool** exercise was completed.

Tasks

1. Create the `StockResource` web service.
   a. Right-click the **BrokerTool** project.
   b. Select New > Other > WebServices and then RESTful Web Services from Patterns.
   c. Click Next.
   d. In the "Select Pattern" dialog box, enter the following information:
      - Design Pattern: Simple Root Resource
      - Click *Next*.
   e. In the "Specify Resource Class" dialog box, enter the following information:
      - Project: BrokerTool
      - Location: Source Packages
      - Resource Package: **`trader.web`**
      - Path: `stocks/{symbol}`
      - Class Name: `StockResource`
      - MIME Type: text/plain
      - Representation Class: `java.lang.String`
   f. Click *Finish*.
   g. If a "REST Resources Configuration" dialog box appears, enter the following information:
   h. Specify the way REST resources will be registered in the application: **`Create default REST servlet adaptor in web.xml (selected)`**
   i. REST Resources Path: `/resources`
   j. Click *ok*.
   k. This web service only allows the reading of a stock price. Remove the `putText` method from the `StockResource` class.
   l. Implement the `getStockPrice` method.
   m. Add or remove import statements as needed.
   n. Rename the `getText` method to `getStockPrice`. While the name of the method does not matter to a RESTful web service client it is good practice.
   o. In the `getStockPrice` method, use JNDI to obtain a reference variable for the `BrokerModelImpl` session bean stub.

```
javax.naming.Context ctx = new InitialContext();
            BrokerModel model =
(BrokerModel)ctx.lookup("java:global/BrokerTool/BrokerModelImpl"
);
```

p. Use the `BrokerModelImpl` session bean to retrieve the current price of the requested stock and return its value as a `String`.

q. Wrap the code for the `getStockPrice` method in a try/catch block. Do not throw an exception from the `getStockPrice(...)` method. Return the `String` *Price unavailable* when an `Exception` occurs.

r. View the `web.xml` deployment descriptor. Notice the changes made by NetBeans when a RESTful web service was created.

2. Compile and deploy the BrokerTool application.

   Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application to the application server manually.

3. Test the `StockResource` web service

   View the text output of the RESTful web service by entering the following URL in a web browser: `http://localhost:8080/BrokerTool/resources/stocks/ORCL`.

   For more complex RESTful web services that use methods such as `PUT` and `DELETE`, there are many ways to test the service. Most IDEs provide some type of test client and RESTful web browser plugins are available.

Practice 14-4: Describing Web Services

Overview

In this practice, you complete a fill-in-the-blank activity to check your understanding of Java web services.

Tasks

1. Fill in the blanks of the following sentences with the missing word or words:
    1. The portable file used to define a web service interface is known as a ____________.
    2. ___________ is the standard web service XML dialog box that is typically transferred through HTTP.
    3. A ___________ and _________ EJB can be a web service endpoint.
    4. The only other Java web service endpoint besides an EJB is a _____________ endpoint.
    5. In JAX-WS, both endpoint types use the __________________ class-level annotation to indicate a web service.
    6. __________________ is the Java API to create web services that do not use SOAP.
    7. Complex objects are return values of a web service method that requires the use of _____________.

Solutions for Practices 1, 2, and 3

You can find example solutions for the exercises in this lab in the following directory:
`D:\Labs\netbeans\solutions\WebServices\`

Solution for Practice 4: Describing Java Web Services

Compare your fill-in-the-blank responses with the following answers:

1. The portable file used to define a web service interface is known as a *WSDL*.
2. *SOAP* is the standard web service XML dialog that is typically transferred through HTTP.
3. A *Stateless Session* and *Singleton Session* EJB can be a web service endpoint.
4. The only other Java web service endpoint besides an EJB is a *Servlet*|*Web*|*POJO* endpoint.
5. In JAX-WS, both endpoint types use the *@javax.jws.WebService* class-level annotation to indicate a web service.
6. *JAX-RS* is the Java API to create web services that do not use SOAP
7. Complex objects are return values of a web service method that requires the use of *JAXB*.

Practices for Lesson 15: Implementing a Security Policy

Chapter 15

Practices for Lesson 15: Overview

Practices Overview

At present, your application has no access control, so it is completely open to all users. In these practices, you implement an end-to-end security policy. That is, you implement a policy that encompasses the business logic and all of its clients, including the `PortfolioController` servlet, any JSP components, and stand-alone clients. This policy is defined in terms of two Java EE roles: admin and customer:

Members of the admin role have complete access to all of the components of the application. They can, therefore, view the portfolio of any customer.

Members of the customer role can only view their own portfolio details.

For ease of testing, you implement the security policy step-by-step, testing at each stage. The first step is to complete the `PortfolioController` servlet. At present, when the user clicks the Show Portfolio link, it results in a call to `getAllCustomerShares` on the `BrokerModelImpl` EJB component. The `getAllCustomerShares` method should use the EJB security API to determine the current user.

The next stage is to apply a security constraint to the web application, so that only authenticated users can invoke the application. Finally, you apply security constraints to the methods of the `BrokerModelImpl` EJB component, to give finer control over access compared to what can be accomplished at the web tier.

Practice 15-1: Using the EJB Security API to Get the User's Identity in an EJB Component

Overview

In this practice, you add security to the `BrokerModelImpl` session bean. The `getAllCustomerShares` method returns an array of `CustomerShares`. The method is modified to use the EJB security API to determine who is logged in. If no user is logged in, it throws an exception.

Assumptions

This practice assumes that the application server and the Java DB database are installed, the application server is running, and the previous **BrokerTool** practice was completed.

Tasks

1. Add security features to the `getAllCustomerShares` method in the `BrokerModelImpl` class:
   a. Add the following import statements:

```
import java.security.Principal;
import javax.annotation.Resource;
import javax.ejb.SessionContext;
```

   b. Declare a session context for the class:

```
@Resource private SessionContext ctx;
```

   c. Use the `getCallerPrincipal` method to get a `java.security.Principal` object for the current logged-in user. The `getCallerPrincipal` method is defined on the `SessionContext` object that is injected by the container when it initializes the EJB component. In other words, make the call:

```
java.security.Principal p =  context.getCallerPrincipal();
```

   d. Call the `getName` method on the `Principal` object to get a `String` representation of the user ID of the logged-in user.

```
Principal principal = ctx.getCallerPrincipal();
String name = principal.getName();
```

   e. If the user ID is `guest` or `anonymous` (in any mixture of uppercase or lowercase) then no user is logged in. In this case, throw a `BrokerException` with the text *Not logged in*.
2. Deploy and test the session bean.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually.
   b. Test the session bean by entering the following URL in a web browser:

```
http://localhost:8080/BrokerTool/AllCustomers
```

   c. Follow the link called `View` in the Portfolio column.

   You should see the error message indicating that you are not logged in.

Practice 15-2: Creating Roles, Users, Groups, and a Web-Tier Security Policy

Overview

So far, you have coded the application to the extent that it is able to determine the details of the current user. However, you do not yet have a method to log in, or any user credentials against which to verify a login attempt.

In this practice, you define the customer and admin security roles at the application level and create two user groups in the application server. You then map the roles onto the user groups. Next, you apply security constraints to the URL patterns that the web browser invokes. This has two effects. First, it restricts access to those URLs to certain users. Second, it forces the web server to prompt the user to authenticate.

Assumptions

This practice assumes that the application server and the Java DB database are installed, the application server is running, and the previous **BrokerTool** practice was completed.

Tasks

1. Create roles in the application. Make changes to the `BrokerModelImpl` class to add roles to the application.
   a. Add an import statement for the `DeclareRoles` annotation.
      `import javax.annotation.security.DeclareRoles;`
   b. Add a class-level annotation in `BrokerModelImpl`. The annotation defines the two available user roles for this class: `@DeclareRoles({"admin","customer"})`
2. Create users and groups in the application server. You add two users to a security realm.
   a. Log in to the administration console:
      `http://localhost:4848`
   b. Select *Configuration* > default-config > *Security* > *Realms* > *file*.
   c. Click the *Manage Users* button.
   d. Add the two users, 111-11-1111 and 123-45-6789. If these users no longer exist in your application, you can use alternative users. Put user 111-11-1111 in the level1 and level2 groups, and put user 123-45-6789 in the level1 group. Use the information in the following table to configure these users.

User ID	Password	Group List
111-11-1111	password	level1,level2
123-45-6789	password	level1

Note: There is no space after the comma in the Group List.

3. Map roles to groups.
   a. Edit the `sun-web.xml` deployment descriptor of the BrokerTool project.
   If the `sun-web.xml` file has not been created, you can create it by right-clicking the **BrokerTool** project. Then select *New* > *Other* > *GlassFish* > *Sun-* Deployment Descriptor.*
   b. Add the mapping inside the `sun-web-app` element after the `context-root` tags.

```
<security-role-mapping>
    <role-name>admin</role-name>
<group-name>level2</group-name>
</security-role-mapping>
<security-role-mapping>
<role-name>customer</role-name>
<group-name>level1</group-name>
</security-role-mapping>
```

   c. At the end of this task, the Java EE security role, `customer`, is mapped onto the `level1` server group, and the `admin` role is mapped onto the `level2` group.
4. Create a security constraint.
   Complete the following steps to create a security constraint in the web module, so the `/PortfolioController` URL is accessible only to the `customer` and `admin` roles:
   a. Add the `@ServletSecurity` annotation to the `PortfolioController` class.

```
@ServletSecurity(@HttpConstraint(rolesAllowed =
{"admin","customer"}))
```

   b. These annotations restrict access to the `/PortfolioController` URL to users in the `admin` or `customer` roles.
   c. Edit the `web.xml` deployment descriptor in the **BrokerTool** project.
   d. Add a `login-config` element inside the `web-app` element right before the closing `web-app` tag:

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>file</realm-name>
</login-config>
```

   e. This login configuration instructs the server to use basic authentication to authenticate users. The realm is a group of users configured in the application server. Different realms can be configured to retrieve users from files, databases, LDAP, password files, etc.
5. Deploy and test the application.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually. Resolve any errors before you continue.
   b. Enter the following URL in a web browser:

```
http://localhost:8080/BrokerTool/AllCustomers
```

   c. Attempt to view a customer's portfolio. The `CustomerController` URL now has a security constraint, and if you have not yet logged in, you should be prompted to log in.

d. Enter the user ID and password for user 123-45-6789.

If the user ID and password are not accepted, complete the following steps:

- Clear all cookies in your system's web browser.
- Close all instances of the browser window, and restart your browser.
- Repeat the steps of this task from step number b.

e. You should see the customer portfolio. Because you are no longer calling the `BrokerModelImpl.getAllCustomerShares` method as the guest or anonymous user, you can see the portfolio data. If this test is successful, it shows that the web tier has authenticated the user and propagated the user credentials to the EJB tier.

f. View other customer portfolios. This should also succeed regardless of what user you logged in as. This is not what is required by the application's security, because only members of the `admin` role should be able to view other customers' portfolios. Members of the `customer` role, such as user 123-45-6789, should only be able to view their own accounts. You fix this in the next practice.

Practice 15-3: Creating an EJB Tier Security Policy

Overview

In Practice 1, you restricted access to the `BrokerModelImpl.getAllCustomerShares` method programmatically, allowing only logged-in users to execute the method. In Practice 2, you protected the web page that shows the results of the `BrokerModelImpl.getAllCustomerShares` method, thereby causing the calls to the `BrokerModelImpl.getAllCustomerShares` to have role and principal credentials.

If other pages are restricted with different roles, any of those pages could execute the `BrokerModelImpl.getAllCustomerShares` method. In this exercise, you restrict all unallowed access to the `BrokerModelImpl.getAllCustomerShares` method both declaratively and programmatically.

This exercise contains the following sections that describe the tasks to restrict the use of the `BrokerModelImpl.getAllCustomerShares` method to members of the `admin` or `customer` role.

Assumptions

This practice assumes that the application server and the Java DB database are installed, the application server is running, and the previous **BrokerTool** practice was completed.

Task

1. Restrict `BrokerModelImpl` methods.
   a. Verify the class-level annotation to `BrokerModelImpl` of:
      ```
      @DeclareRoles({"admin","customer"})
      ```
      This states that the admin and customer roles are used in this EJB.
   b. Import the `@RolesAllowed` annotation.
      ```
      import javax.annotation.security.RolesAllowed;
      ```
   c. Add a method-level annotation to the `BrokerModelImpl.getAllCustomerShares` method of:
      ```
      @RolesAllowed({"admin","customer"})
      ```
      This prohibits anyone not in the admin or customer role from calling the `getAllCustomerShares` method.
2. Customize the `BrokerModelImpl` methods by role. In the previous task, you declared that only the `admin` and `customer` roles are allowed to call the `getAllCustomerShares` method. A customer should not be allowed to view other customer shares. There is no way to define this restriction declaratively; it must be done programmatically.
   a. Comment out the code at the beginning of the `getAllCustomerShares` method that deals with anonymous or guest users.
   b. Modify that `getAllCustomerShares` method so that a `BrokerException` is thrown if one of the conditions does not pass:
   c. The caller is not in the `admin` role. Use the `ctx.isCallerInRole` method.
   d. If you do not have a context reference you can obtain one by adding `@Resource private SessionContext ctx;` as an instance-level variable.

e. The principal's name does not match the ID passed as an argument to the `getAllCustomerShares` method.

3. Deploy the application and test it.
   a. Save any modified files. If Deploy on Save is not enabled, deploy the **BrokerTool** web application manually. Resolve any errors before you continue.
   b. Enter the following URL in a browser: `http://localhost:8080/BrokerTool/AllCustomers`
   c. Select a customer's portfolio to view. You should be prompted for a password. Enter the user name and password for an account in the admin role. You should be able to view all customer portfolios.
   d. Close all instances of your web browser to log out.
   e. Launch a new web browser and enter the following URL: `http://localhost:8080/BrokerTool/AllCustomers`
   f. Select a customer's portfolio to view. You should be prompted for a password. Enter the user name and password for an account NOT in the admin role. You should only be able to view that customer's portfolio.

Practice 15-4: Describing Java EE Security

Overview

In this practice, you complete a fill-in-the-blank activity to check your understanding of Java EE Security.

Tasks

Fill in the blanks of the following sentences with the missing word or words:

a. To check the calling user, an EJB would use its _______________.

b. The web tier equivalent of `isUserInRole(...)` is ____________________.

c. Two common security annotations used in an EJB are ______________ and ______________.

d. Web-tier components configure their security settings in the _____________ file.

e. The ______________ version of enterprise Java first allowed the `@ServletSecurity` annotation to be used in a servlet.

Practice Solutions

Use the following solutions to check your answers to the exercises in this lab.

Solutions for Practice 1, 2, and 3

You can find sample solutions for the exercises in this lab in the following directory:
`D:\labs\netbeans\solutions\Security.`

Solution for Practice 4

Compare your fill-in-the-blank responses with the following answers:

a. To check the calling user, an EJB would use its *EJBContext or SessionContext*.
b. The web-tier equivalent of `isUserInRole(...)` is *isCallerInRole(...)*.
c. Two common security annotations used in an EJB are *@DeclareRoles* and *@RolesAllowed*.
d. Web-tier components configure their security settings in the *web.xml* file.
e. The *JavaEE6* version of enterprise Java first allowed the `@ServletSecurity` annotation to be used in a servlet.