Developing Applications for the Java EE 6 Platform

Activity Guide - Windows

FJ-310-EE6

D65269GC11

Edition 1.1

May 2010

D67414

ORACLE

This page intentionally left blank.

This page intentionally left blank.

Table of Contents

v

vii

ix

Developing Applications for the Java™ EE Platform

<u>Lab Preface</u>

About This Workbook

Workbook Goals

Upon completion of this workbook, you should be able to:

- Package and deploy a Java™ Platform, Enterprise Edition (Java™ EE platform) application

- Implement a servlet, a JavaServer Page™ (JSP™), a JavaServer Face (JSF) Facelet, a session bean, an entity class, a message-driven bean, and a web service

- Implement a web-based user interface that follows the Model-View-Controller (MVC) design paradigm

- Make use of design patterns

- Predict how the choice of transaction policy affects rollback scope

- Apply a container-managed security policy

This workbook provides the lab exercises for the modules in the Student Guide.

Lab Overview

All of the hands-on exercises in this course are based on the same sample application. This application is a rudimentary stock trading system with a web-based user interface. The application is unrealistic in the ways necessary to make it possible for you to complete a full, working application in the limited time available.

Despite its simplicity, the exercise application demonstrates many important features of the Java EE platform, including:

- Two types of Enterprise JavaBeans™ (EJB™) components

- The Java Persistence API

- Servlets and JSP technology-based components

- JSF Facelet pages

- Asynchronous messaging

- Design patterns

- The use of an IDE tool to simplify assembling and deploying an application

The application strictly follows the multi-tier model of the Java EE platform. Each tier can be tested independently, and no tier accepts input from, or produces output to, more than one other tier. In the first exercises, you create web components. You test these components using a web browser. The user interface developed allows you to test business components as they are developed. In a later exercise, you use entity classes to persist data in a relational database table. When both the business logic and presentation logic are working, you apply an authorization policy to secure the application.

Lab Strategy

The hands-on exercises in this course are not self-contained. That is, you must complete each exercise before you can move to the next exercise. If you do not want to complete a particular exercise, or feel that you do not have time to complete an exercise, then you can catch up using the solution that is provided for the exercise when working on the next exercise.

Solutions are contained in a directory called `solutions` on your workstation:

* On Microsoft systems, this is typically a subdirectory of
  `D:\Labs\student\solutions`

The `solutions` directory contains a subdirectory for each exercise. Each subdirectory contains a solution. The `resources` directory contains any templates or test code that you might need for the exercise.

One of the design philosophies that underlies this course is that you should develop all Java EE platform-related code yourself. Apart from plain Java classes with some business logic, no other Java language code is provided for you to use. The IDE tool generates some of the boilerplate code that all Java EE components require, but apart from this, you are expected to write every line yourself. If you follow the procedures described in this document, you can feel confident that you have developed an application yourself and have used procedures that would be effective in a full-scale Java EE project. Of course, you are welcome to use the code in the solutions as a guide if necessary.

Conventions

The following conventions are used in this course to represent various training elements and alternative learning resources.

Icons

Note – Indicates additional information that can help you, but is not crucial to your understanding of the concept being described. You should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.

GUI Conventions

Table-1 shows the verbs that are used in this workbook to describe the actions that you commonly perform using the graphical user interface (GUI).

Table-1   GUI Verbs

Verb	Action	Example
Choose	To open a menu or initiate a command	Choose New and then choose Java Package.
Click	To press and release a mouse button without moving the pointer	Click New in the Project Manager window. Click Finish.
Double-click	To click a mouse button twice quickly without moving the pointer	Double-click the NetBeans icon on the desktop.
Open	To start or activate an application, or to access a document, file, or folder	Open the IDE. Open a terminal window.
Right-click	To press and release the right mouse button without moving the pointer	Right-click the bank package that you created in the previous step.

Typographical Conventions

Courier is used for the names of commands, files, directories, programming code, and on-screen computer output. For example:

JAR files are normally given names that end in .jar.

Courier is also used to indicate programming constructs, such as class names, methods, and keywords. For example:

For example, the getName and setName methods represent the name property.

Courier bold is used for characters and numbers that you type. For example:

To list the files in this directory, type:
```
# ls
```

Courier bold is also used for each line of programming code that is referenced in a textual description; for example:

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
```

Notice the javax.servlet interface is imported to allow access to its life-cycle methods (Line 2).

Courier italics is used for variables and command-line placeholders that are replaced with a real name or value. For example:

To delete a file, use the rm *filename* command.

Courier italic bold is used to represent variables whose values are to be entered by the student as part of an activity. For example:

Type **chmod a+rwx *filename*** to grant read, write, and execute rights for *filename* to world, group, and users.

Palatino italics is used for book titles, new words or terms, or words that you want to emphasize. For example:

Read Chapter 6 in the *User's Guide*.
Each service uses a *state engine* that acts like a protocol checker.

Additional Conventions

Java programming language examples use the following additional conventions:

- Method names are not followed with parentheses unless a formal or actual parameter list is shown; for example:

  "The doIt method..." refers to any method called doIt.

  "The doIt() method..." refers to a method, called doIt, that takes no arguments.

- Line breaks occur only where there are separations (commas), conjunctions (operators), or white space in the code. Broken code is indented four spaces under the starting code.

Lab 1

Placing the Java™ EE Model in Context

Objectives

Upon completion of this lab, you should be able to:

- Categorize Java EE services
- Describe the Java EE platform layers
- Explain the existing Java SE **BrokerTool** project

Exercise 1: Categorizing Java EE Services

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

Preparation

No preparation is needed for this exercise.

Task

Use Table 1-1 to place each Java EE service in the appropriate service category:

- Persistence
- Scalability
- Naming
- Threading

- Remote object communication
- Connector
- Load Balancing
- Failover

- Security
- Life-cycle services
- Transaction
- Messaging

Table 1-1 Service Categories Exercise

Service Category	Java EE Services
Deployment-based services	• • •
API-based services	• • •
Inherent services	• • •

Table 1-1   Service Categories Exercise (Continued)

Service Category	Java EE Services
Vendor-specific functionality	• • •

Exercise 2: Describing the Java EE Platform Layers

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

Preparation

No preparation is needed for this exercise.

Task

Match the number of each description with the corresponding layer in Figure 1-1:

1. Databases and other back-end services

2. API layer

3. Service layer

4. Component layer

Vendor-neutral

Defined by the Java EE
specification (vendor-neutral)

Vendor-specific
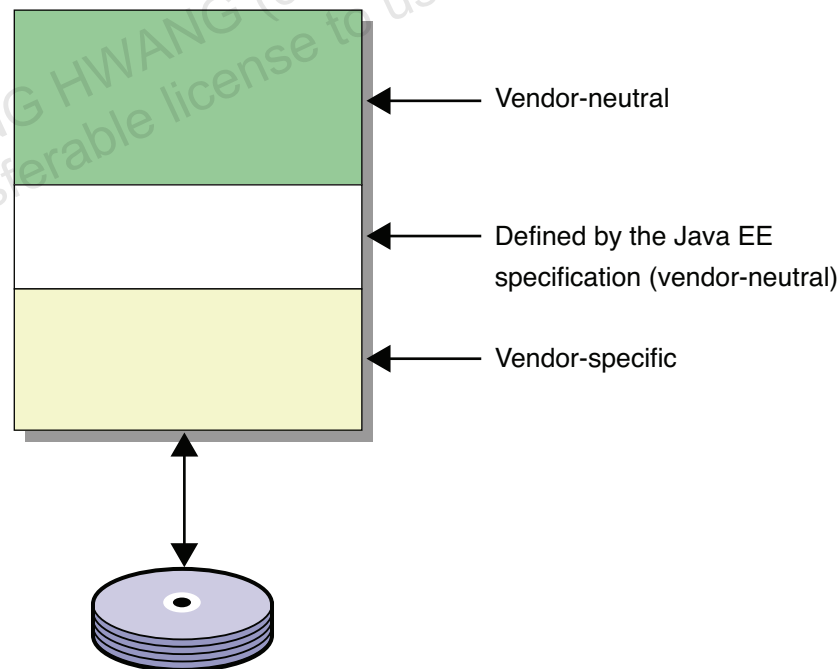
Figure 1-1   Layers Exercise

Exercise 3: Examining the Java SE BrokerTool Application

In this exercise, you examine the existing **BrokerTool_SE** project. The **BrokerTool_SE** project is a Java Platform, Standard Edition (Java SE) Netbeans™ project. This project is similar in function to the project you will be working on all week.

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.

This exercise contains the following sections:

- "Task 1 – Opening the BrokerTool_SE Project"
- "Task 2 – Building the BrokerTool_SE Project"
- "Task 3 – Running the BrokerTool_SE Project"

Preparation

This exercise assumes that Netbeans is installed and the **BrokerTool_SE** project is present on your system.

Task 1 – Opening the BrokerTool_SE Project

Tool Reference – Java Development: Java Application Projects: Opening Projects

Complete the following steps:

1. Open the **BrokerTool_SE** project:

    - Project Location: **D:\Labs\student\projects**
    - Project Name: **BrokerTool_SE**
    - Set as Main Project: **(checked)**

Note – *D:\Labs* represents the directory your student lab files are installed in.

Task 2 – Building the BrokerTool_SE Project

Tool Reference – Java Development: Java Application Projects: Building Projects

Complete the following step:

1.   Build the **BrokerTool_SE** project.

Task 3 – Running the BrokerTool_SE Project

Tool Reference – Java Development: Java Application Projects: Running Projects

Complete the following steps:

1.   Run the **BrokerTool_SE** project.

2.   Using the All Customers tab, view the list of customers. Write down several Customer IDs.

3.   Click the Customer Details tab. Using a Customer ID that you wrote down, complete the Customer Identity field and press the Get Customer button.

4.   Try the other buttons.

5.   Quit the **BrokerTool_SE** application.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

• Experiences

• Interpretations

• Conclusions

• Applications

Exercise Solutions

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

Solution for Exercise 1: Categorizing Java EE Services

Compare your answers to the service category and Java EE services shown in Table 1-2.

Table 1-2   Service Categories Exercise

Service Category	Java EE Services
Deployment-based services	• *Persistence* • *Transaction* • *Security*
API-based services	• *Naming* • *Messaging* • *Connector*
Inherent services	• *Life-cycle services* • *Threading* • *Remote object communication*
Vendor-specific functionality	• *Scalability* • *Failover* • *Load balancing*

Solution for Exercise 2: Describing the Java EE Platform Layers

Compare your answers to the number and description of the layers of Figure 1-2.

4. Component Layer	←	Vendor-neutral
2. API Layer	←	Defined by the Java EE specification (vendor-neutral)
3. Service Layer	←	Vendor-specific

1. Databases and other back-end services

Figure 1-2   Layers Exercise

Lab 2

Java EE Component Model and Development Steps

Objectives

Upon completion of this lab, you should be able to:

- Describe Java EE roles and responsibilities
- Describe the options for packaging applications

Exercise 1: Describing Java EE Roles and Responsibilities

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

Preparation

No preparation is needed for this exercise.

Task

Using Table 2-1, write the number and description of each of the following Java EE roles in the left column of the table, next to the responsibility that best describes the role.

1. Application component provider

2. Application assembler

3. Deployer

4. System administrator

5. Tool provider

6. Product provider

Table 2-1   Java EE Roles and Responsibilities

Role	Responsibility
	Resolves references to external resources, and configures the run-time environment of the application
	Is the vendor of the application server
	Maintains and monitors the application server environment
	Implements development, packaging, assembly, and deployment tools
	Develops EJB components and web components
	Resolves cross-references between components

Developing Applications for the Java™ EE Platform
Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision F

Exercise 2: Describing Options for Packaging Applications

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

Preparation

No preparation is needed for this exercise.
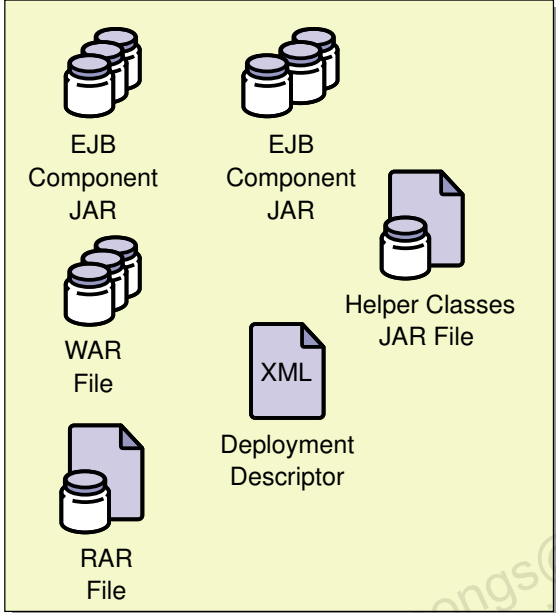
Task

For each figure shown in the File Contents column of the following table, write 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

Table 2-2   Options for Packaging Applications

File Contents	Type of File

Table 2-2   Options for Packaging Applications (Continued)

File Contents	Type of File

Table 2-2 Options for Packaging Applications (Continued)

File Contents	Type of File

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Exercise Solutions

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

Solution for Exercise 1: Describing Java EE Roles and Responsibilities

Table 2-3 shows the answers for the Java EE roles and responsibilities matching activity.
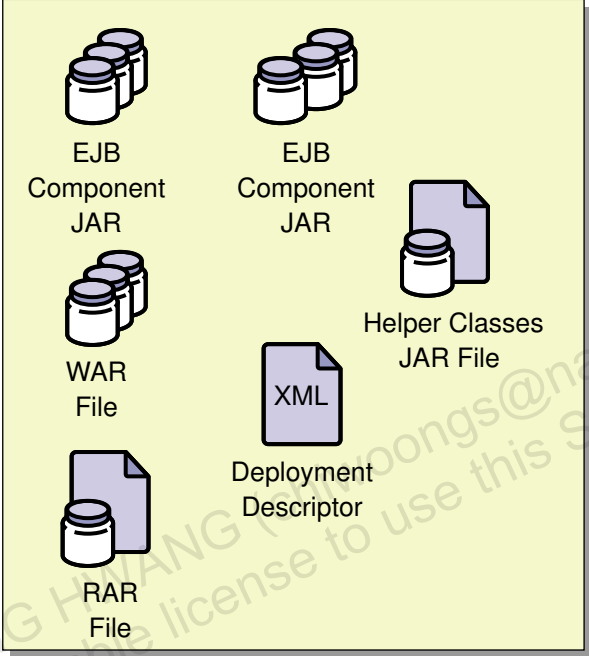
Table 2-3   Java EE Roles and Responsibilities

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

Solution for Exercise 2: Describing Options for Packaging Applications
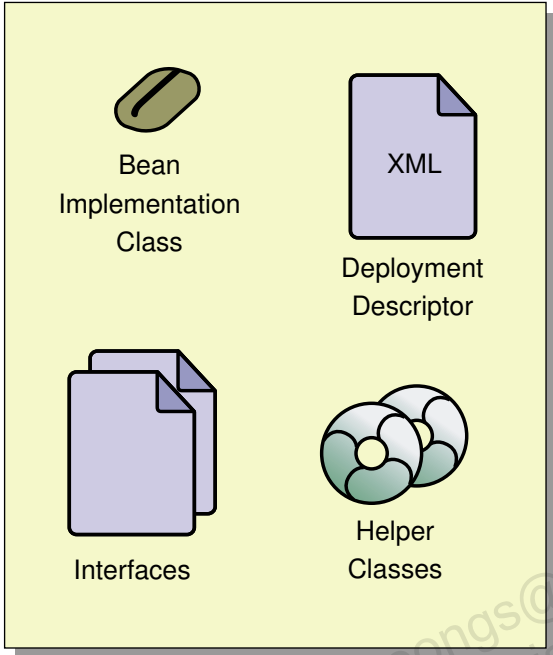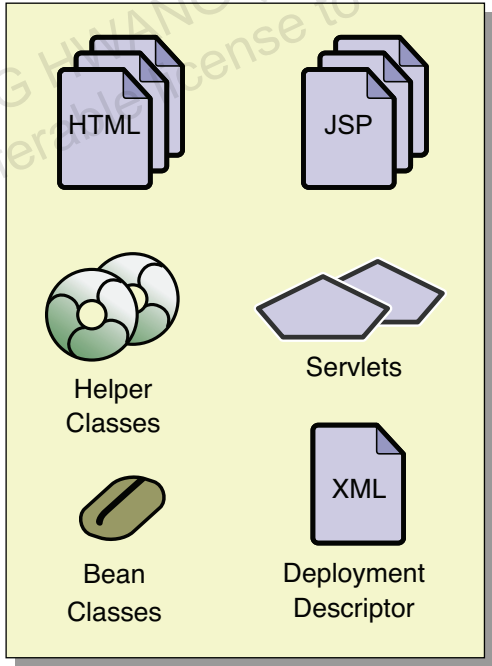
Table 2-4 shows the answers for the matching activity for packaging applications.

Table 2-4 Options for Packaging Applications

File Contents	Type of File
EJB Component JAR EJB Component JAR WAR File Helper Classes JAR File XML Deployment Descriptor RAR File	*EAR file*

Table 2-4   Options for Packaging Applications (Continued)

File Contents	Type of File
	EJB component JAR file
	Web component, web archive, or WAR file

Lab 3

Web Component Model

Objectives

Upon completion of this lab, you should be able to:

- 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

Introduction

In this lab, you create two web components, a servlet and a JSP component that will perform in a typical Hello World fashion. You can use these components as templates in future labs.

Developing Applications for the Java™ EE Platform

Exercise 1: Creating a Basic JSP Component

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

This exercise contains the following sections:

- "Task 1 – Developing a Basic JSP Page"

- "Task 2 – Deploying and Testing the Sample Web Application"

Preparation

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

Task 1 – Developing a Basic JSP Page

Tool Reference – Java EE Development: Web Applications: Web Application Projects: Creating a Web Application Project

Complete the following steps from NetBeans:

1. From the menu select *File* then *New Project*.

2. Under Categories, click Java *Web*.

3. Under Project, click *Web Application*.

4. Click the *Next* button.

5. The new Web Application project should have the following characteristics:

    - Project Name: **SampleWebApplication**

    - Project Location: **D:\Labs\student\projects**

    - Use Dedicated Folder for Storing Libraries: (unchecked)

    - Set as Main Project: **(checked)**

6. Click the *Next* button.

7. The *Server and Settings* dialog should have the following information:

    - Server: **GlassFish v3 Domain**

    - Java EE Version: **Java EE 6 Web**

    - Context Path: **/SampleWebApplication**

8. Click the *Finish* button.

9. 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.

    The following is an example of what you might enter:

    ```
    <%= new java.util.Date() %>
    ```

Task 2 – Deploying and Testing the Sample Web Application

Complete the following steps:

1. Save any modified files. If Deploy on Save is not enabled then deploy the **SampleWebApplication** Web project by right-clicking on the project folder in NetBeans and selecting *Deploy*.

2. Test the application by right-clicking on the project folder and selecting *Run*, or by pointing a web browser at:

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

Exercise 2: Troubleshooting a Web Application

This exercise contains the following sections:

- "Task 1 – Creating a Faulty Web Component"
- "Task 2 – Deploying and Testing the Faulty Web Application"
- "Task 3 – Viewing Application Server Error Messages"

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

Preparation

This exercise assumes that the previous exercise has been completed.

Task 1 – Creating a Faulty Web Component

Complete these steps:

1. 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();
   %>
   ```

Task 2 – Deploying and Testing the Faulty Web Application

Complete the following steps:

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

2. Test the application by selecting *Run* or pointing a web browser at:

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

   Notice the error message displayed in your web browser.

Task 3 – Viewing Application Server Error Messages

Tool Reference – Server Resources: Java EE Application Servers: Examining Server Log Files

Complete the following steps:

1.   Bring up the Application Server log in the IDE. The log will be displayed in a *GlassFish v3 Domain* tab in the output pane.

2.   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
```

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

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

Exercise 3: Creating a Basic Servlet Component

This exercise contains the following sections:

- "Task 1 – Coding the Servlet"

- "Task 2 – Deploying and Testing the Application"

In this exercise you create a basic servlet.

Preparation

This exercise assumes that the application server is installed and running.

Task 1 – Coding the Servlet

Complete the following steps:

1. Right-click the **SampleWebApplication** project select *New* then *Servlet*.

2. Enter for the following information for the servlet:

   - Class Name: **BasicServlet**

   - Project: **SampleWebApplication**

   - Location: **Source Packages**

   - Package: **test**

3. Click *Next*. Do **NOT** check the box to *Add information to deployment descriptor*. By leaving the box unchecked you are instructing the IDE to add configuration annotations. This eliminates the need for the web.xml configuration file.

4. Click *Finish*.

5. Modify the `processRequest` method `BasicServlet` servlet to display a dynamically generated message. First, 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();
    }
}
```

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

7. 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 {
```

8. 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.

Task 2 – Deploying and Testing the Application

Complete the following steps:

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

2. Test the application by selecting *Run* or by pointing a web browser at:

   ```
   http://localhost:8080/SampleWebApplication/BasicServlet
   ```

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

Exercise 4: Describing Web Components

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

Preparation

No preparation is needed for this exercise.

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.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Exercise Solutions

Use the following solutions to check your answers to the exercises in this lab.
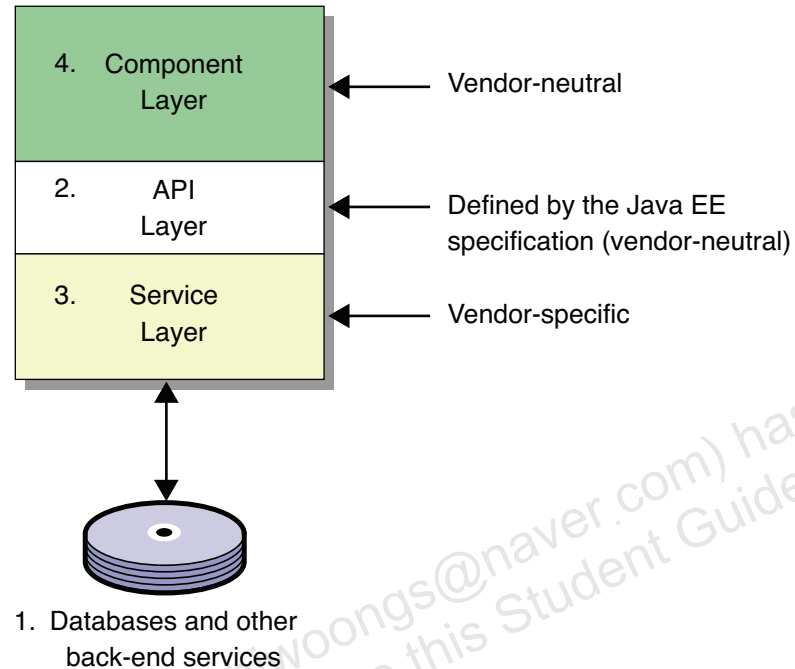
Solutions for Exercise 1 Through Exercise 3

You can find example solutions for the lab exercises in the following directory:
`solutions/WebComponents/`

Solution for Exercise 4: Describing Web Components

Compare your fill-in-the-blank responses to 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.

Lab 4

Developing Servlets

Objective

Upon completion of this lab, you should be able to

- Create a Java EE Web Application Project in NetBeans
- Create servlets to dynamically process form data
- Describe controller components

Introduction

In this lab, you create the **BrokerTool** Web Application project that is used for most of the exercises remaining in this course. You create a controller servlet that handles form submission.

Figure 4-1 shows an example of what the user interface will resemble.
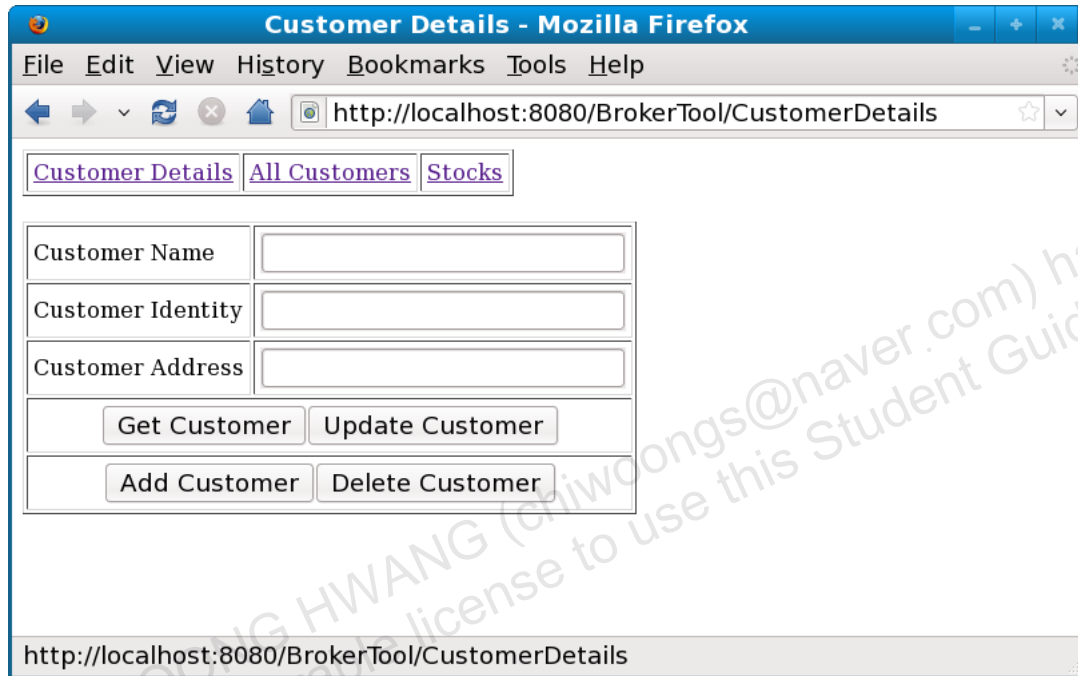
Figure 4-1   The `trader.web.CustomerDetails` Servlet Output

Exercise 1: Exploring the Customer View as implemented by the CustomerDetails Servlet

This exercise contains the following sections that describe the tasks to create the **BrokerTool** Web Application, copy provided Java classes, and to make the CustomerDetails servlet the default index page:

- "Task 1 – Creating the BrokerTool Project"

- "Task 2 – Copying the BrokerTool_SE classes"

- "Task 3 – Copying the CustomerDetails Servlet"

- "Task 4 – Setting the BrokerTool default home page"

- "Task 5 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Creating the BrokerTool Project

Tool Reference – Java EE Development: Web Applications: Web Application Projects: Creating a Web Application Project

Complete the following steps from NetBeans:

1. From the menu select *File* then *New Project*.

2. Under Categories, click Java *Web*.

3. Under Project, click *Web Application*.

4. Click the *Next* button.

5. The new Web Application project should have the following characteristics:

   - Project Name: **BrokerTool**

   - Project Location: **D:\Labs\student\projects**

   - Use Dedicated Folder for Storing Libraries: (unchecked)

   - Set as Main Project: **(checked)**

6. Click the *Next* button.

7. The Server and Settings dialog should have the following information:

   • Server: **GlassFish v3 Domain**

   • Java EE Version: **Java EE 6 Web**

   • Context Path: **/BrokerTool**

8. Click the *Finish* button.

9. An index.jsp file is created for you automatically. You will not use the index.jsp file in this project, delete it.

Task 2 – Copying the BrokerTool_SE classes

Complete the following steps:

1. Right-click the **BrokerTool** project icon.

2. Select *New* then *Java Package*.

3. Enter trader for the package name.

4. Click the *Finish* button.

5. Copy classes from the **BrokerTool_SE** project to the trader package in the **BrokerTool** project.

   a. Open the **BrokerTool_SE** project.

   b. Expand *Source Packages*, then *trader*.

   c. To copy a class, right-click the class you want to copy in the **BrokerTool_SE** *trader* package and select *Copy*. Next, right-click on the target *trader* package in the **BrokerTool** project and select *Paste*, then *Refactor Copy*. A confirmation dialog appears. Choose *Refactor*. The copy of the class is complete.

   d. Copy the following classes:

      • trader.BrokerException

      • trader.BrokerModel

      • trader.BrokerModelImpl

      • trader.Customer

      • trader.CustomerShare

      • trader.Stock

6. Close the **BrokerTool_SE** project.

Task 3 – Copying the CustomerDetails Servlet

1. Right-click the **BrokerTool** project icon.

2. Select *New* then *Java Package*.

3. Enter trader.web for the package name.

4. Enable the Favorites tab if it is not displayed already. From the *Window* menu select *Favorites*.

5. Add the D:\Labs\student\resources directory to the *Favorites* window if it is not already present. Right-click in an empty area in the Favorites window and select *Add to Favorites*. From the file chooser dialog box find and select the D:\Labs\student\resources directory.

Note – You can also add the D:\Labs\student\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.

6. In the *Favorites* window copy the CustomerDetails.java file from *resources -> brokertool* to the clipboard.

7. In the *Projects* window paste the CustomerDetails.java file into the trader.web package of the **BrokerTool** project.

8. 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 lab. In the lab for the next module you will replace the CustomerDetails servlet with a JSP based view.

Task 4 – Setting the BrokerTool default home page

Tool Reference – Java EE Development: Web Applications: Web Deployment Descriptors: Creating the Standard Deployment Descriptor

Since the index.jsp file was deleted a 404 error will be generated when visiting http://localhost:8080/BrokerTool/. To fix this a new welcome page must be set using a deployment descriptor.

1. Open the New File dialog and specify the following values:

    •   Project: **BrokerTool**

    •   Categories: **Web**

    •   File Types: **Standard Deployment Descriptor (web.xml)**

2. Press Next and Finish. The web.xml deployment descriptor should be created and opened.

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

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

5. 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>
```

Task 5 – Configuring, Deploying, and Testing the Application

Complete the following steps:

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

2. Run the application by pointing a browser at:

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

    You should see an empty customer details form.

Exercise 2: Implementing Controller Components

This exercise contains the following sections that describe the tasks to create the `CustomerController` and `PortfolioController` servlet:

• "Task 1 – Creating the `CustomerController` Servlet"

• "Task 2 – Creating the `PortfolioController` Servlet"

• "Task 3 – Configuring, Deploying, and Testing the Application"

Preparation

This exercise assumes that the application server is installed and exercise 1 has been completed.

Task 1 – Creating the `CustomerController` Servlet

Complete the following steps:

1. Create a new servlet in the **BrokerTool** project.

2. In the Name and Location dialog, enter the following information:

    *   Class Name: **CustomerController**

    *   Location: **Source Packages**

    *   Package: **trader.web**

3. Click the *Finish* button.

4. Using the `@WebServlet` annotation and the `urlPatterns` attribute make the servlet available at two URLs:

    *   `/CustomerController`

    *   `/AllCustomers`

5. Import the contents of the `trader` package.

6. Import `javax.servlet.RequestDispatcher`.

7. Use the following steps to code the `processRequest` method:

    a. Remove all code from the `processRequest` method. Remove any unused imports.

    b. Retrieve the singleton instance of the `BrokerModelImpl` class by adding the following line:

    `BrokerModel model = BrokerModelImpl.getInstance();`

    c. Use the `HttpServletRequest` object to get the path used to invoke the servlet.

    `String path = request.getServletPath();`

    d. If the path was `/CustomerController` perform the following:

        1. Retrieve the request form parameter values for `customerIdentity`, `customerName`, `customerAddress` and `submit`. Assign the values to String variables `id`, `name`, `address`, and `submit`.

        2. Use the value of the `submit` variable to determine if one of the submit buttons in the `CustomerDetails` servlet was pressed to invoke this servlet, perform the following actions if so:

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

- If the *Update Customer* submit button was pressed, 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).

- If the *Add Customer* submit button was pressed, 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).

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

- 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`.

- Use a `RequestDispatcher` to forward the response to the `CustomerDetails` servlet.

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

e. If the path `/AllCustomers` was used to invoke this servlet, perform the following actions:

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

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

- 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`.

- Use a `RequestDispatcher` to forward the response to `AllCustomers.jsp`.

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

Task 2 – Creating the `PortfolioController` Servlet

The `PortfolioController` servlet is used in the next module. It is designed to retrieve a customer's portfolio and forward that data to a `Portfolio.jsp` for display.

Complete the following steps:

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

2. In the name and location dialog, enter the following information:

    • `Class Name:` **PortfolioController**

    • `Location:` **Source Packages**

    • `Package:` **trader.web**

3. Click the *Finish* button.

4. Using the `@WebServlet` annotation set the URL of the `PortfolioController` servlet to `/PortfolioController`.

5. Import the contents of the `trader` package.

6. Import `javax.servlet.RequestDispatcher`.

7. Implement the `processRequest` method as follows:

You can use the `PortfolioController` template located at `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);
    }
```

Task 3 – Configuring, Deploying, and Testing the Application

Complete the following steps:

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

2. Run the application or test the servlet by pointing a browser at:

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

3. 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 on the customer details page. Links such as *View Portfolio*, *All Customers*, and *Stocks.xhtml* will not function yet. Fix any errors that occur when using the Customer Details form.

Exercise 3: Describing Servlet Components

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

Preparation

No preparation is needed for this exercise.

Task

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

1. The _____ package contains the HTTP-specific servlet classes.

2. 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.

3. The method signature of the initialization method inherited from `HttpServlet`, which is recommended for use is _____.

4. The _____ method typically calls the `doGet` or `doPost` method.

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

6. In Java EE 5, the _____ annotation can be used in place of the `init` method for a servlet.

7. To read form data, the _____ method of an `HttpServletRequest` is used.

8. _____ class has a forward and include method used to invoke a servlet from within another servlet.

9. In Java EE 6, the _____ annotation can be used in-place of the deployment descriptor to specify a URL for a servlet.

10. Every time a `request.getSession` method is called, the server attempts to send a _____ to the client.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Exercise Solutions

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

Solutions for Exercises 1 and 2

You can find example solutions for the exercises in this lab in the following directory: solutions/Servlets/.

Solution for Exercise 3: Describing Servlet Components

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

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

2. 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.

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

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

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

6. In Java EE 5, the @*PostConstruct* annotation can be used in place of the init method for a servlet.

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

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

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

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

Lab 5

Developing With JavaServer Pages™ Technology

Objectives

Upon completion of this lab, you should be able to:

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

Introduction

In this lab, you create two JSP components. One, the `AllCustomers.jsp`, displays all the customers currently stored in the **BrokerTool** application. The other, `Portfolio.jsp`, displays the type and quantity of stocks for a customer.

The first exercise uses scriplets. The second exercise uses the JSTL and the EL. The second exercise demonstrates the preferred style of JSP coding.

Figure 5-1 shows an example of what the `AllCustomers.jsp` might display.

Figure 5-1   `AllCustomers.jsp` output

Figure 5-2 shows an example of what the `Portfolio.jsp` might display.
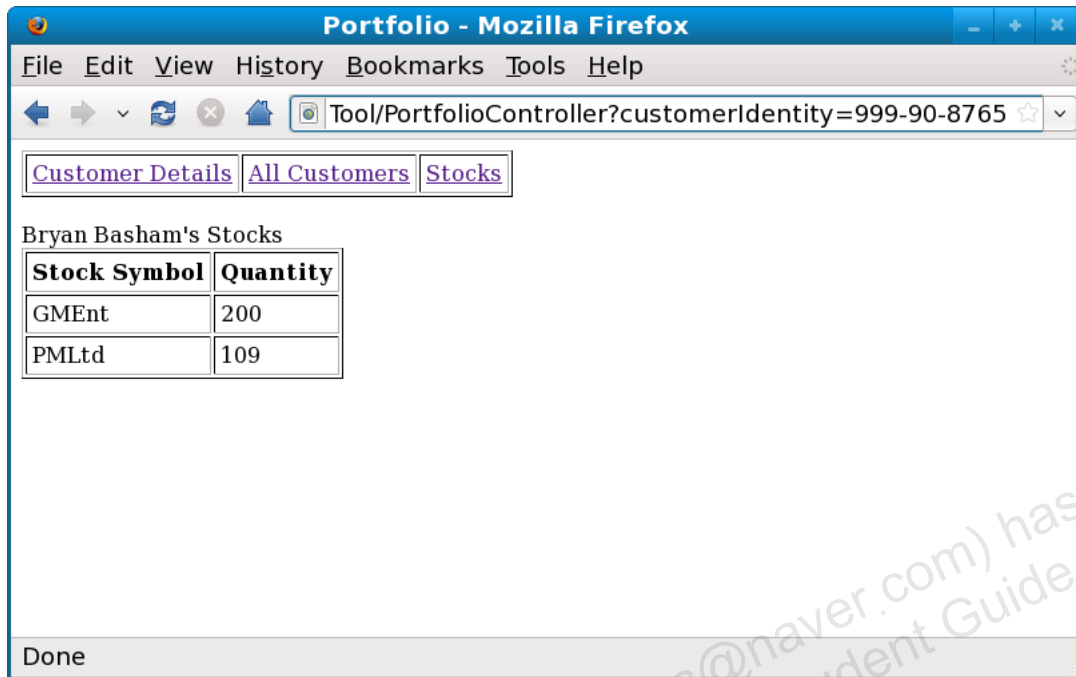
Figure 5-2   `Portfolio.jsp` output

Exercise 1: Creating the `AllCustomers.jsp` Component

This exercise contains the following sections that describe the tasks to create the `AllCustomers.jsp` component which displays the identities, names, and addresses of all customers registered in the **BrokerTool** Application:

• "Task 1 – Creating the `AllCustomers.jsp` Component"

• "Task 2 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Creating the `AllCustomers.jsp` Component

Tool Reference – Java EE Development: Web Applications: Creating JavaServer Pages

Create the `AllCustomers.jsp` component in the **BrokerTool** application as follows:

1. Right-click the **BrokerTool** icon.

2. Select *New* then *JSP*.

3. Enter the following information in the Name and Location dialog:

   • File Name: **AllCustomers**

   • Location: **Web Pages**

   • Folder: **(empty)**

   • Options: **JSP File (Standard Syntax)**

4. Click the *Finish* button.

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

   • Add a page import directive to import the classes in the `trader` package.

   • Make the page title *All Customers*.

- 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>`

- Create a table with the following headers `Customer Id`, `Name`, `Address`, and `Portfolio`.

- 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`.

- 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. Example portfolio link:

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

- Close the table.

- 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>");
    }
%>
```

Task 2 – Configuring, Deploying, and Testing the Application

Complete the following steps:

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

2. Run the application. Test your JSP by pointing a browser at:

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

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

Exercise 2: Creating the `Portfolio.jsp` Component

This exercise contains the following sections that describe the tasks to create the `Portfolio.jsp` component, which displays the symbols and quantities of stocks owed by the customer most recently selected on the `CustomerDetails` page:

- "Task 1 – Creating the `Portfolio.jsp` Component"

- "Task 2 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Creating the `Portfolio.jsp` Component

Create the `Portfolio.jsp` component as follows:

1. Create the `Portfolio.jsp` component in the **BrokerTool** project.

2. Enter the following information in the Name and Location Dialog:

   - JSP File Name: **Portfolio**

   - Location: **Web Pages**

   - Folder: **(empty)**

   - Options: **JSP File (Standard Syntax)**

3. Add a page import directive to import the classes in the `trader` package.

4. Add a directive to include the Java Standard Tag library, for example:

   - `<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>`

5. Give the page a title of *Portfolio*.

6. 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>`

7. Create a JSTL `<c:choose>` tag. Inside the `<c:choose>` create a `<c:when test="${...}">` tag and a `<c:otherwise>` tag.

8. 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}`

    -   If there is no message:

        -   Display a message with the customer's name:
            `${requestScope.customer.name}'s Stocks`
            `<br />`

        -   Create a table with `Stock Symbol` and `Quantity` as headers.

        -   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>
```

    -   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 `message` in the request scope.

Task 2 – Configuring, Deploying, and Testing the Application

Complete the following steps:

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

2. Run the application. Test your JSP by pointing a browser at:

    `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 Portfolio* link to view their portfolio.

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

Optional Exercise 3: Creating the `CustomerDetails.jsp` Component

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.

This exercise contains the following sections that describe the tasks to create the `CustomerDetails.jsp` component which displays the identities, names, and addresses of all customers registered in the **BrokerTool** Application:

*   "Task 1 – Creating the `CustomerDetails.jsp` Component"

*   "Task 2 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Creating the `CustomerDetails.jsp` Component

Tool Reference – Java EE Development: Web Applications: Creating JavaServer Pages

Create the `CustomerDetails.jsp` component as follows:

1. Create the `CustomerDetails.jsp` component in the **BrokerTool** project.

2. Enter the following information in the Name and Location Dialog:

    *   JSP File Name: **CustomerDetails**

    *   Location: **Web Pages**

    *   Folder: **(empty)**

    *   Options: **JSP File (Standard Syntax)**

3. View the details of a customer by using the existing `CustomerDetails` servlet at `http://localhost:8080/BrokerTool/`.

4. View the HTML output of the `CustomerDetails` servlet using your web browser.

5. View the behavior of the `CustomerController` servlet.

6. Using the information gathered in step 4 and 5, code the functionality of the `CustomerDetails.jsp`. Use JSTL and EL, avoid scriptlets.

7. Modify the `CustomerController` servlet to forward to the new JSP page.

8. Modify the default welcome page in the `web.xml` deployment descriptor.

9. In all JSP pages, modify any links that point to the `CustomerDetails` servlet to point to `CustomerDetails.jsp`.

Task 2 – Configuring, Deploying, and Testing the Application

Complete the following steps:

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

2. Run the application. Test your JSP by pointing a browser at:

3. `http://localhost:8080/BrokerTool/`

Exercise 4: Describing JavaServer Page Components

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

Preparation

No preparation is needed for this exercise.

Task

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.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Exercise Solutions

This section contains the exercise solutions.

Solutions for Exercises 1 and 2

You can find example solutions for the exercises in this lab in the following directory: `solutions\JSPs\`.

Solution for Exercise 4: Describing JavaServer Page Components

Compare your responses to 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.

Exercise Solutions

Developing Applications for the Java™ EE Platform

Lab 6

Developing With JavaServer Faces Technology

Objectives

Upon completion of this lab, you should be able to:

- 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

Introduction

In this lab, you create a JSF Facelet page and its corresponding backing bean. The Facelet, `Stocks.xhtml`, displays all the stocks currently stored in the **BrokerTool** application.

Figure 6-1 shows an example of what the `Stocks.xhtml` might display.
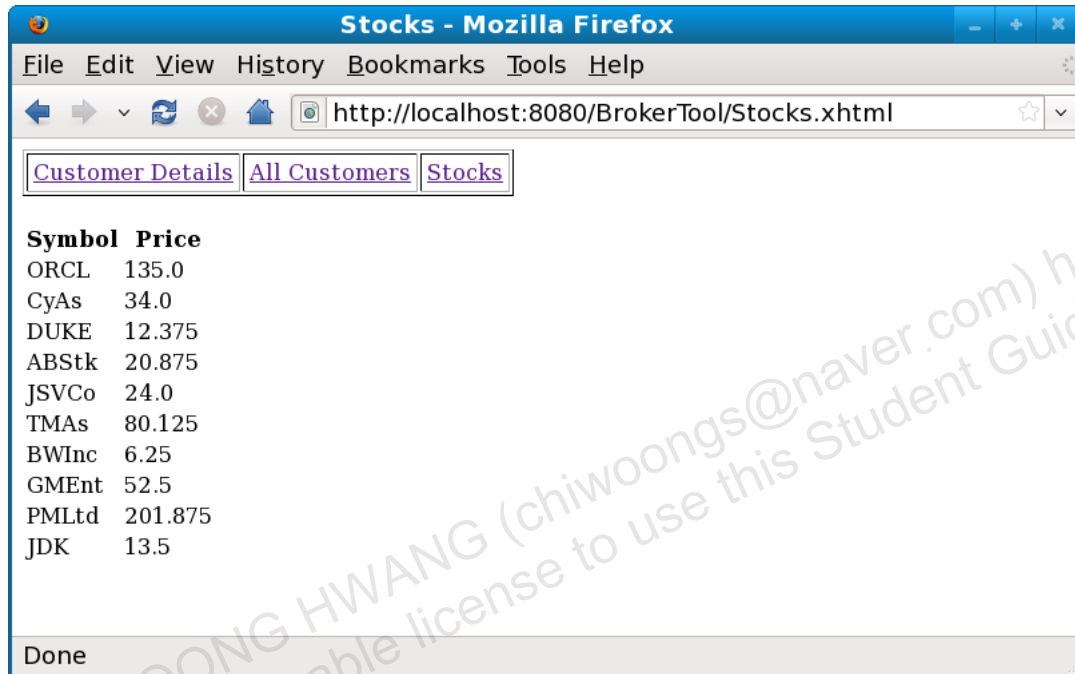
Figure 6-1   `Stocks.xhtml` output

Exercise 1: Creating the Stocks.xhtml Component

This exercise contains the following sections that describe the tasks to create the Stocks.xhtml component which displays the names, and prices of all stocks in the **BrokerTool** Application:

- "Task 1 – Configure the JSF Facelet Servlet"

- "Task 2 – Creating the StocksManagedBean JSF component"

- "Task 3 – Creating the Stocks.xhtml Facelet Page"

- "Task 4 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Configure the JSF Facelet Servlet

Tool Reference – Java EE Development: Web Applications: Web Deployment Descriptors: Opening the Standard Deployment Descriptor

Tool Reference – Java EE Development: Web Applications: Web Deployment Descriptors: Servlet Configuration

Configure the JSF Facelet Servlet as follows:

1. Open the web.xml deployment descriptor.

2. Add a new Servlet Element.

   - Servlet Name: **Faces Servlet**

   - Servlet Class: **javax.faces.webapp.FacesServlet**

   - URL Pattern(s): ***.xhtml**

   This creates a servlet entry in NetBeans. The configuration form for this servlet is displayed in the NetBeans.

3. In the Faces Servlet form, set the **Startup Order** to 1.

4. Click on the **XML** button.

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>
```

Task 2 – Creating the `StocksManagedBean` JSF component

Tool Reference – Java EE Development: JavaServer Faces: Managed Beans: Creating Managed Beans

Create the `StocksManagedBean` class in the **BrokerTool** application as follows:

1. Open the *New File* dialog and specify the following values:

    • Categories: **JavaServer Faces**

    • File Types: **JSF Managed Bean**

2. Press Next. Enter the following values in the *Name and Location* dialog:

    • Class Name: **StocksManagedBean**

    • Project: **BrokerTool**

    • Location: **Source Packages**

    • Package: **trader.web**

    • Name: **stocks**

    • Scope: **request**

3. Click the *Finish* button.

4. Add an instance variable of type `BrokerModel` in the `StocksManagedBean` class.

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

5. Declare a the following method:

```
public Stock[] getAllStocks() { }
```

6. Add any required import statements.

7. Implement the getAllStocks method as follows:

   • Retreive an array of all Stock objects using the model variable.

   • Catch any exceptions that occur. Return null if an Exception occurs.

Task 3 – Creating the Stocks.xhtml Facelet Page

Tool Reference – Java EE Development: JavaServer Faces: Working with JSF Pages: Creating JSF Pages

Create the Stocks.xhtml Facelet page in the **BrokerTool** application as follows:

1. Open the *New File* dialog and specify the following values:

   • Project: **BrokerTool**

   • Categories: **JavaServer Faces**

   • File Types: **JSF Page**

2. Press Next. Enter the following values in the *Name and Location* dialog:

   • File Name: **Stocks**

   • Project: **BrokerTool**

   • Location: **Web Pages**

   • Folder: **(empty)**

   • Options: **Facelets (selected)**

3. Click the *Finish* button.

4. 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">
```

5. Modify the Stocks.xhtml page as follows:

   • Specify a value of *Stocks* for the page title.

   • Delete the content of the existing body tag.

   • Create a table based navigational menu along the top that includes links to:

      • Customer Details

- • `<a href='AllCustomers'>All Customers</a>`
- • `<a href='Stocks.xhtml'>Stocks</a>`
- • Use the `h:dataTable` tag to display the symbol and price of all stocks.

Task 4 – Configuring, Deploying, and Testing the Application

Complete the following steps:

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

2. Run the application. Test your JSF page by pointing a browser at:

   `http://localhost:8080/BrokerTool/Stocks.xhtml`

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

Optional Exercise 2: Implementing the JSF `CustomerDetails.xhtml` View

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.

This exercise contains the following sections that describe the tasks to create the `CustomerDetails.xhtml` page which displays the identities, names, and addresses of all customers registered in the **BrokerTool** Application:

- "Task 1 – Creating the `CustomerManagedBean` JSF component"

- "Task 2 – Creating the `CustomerDetails.xhtml` Facelet Page"

- "Task 3 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Creating the `CustomerManagedBean` JSF component

Tool Reference – Java EE Development: JavaServer Faces: Managed Beans: Creating Managed Beans

Create the `CustomerManagedBean` class in the **BrokerTool** application as follows:

1. Open the *New File* dialog and specify the following values:

   - Categories: **JavaServer Faces**

   - File Types: **JSF Managed Bean**

2. Press Next. Enter the following values in the *Name and Location* dialog:

   - Class Name: **CustomerManagedBean**

   - Project: **BrokerTool**

   - Location: **Source Packages**

   - Package: **trader.web**

- • Name: **customerDetails**

- • Scope: **request**

3. Click the *Finish* button.

4. Add the following import statements to the class.

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

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

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

6. 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 = "";`

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

7. Create controller methods as instructed below.

   a. All methods should use the variables from step 5 and the `model` variable from step 4 during execution.

   b. In the event of an a `BrokerException` the `message` variable should be set to the value of `Exception.getMessage()`.

   c. Return a `String` value of *CustomerDetails*.

   d. Add the method signatures listed below. Implement the functionality of the methods as indicated by the method name.

      - • `public String retrieveCustomer()`

      - • `public String updateCustomer()`

      - • `public String addCustomer()`

      - • `public String deleteCustomer()`

Task 2 – Creating the `CustomerDetails.xhtml` Facelet Page

Tool Reference – Java EE Development: JavaServer Faces: Working with JSF Pages: Creating JSF Pages

1. Open the *New File* dialog and specify the following values:

   • Project: **BrokerTool**

   • Categories: **JavaServer Faces**

   • File Types: **JSF Page**

2. Press Next. Enter the following values in the *Name and Location* dialog:

   • File Name: **CustomerDetails**

   • Project: **BrokerTool**

   • Location: **Web Pages**

   • Folder: **(empty)**

   • Options: **Facelets (selected)**

3. Click the *Finish* button.

4. 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.

Task 3 – Configuring, Deploying, and Testing the Application

Complete the following steps:

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

2. Run the application. Test your JSP by pointing a browser at:
   `http://localhost:8080/BrokerTool/CustomerDetails.xhtml`

Note – 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.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Exercise Solutions

This section contains the exercise solutions.

Solutions for Exercises 1 and 2

You can find example solutions for the exercises in this lab in the following directory: `solutions\JSF\`.

Lab 7

EJB™ Component Model

Objectives

Upon completion of this lab, you should be able to:

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

Introduction

The purpose of these exercises 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.

Later labs demonstrate the essential features of the Java Naming and Directory Interface™ (J.N.D.I. or JNDI) API as an alternative way to locate EJB references.

Do not worry about the details of the EJB used in this application. Session Beans will be explained in detail later.

Exercise 1: Creating and Deploying a Simple EJB Application

In this exercise, you create and deploy a simple EJB module and EJB client.

This exercise contains the following sections:

- "Task 1 – Creating an EJB Application Module"
- "Task 2 – Creating a Basic Session EJB"
- "Task 3 – Adding a Business Method to the SimpleSession EJB"
- "Task 4 – Adding the EJB Project to the Libraries of the SimpleWebApplication"
- "Task 5 – Using Annotations in the BasicServlet to Look Up and Use the Simple Session EJB"
- "Task 6 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Creating an EJB Application Module

Tool Reference – Java EE Development: EJB Modules: Creating EJB Modules in a Java EE Application

Complete the following steps to create a new EJB Module project.

1. From the NetBeans menu select *File* then *New Project*.

2. Select *Java EE*, then *EJB Module*.

3. Click *Next*.

4. Enter the following information in the *Name and Location* dialog.

    - Project Name: **SampleEJBApplication**
    - Project Location: **D:\Labs\student\projects**
    - Use Dedicated Folder for Storing Libraries: (unchecked)
    - Set as Main Project: (unchecked)

5.   Click *Next*.

6.   In the *Server and Settings* dialog enter the following information:

- Server: **GlassFish v3 Domain**

- Java EE Version: **Java EE 6**

7.   Click *Finish*.

Task 2 – Creating a Basic Session EJB

Tool Reference – Java EE Development: EJB Modules: Session Beans: Creating Session Beans

Complete the following steps to create a new session bean in the **SampleEJBApplication** project:

1.   Right click the **SampleEJBApplication** project and select *New* then *Session Bean*.

2.   In the Name and Location dialog, enter the following information:

- EJB Name: **BasicSession**

- Location: **Source Packages**

- Package: **test**

- Session Type: **Stateless**

- Create Interface: **Remote**

3.   Click *Finish*.

Task 3 – Adding a Business Method to the `SimpleSession` EJB

Tool Reference – Java EE Development: EJB Modules: Session Beans: Adding Business Methods

Complete the following steps:

1.   Add a method signature in `BasicSessionRemote.java`.

```
String getMessage();
```

2.   Add a business method to `BasicSessionBean.java`. The method should be:

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

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

Task 4 – Adding the EJB Project to the Libraries of the SimpleWebApplication

Tool Reference – Java Development: Java Application Projects: Modifying Project Libraries

The servlet client requires a copy of the BasicSessionRemote class. Add the **SampleEJBApplication** project as a library of the **SampleWebApplication** project. Complete the following steps:

1. Load the **SampleWebApplication** project.

2. Right-click the *Libraries* folder.

3. Select *Add Project*.

4. Navigate to and select the **SampleEJBApplication** project.

5. Click *Add Project JAR Files*.

Task 5 – Using Annotations in the BasicServlet to Look Up and Use the Simple Session EJB

In the BasicServlet of the **SampleWebApplication** project, complete the following steps:

1. Import the javax.ejb.* classes. Most EJB annotations reside in this package.

2. 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;

3. 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());

Task 6 – Configuring, Deploying, and Testing the Application

Complete the following steps:

> **Note –** Do **NOT** deploy the **SampleEJBApplication** EJB module. Since the **SampleEJBApplication** is a library it will be archived as a JAR file and placed inside of the WAR for the **SampleWebApplication**. Java EE 6 application servers will deploy EJB components that exist within library JAR files of web archives. Attempting to deploy both projects will result in an error.

1. Save any modified files. If Deploy on Save is not enabled then deploy the **SampleWebApplication** WAR module manually.

2. Test your modified servlet and new EJB by pointing a browser at:

```
http://localhost:8080/SampleWebApplication/BasicServlet
```

You should see the *Hello EJB World* message displayed.

Exercise 2: Describing the EJB Component Model

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

Preparation

No preparation is needed for this exercise.

Task

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

1. The two type of EJBs 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 be 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.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Exercise Solutions

This section contains the exercise solutions.

Solutions for Exercise 1

You can find example solutions for the exercises in this lab in the following directory: `solutions\EJBComponents`

Solution for Exercise 2: Describing the EJB Component Model

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

1. The two type of EJBs 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 be 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.

Lab 8

Developing Session Beans

Objectives

Upon completion of this lab, you should be able to:

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

Introduction

The purpose of these exercises is to learn how to create a singleton session bean as part of a web application project. A singleton EJB is used for in-memory persistence. Later labs will modify the application to use a database for persistence and will allow for a stateless EJB model.

These exercises 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.

Developing Applications for the Java™ EE Platform
Copyright 2010 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision

Exercise 1: Coding the EJB Component and client

This exercise contains the following sections that describe the tasks to code a session EJB component:

- "Task 1 – Modifying the `BrokerModelImpl` Class to be a Local Singleton Session Bean"

- "Task 2 – Modifying the `BrokerModel` Clients to Use the `BrokerModelImpl` Local Session Bean"

- "Task 3 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Modifying 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.

1. 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. 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() {
```

2. Add the following `import` statements:

- `import javax.ejb.Local;`

- • `import javax.ejb.Singleton;`

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

- • `@Local @Singleton`

Task 2 – Modifying 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.

To modify the controller classes, complete the following steps:

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

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

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

- • Remove any lines of code that call `BrokerModelImpl.getInstance()`.

- • Remove any `model` local variable declarations.

- • Create an instance level variable:

`@EJB private BrokerModel model;`

- • Add the needed import for the `@EJB` annotation.

- • If needed, modify any methods that do not compile to use the new `model` variable.

Note – The `@EJB` annotation can not be applied to local variables.

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 lifecycle, security handling, and container managed transactions are lost when calling `new` on a EJB bean class.

Task 3 – Configuring, Deploying, and Testing the Application

Complete the following steps:

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

2. Test your application by pointing a browser at:

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

Optional Exercise 2: Create a Java SE EJB Client that uses JNDI

This exercise contains the following sections that describe the tasks to code a session EJB component:

- "Task 1 – Creating the SampleEJBClient Project"

- "Task 2 – Undeploy the SampleWebApplication Module"

- "Task 3 – Open the SampleEJBApplication Project"

- "Task 4 – Configure the SampleEJBClient Project Libraries"

- "Task 5 – Code the EJB Client Class"

- "Task 6 – Testing the EJB Client Application"

Preparation

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

Task 1 – Creating the SampleEJBClient Project

Tool Reference – Java Development: Java Application Projects: Creating Projects

Complete the following steps to create a new Java SE project.

1. From the NetBeans menu select *File* then *New Project*.

2. Select *Java*, then *Java Application*.

3. Click *Next*.

4. Enter the following information in the *Name and Location* dialog.

    - Project Name: **SampleEJBClient**

    - Project Location: **D:\Labs\student\projects**

    - Use Dedicated Folder for Storing Libraries: (unchecked)

    - Create Main Class: **test.Main**

    - Set as Main Project: (**checked**)

5. Click *Finish*.

Task 2 – Undeploy the SampleWebApplication Module

Undeploy the SampleWebApplication. This step 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.

Tool Reference – Java EE Development: Enterprise Application Projects: Undeploying Java EE Applications

Task 3 – Open the SampleEJBApplication Project

Open the SampleEJBApplication created in lab 7, EJB Component Model. Make sure to build the SampleEJBApplication project.

Task 4 – Configure the SampleEJBClient Project Libraries

Tool Reference – Java Development: Java Application Projects: Modifying Project Libraries

1. 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 v3 server.

   - Library Type: **JAR/Folder**

   - Look In: **sges-v3\glassfish\modules**

   - File Name: **gf-client.jar**

Note – The placement of the GlassFish v3 installation directory, `sges-v3`, may vary depending on choices made during installation. It may be located in `C:\Program Files\` directory. Consult your instructor if needed.

2. Add the SampleEJBApplication as a library to the SampleEJBClient project. The SampleEJBApplication library provides the `BasicSessionRemote` interface to the EJB client.

   • Library Type: **Project**

   • Look In: **D:\Labs\student\projects**

   • Project Name: **SampleEJBApplication**

Task 5 – Code the EJB Client Class

In the `test.Main` class make the following changes:

1. In the `main` method obtain a JNDI Context.

   `Context ctx = new InitialContext();`

2. Use JNDI to perform a lookup of the BasicSessionBean EJB. The syntax of the JNDI name used is `java:global/<module-name>/<bean-name>`.

3. Cast the result of the JNDI lookup to `BasicSessionRemote` and store it in a local variable. This is the EJB stub.

4. Use the EJB stub to call the `getMessage` method of the `BasicSessionBean` and print the result.

5. Catch any exceptions that occur using a try/catch block.

Task 6 – Testing the EJB Client Application

Tool Reference – Java Development: Java Application Projects: Running Projects

To run the SampleEJBClient application. Follow these steps.

1. Build the SampleEJBClient project. Do not run the project yet. Correct any errors that occur.

2. Deploy the the SampleEJBApplication project.

3. Run the SampleEJBClient project.

Note - 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/BasicSessionBean' in
SerialContext,orb'sInitialHost=localhost,orb'sInitialPort=37
00 [Root exception is javax.naming.NameNotFoundException:
SampleEJBApplication]
```

Note - 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.

Exercise 3: Describing Session Beans

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

Preparation

No preparation is needed for this exercise.

Task

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

1. The three type 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 _____.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

• Experiences

• Interpretations

• Conclusions

• Applications

Exercise Solutions

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

Solutions for Exercise 1 and 2

You can find example solutions for the exercises in this lab in the following directory: `solutions\SessionBeans\`.

Solution for Exercise 3: Describing Session Beans

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

1. The three type 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*.

Lab 9

The Java Persistence API

Objectives

Upon completion of this lab, you should be able to:

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

Introduction

In this lab, 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.

Exercise 1: Creating the Java Persistence API Version of the BrokerTool Project

This exercise contains the following sections:

- "Task 1 – Creating the `StockMarket` Database"

- "Task 2 – Creating a Persistence Unit"

- "Task 3 – Converting the `Customer.java` Class to a Java Persistence API Class"

- "Task 4 – Converting `CustomerShare` and `Stock` to Use the Java Persistence API"

- "Task 5 – Modifying the `BrokerModelImpl.java` Class to Use the Java Persistence API"

- "Task 6 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Creating the StockMarket Database

Tool Reference – Server Resources: Databases: Starting the Java DB Database

Complete the following steps:

1.   Start the Java DB database server from NetBeans.

   a.   Click the *Services* tab.

   b.   Open the *Databases* folder.

   c.   Right-click the *Java DB* icon.

   d.   Select *Start Server*.

Note – If *Start Server* is grayed-out that means the database was already started by NetBeans.

Tool Reference – Server Resources: Databases: Creating a Java DB Database

2.   Right-click the *Java DB* icon and select *Create Database*.

3.   Enter the following information for the database:

   •   Database Name: **StockMarket**

   •   User Name: **public**

   •   Password: **public**

4.   Click *OK*. This creates a the database and adds a connection for the database under the Databases icon.

Tool Reference – Server Resources: Databases: Connecting to Databases

5.   Connect to the newly created database by right-clicking the jdbc:derby://localhost:1527/StockMarket connection and selecting *Connect*.

Tool Reference – Server Resources: Databases: Executing SQL Queries

6.   Right-click the connection you just created and select *Execute Command*. This opens an SQL Command window in NetBeans.

7.   From with-in NetBeans open the StockMarket.sql file provided in the resources\brokertool directory.

8.   In the StockMarket.sql tab select jdbc:derby://localhost:1527/StockMarket as the connection.

9. Click on the *Run SQL* icon to execute the SQL statements.

Tool Reference – Server Resources: Databases: Interacting with Databases

10. View the data stored in the CUSTOMER, STOCK, and SHARES tables.

Task 2 – Creating a Persistence Unit

A persistence unit specifies which entity classes are grouped together in a persistence context and which data source is used for storage.

To create a persistence unit, complete the following steps:

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

2. In the *New File* dialog select the *Persistence* category and *Persistence Unit* as the file type.

3. Press *Next*.

4. Enter the following information in the *Provider and Database* dialog:

    •   Persistence Unit Name: **BrokerToolPU**

    •   Persistence Provider: **EclipseLink(JPA.2.0)(Default)**

    •   Data Source: **New Data Source...**

        •   JNDI Name: **StockMarket**

        •   Database Connection:
            jdbc:derby://localhost:1527/StockMarket [public on PUBLIC]

    •   Use Java Transaction APIs **(checked)**

    •   Table Generation Strategy: **None**

5. Click *Finish*.

Task 3 – Converting 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:

1. Import the javax.persistence package.

2. Add an @Entity annotation to the Customer class.

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

4. 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.

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

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

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

Task 4 – Converting `CustomerShare` and `Stock` to Use the Java Persistence API

Complete the following steps in the **BrokerTool** project for the `CustomerShare` and `Stock` classes:

1. Import the `javax.persistence` package.

1. Add no-arg constructors for each class.

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

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

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

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

Task 5 – Modifying the `BrokerModelImpl.java` Class to Use the Java Persistence API

In the **BrokerTool** project, edit `BrokerModelImpl.java` to use the Java Persistence API. Modify all methods to use the newly modified `Customer`, `CustomerShare`, and `Stock` entity classes. Follow these steps to make the changes:

1. Import the `javax.persistence` package.

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

   `@PersistenceContext private EntityManager em;`

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

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

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

   c. Remove all code in the constructor.

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

4. 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);
    }
}
```

Task 6 – Configuring, Deploying, and Testing the Application

To test the **BrokerTool** application, complete the following steps:

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

2. Test your application by pointing a browser at:

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

Exercise 2: Describing the Java Persistence API

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

Preparation

No preparation is needed for this exercise.

Task

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 _____.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Developing Applications for the Java™ EE Platform

Exercise Solutions

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

Solutions for Exercise 1

You can find example solutions for the exercises in this lab in the following directory: solutions\Persistence\.

Solution for Exercise 2: Describing Java Persistence API

Compare your responses to 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*.

Exercise Solutions

Lab 10

Implementing a Transaction Policy

Objective

Upon completion of this lab, you should be able to:

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

Exercise 1: Determining When Rollbacks Occur

In this exercise, you are presented with scenarios of interactions between the various parts of a theoretical bank application.

The purpose of this exercise is for you to become familiar with the effect of transaction attributes on the handling of transaction rollback when failures occur in various parts of an application.

Preparation

No preparation is needed for this exercise.

Task

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:

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

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

3. 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()
```

1. 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?

2. 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?

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

Exercise 2: Using the Versioning Features of the Persistence API to Control Optimistic Locking

This exercise contains the following sections:

- "Task 1 – Demonstrating Lost Updates in the BrokerTool Application"

- "Task 2 – Modifying the StockMarket Database to Support Versioning"

- "Task 3 – Updating the BrokerLibrary Entity Classes to Support Versioning"

- "Task 4 – Adding a Hidden Version Form Input Field to the CustomerDetails Servlet"

- "Task 5 – Modifying CustomerController to use the Version Value"

- "Task 6 – Modifying BrokerModelImpl to Use Versioning"

- "Task 7 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Demonstrating Lost Updates in the BrokerTool Application

Complete the following steps:

1. Deploy the **BrokerTool** web application if it is not deployed currently.

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

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

4. In both Browser A and B, retrieve the same customer's details.

5. In Browser A, change the customer's name or address and press the update button.

6. 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 press the update button. The changes made in Browser B overwrite those made in Browser A. This is called a lost update.

Task 2 – Modifying the StockMarket Database to Support Versioning

Complete the following steps:

1. Start the Java DB database server if it is not already started.

2. Connect to the StockMarket database using the `jdbc:derby://localhost:1527/StockMarket` connection.

3. Recreate the database tables and populate them by executing the `VersioningStockMarket.sql` provided in the `resources\brokertool` directory.

4. View the data stored in the CUSTOMER, STOCK, and SHARES tables.

Task 3 – Updating the `BrokerLibrary` Entity Classes to Support Versioning

Complete the following steps:

1. Modify `Customer.java`, `CustomerShare.java`, and `Stock.java` to support versioning by add the following code to each class:

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

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

2. Add any required import statements.

3. 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;
}
```

Task 4 – Adding 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. Complete the following steps:

1. 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.

2. 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 + "'/>");
```

Task 5 – Modifying `CustomerController` to use the Version Value

Complete the following steps:

1. 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"));
}
```

2. 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 3.

Task 6 – Modifying `BrokerModelImpl` to Use Versioning

Complete the following step:

1. 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. 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");
    }
```

2. Add any required import statements.

Task 7 – Configuring, Deploying, and Testing the Application

To test the Customer entity class, complete the following steps:

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

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

3. In both Browser A and B, launch the **BrokerTool** application using the URL http://localhost:8080/BrokerTool/.

4. In both Browser A and B, retrieve the same customers details.

5. In Browser A, change the customer's name or address and press the update button.

6. 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 press the update button. The changes made in Browser B are no longer be accepted because the customer's data has been modified by another client.

Note – 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 using a HttpSession.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Exercise Solutions

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

Solution for Exercise 1: Determining When Rollbacks Occur

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()
```

1. 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 2 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 catches 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 entity bean code from a transaction.

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

    None.

    If there were 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.

3. 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 from 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()
```

1. 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.

2. 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.

3. 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 Exercise 2: Use the Versioning Features of the Persistence API to Control Optimistic Locking

You can find example solutions for exercise 2 in this lab in the following directory: solutions\Transactions\.

Lab 11

Developing Java EE Applications Using Messaging

There are no labs associated with this module.

Developing Applications for the Java™ EE Platform

Lab 12

Developing Message-Driven Beans

Objectives

Upon completion of this lab, you should be able to:

- Implement a message-driven bean
- Describe message-driven beans

Introduction

In this lab, 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.

Developing Applications for the Java™ EE Platform

Exercise 1: Implementing the Message-Driven Bean

This exercise contains the following sections:

- "Task 1 – Creating the Managed Resources"
- "Task 2 – Copying the `StockMessageProducerBean` EJB"
- "Task 3 – Creating the Message-Driven Bean"
- "Task 4 – Configuring, Deploying, and Testing the Application"

Preparation

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

Task 1 – Creating 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.

Tool Reference – Java EE Development: Configuring Java EE Resources: Configuring JMS Resources: Creating a JMS Resource

These steps create a JMS queue with a JNDI name of `jms/UpdateStock`.

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

2. In the *New File* dialog select the *GlassFish* category and *JMS Resource* as the file type.

3. Press *Next*.

4. Enter the following information in the *General Attributes - JMS Resource* dialog.

   - JNDI Name: **jms/UpdateStock**
   - Admin Object Resource: **javax.jms.Queue**

5. Click *Next*.

6. In the *JMS Properties* dialog 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*.

These steps create a JMS connection factory with a JNDI name of `jms/UpdateStockFactory`.

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

9. In the *New File* dialog select the *GlassFish* category and *JMS Resource* as the file type.

10. Press *Next*.

11. Enter the following information in the *General Attributes - JMS Resource* dialog.

    •   JNDI Name: **jms/UpdateStockFactory**

    •   Connector Resource: **javax.jms.QueueConnectionFactory**

12. Click *Finish*.

Task 2 – Copying the `StockMessageProducerBean` EJB

The JMS message producer is provided to you in the form of a scheduled no-interface local session EJB.

1. In the *Favorites* window copy the `StockMessageProducerBean.java` file from *resources -> brokertool* to the clipboard.

2. Paste the `StockMessageProducerBean.java` file into the `trader` package of the **BrokerTool** project.

3. View the source code for the `StockMessageProducerBean` EJB. The EJB tries to update the *ORCL* stock price once per minute.

Task 3 – Creating the Message-Driven Bean

Complete the following steps:

Create a new message-driven bean in the **BrokerTool** project.

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

2. In the *New File* dialog select the *Java EE* category and *Message-Driven Bean* as the file type.

3. Press *Next*.

4. Enter the following information in the *Name and Location* dialog.

- • EJB Name: **UpdateStockBean**

- • Project: **BrokerTool**

- • Location: **Source Packages**

- • Package: **trader**

- • Project Destinations: **jms/UpdateStock**

5. Click *Finish*.

6. Add an annotated EJB reference variable for the BrokerModelImpl session bean to the UpdateStockBean.

   @EJB private BrokerModel model;

7. The onMessage method should receive a javax.jms.TextMessage object containing a message in the format of *ORCL,200.75*. Parse this message using the String class method split(",") and the Double.parseDouble("") method. Use the model reference obtained in Step 6 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.

Task 4 – Configuring, Deploying, and Testing the Application

Complete the following steps:

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

Tool Reference – Server Resources: Java EE Application Servers: Examining Application Servers

2. After deployment, log into the application server administrative interface. Click on *Resources* -> *JMS Reources*. 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.

3. View the stock prices by pointing a browser at:

   http://localhost:8080/BrokerTool/Stocks.xhtml

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

Exercise 2: Describing Message-Driven Beans

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

Preparation

No preparation is needed for this exercise.

Task

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

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

   a. Synchronous message consumers

   b. Asynchronous message consumers

2. The _____ method in a JMS MDB is called by the server when a message arrives.

3. To send a message to a queue a JMS client would need to obtain a _____ and a _____ using either JNDI or dependence injection.

4. True or False: A message-driven bean must have an `onMessage` method.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

• Experiences

• Interpretations

• Conclusions

• Applications

Exercise Solutions

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

Solutions for Exercise 1

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

Solution for Exercise 2: Describing Message-Driven Beans

Compare your responses to the following answers:

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

    •   *Asynchronous message consumers*

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

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

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

Lab 13

Web Service Model

There are no labs associated with this module.

Lab 14

Implementing Java EE Web Services with JAX-RS & JAX-WS

Objectives

Upon completion of this lab, you should be able to:

- Create a JAX-WS POJO end point web service
- Create a web service client from a WSDL file
- Create a JAX-RS POJO end point web service
- Describe Java web services

Lab 14-1

Introduction

In these labs, 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.

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

Exercise 1: Creating a JAX-WS Web Service

This exercise contains the following sections:

- "Task 1 – Creating the StockPrice Web Service"

- "Task 2 – Compiling and Deploying the BrokerTool Application"

- "Task 3 – Testing the StockPrice Web Service"

In this exercise, 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.

Preparation

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

Task 1 – Creating the StockPrice Web Service

In the **BrokerTool** project, create a web service by completing the following steps:

Tool Reference – Java EE Development: Web Services: JAX-WS Web Services: Creating an Empty JAX-WS Web Service

1. Right-click the **BrokerTool** project

2. Select *New* then *Web Service*.

3. In the *Name and Location* dialog, enter the following information:

    • Web Service Name: **StockPrice**

    • Location: **Source Packages**

    • Package: **trader.web**

    • Create Web Service from Scratch **(Selected)**

4. Click *Finish*.

5. Add the getStockPrice method to the StockPrice class:

    a. Add a method with the following signature:

    public String getStockPrice(String symbol) { .. }

    b. Add the @WebMethod annotation to the getStockPrice(...) method.

    c. Add an annotated EJB reference variable for the BrokerModelImpl session bean to the StockPrice class.

    @EJB private BrokerModel model;

    d. Add any needed import statements.

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

    f. 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.

Task 2 – Compiling and Deploying the BrokerTool Application

Deploying the `StockPrice` web service generates several supporting classes and a WSDL on the application server

Complete the following steps:

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

Task 3 – Testing the `StockPrice` Web Service

Complete the following steps:

1. View the XML output generated in WSDL by pointing a web browser at `http://localhost:8080/BrokerTool/StockPriceService?WSDL`.

2. Test the `StockPrice` web service by pointing a web browser at `http://localhost:8080/BrokerTool/StockPriceService?Tester`

Exercise 2: Creating a Web Service Client

This exercise contains the following sections:

- "Task 1 – Creating the Web Service `Port` or `Proxy` Classes"

- "Task 2 – Coding the Web Service Client Application"

- "Task 3 – Compiling and Executing the WebServiceTester Application"

In this exercise 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.

Preparation

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

Task 1 – Creating the Web Service `Port` or `Proxy` Classes

Create a new Java Application Project named the **WebServiceTester** project by completing the following steps:

Tool Reference – Java Development: Java Application Projects: Creating Projects

1.   Select *File* then *New Project*.

2.   Choose *Java* then *Java Application*.

3.   Click the *Next* button.

4.   In the *Name and Location* dialog enter the following information.

  •   Project Name: **WebServiceTester**

  •   Project Location: **D:\Labs\student\projects**

  •   Use Dedicated Folder for Storing Libraries: (unchecked)

  •   Create Main Class: **webservicetester.Main**

  •   Set as Main Project: (unchecked)

5.   Click *Finish*.

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:

Tool Reference – Java EE Development: Web Services: Creating Web Services Clients

6.   Right-click the **WebServiceTester** project.

7.   Select *New* then *Web Service Client*.

8.   Enter the following information for the web service client in the *WSDL and Client Location* dialog:

  •   WSDL URL:
    http://localhost:8080/BrokerTool/StockPriceService?WSDL

  •   Package: **webservicetester**

  •   Client Style: **JAX-WS Style**

9.   Click *Finish*.

Note – 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.

Task 2 – Coding the Web Service Client Application

Complete the following step:

1. 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"));
```

Task 3 – Compiling and Executing the WebServiceTester Application

Complete the following step:

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

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

Exercise 3: Creating a JAX-RS Web Service

This exercise contains the following sections:

- "Task 1 – Creating the `StockResource` Web Service"

- "Task 2 – Compiling and Deploying the BrokerTool Application"

- "Task 3 – Testing the `StockResource` Web Service"

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

Preparation

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

Task 1 – Creating the StockResource Web Service

In the **BrokerTool** project, create a web service by completing the following steps:

Tool Reference – Java EE Development: Web Services: JAX-RS Web Services: Creating a RESTful Web Service

1. Right-click the **BrokerTool** project

2. Select *New* then *RESTful Web Services from Patterns*.

3. In the *Select Pattern* dialog, enter the following information:

    • Design Pattern: **Simple Root Resource**

4. Click *Next*.

5. In the *Specify Resource Class* dialog, enter the following information:

    • Location: **Source Packages**

    • Resource Package: **trader.web**

    • Path: **stocks/{symbol}**

    • Class Name: **StockResource**

    • MIME Type: **text/plain**

    • Representation Class: **java.lang.String**

6. Click *Finish*.

7. If a *REST Resources Configuration* dialog appears, enter the following information:

    • Specify the way REST resources will be registered in the application: **Create default REST servlet adaptor in web.xml (selected)**

    • REST Resources Path: **\resources**

8. Click *ok*.

9. This web service only allows the reading of a stock price. Remove the putText method from the StockResource class.

10. Implement the getStockPrice method.

    a. Add or remove import statements as needed.

    b. 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.

c.   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");
```

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

e.   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.

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

Tool Reference – Java EE Development: Web Applications: Web Deployment
Descriptors: Opening the Standard Deployment Descriptor

Task 2 – Compiling and Deploying the BrokerTool Application

Complete the following steps:

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

Task 3 – Testing the `StockResource` Web Service

Complete the following steps:

1.   View the text output of the RESTful web service by pointing a web browser
at
`http://localhost:8080/BrokerTool/resources/stocks/ORCL`.

Note – 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.

Exercise 4: Describing Java Web Services

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

Preparation

No preparation is needed for this exercise.

Task

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 that is typically transferred through HTTP.

3. Only a _____ EJB can be a web service endpoint.

4. The only other Java web service endpoint besides an EJB is a _____ endpoint.

5. 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 _____.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Exercise Solutions

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

Solutions for Exercises 1, 2, and 3

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

Solution for Exercise 4: Describing Java Web Services

Compare your fill-in-the-blank responses to 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. Only a *Stateless 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. 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*.

Lab 15

Implementing a Security Policy

Objectives

Upon completion of this lab, you should be able to:

- Secure the `PortfolioController` servlet
- Use the EJB security API to get the user's identity in an EJB component
- Create roles and users
- Create a web tier security policy
- Create an EJB tier security policy
- Describe Java EE security

Introduction

At present, your application has no access control, so it is completely open to all users. In this lab, 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 standalone 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 than can be accomplished at the web tier.

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

This exercise contains the following sections and is an example of programmatic access control.

- "Task 1 – Securing the `getAllCustomerShares` Method"

- "Task 2 – Deploying and Testing the Session Bean"

In this exercise, 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.

Preparation

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

Task 1 – Securing the `getAllCustomerShares` Method

Add security features to the `getAllCustomerShares` method in the `BrokerModelImpl` class:

1. Add the following import statements:

   ```
   import java.security.Principal;

   import javax.annotation.Resource;

   import javax.ejb.SessionContext;
   ```

2. Declare a session context for the class:

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

3. 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. 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();
   ```

4. 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.*

Task 2 – Deploying and Testing the Session Bean

Complete the following steps:

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

2. Test the session bean by pointing your web browser at:

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

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

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

Exercise 2: Creating Roles, Users, Groups, and a Web Tier Security Policy

This exercise contains the following sections:

- "Task 1 – Creating Roles in the Application"

- "Task 2 – Creating Users and Groups in the Application Server"

- "Task 3 – Mapping Roles to Groups"

- "Task 4 – Creating a Security Constraint"

- "Task 5 – Deploying and Testing the Application"

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 exercise, 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.

Preparation

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

Task 1 – Creating Roles in the Application

Complete the following step:

Make changes to the `BrokerModelImpl` class to add roles to the application.

1. Add an import statement for the `DeclareRoles` annotation.

    `import javax.annotation.security.DeclareRoles;`

2. Add a class-level annotation in `BrokerModelImpl`. The annotation defines the two available user roles for this class:

    `@DeclareRoles({"admin","customer"})`

Task 2 – Creating Users and Groups in the Application Server

Tool Reference – Server Resources: Java EE Application Servers: Administering Security

In this task, you add two users to a security realm.

1. Login to the administration console:

    `http://localhost:4848`

2. Select *Configuration > Security > Realms > file*.

3. Click on the *Manage Users* button.

4. 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 Table 15-1 to configure these users.

Table 15-1 Users in the Security Realm

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

Task 3 – Mapping Roles to Groups

Complete the following steps to map roles to groups:

1. Edit the `sun-web.xml` deployment descriptor in the **BrokerTool** project.

Note – 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 > GlassFish Deployment Descriptor.*

2. 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>
```

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.

Task 4 – Creating 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:

1. Add the `@ServletSecurity` annotation to the `PortfolioController`. class.

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

These annotations restrict access to the `/PortfolioController` URL to users in the `admin` or `customer` roles.

2. Edit the `web.xml` deployment descriptor in the **BrokerTool** project.

Tool Reference – Java EE Development: Web Applications: Web Deployment Descriptors: Security Configuration

3. 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>
```

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, passwd files, etc.

Task 5 – Deploying and Testing the Application

To deploy and test the application, complete the following steps:

1. Save any modified files. If Deploy on Save is not enabled then deploy the **BrokerTool** web application manually. Resolve any errors before you continue.

2. Point your web browser at:

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

    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.

3. Type the user ID and password for user 123-45-6789.

    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.

4. 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 exercise.

Exercise 3: Creating an EJB Tier Security Policy

In Exercise 1, you restricted access to the `BrokerModelImpl.getAllCustomerShares` method programmatically, allowing only logged-in users to execute the method. In Exercise 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, then 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:

*   "Task 1 – Restricting `BrokerModelImpl` Methods"
*   "Task 2 – Customizing `BrokerModelImpl` Methods by Role"
*   "Task 3 – Deploying and Testing the Application"

Preparation

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

Task 1 – Restricting `BrokerModelImpl` Methods

Complete the following steps:

1. Verify the class-level annotation to `BrokerModelImpl` of:

    `@DeclareRoles({"admin","customer"})`

    This states that the admin and customer roles are used in this EJB.

2. Import the `@RolesAllowed` annotation.

    `import javax.annotation.security.RolesAllowed;`

3.   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.

Task 2 – Customizing `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.

Complete the following steps:

1.   Comment out the code at the beginning of the `getAllCustomerShares`
     method that deals with anonymous or guest users.

2.   Modify that `getAllCustomerShares` method so that a
     `BrokerException` is thrown if one of the conditions does not pass:

     a.   The caller is not in the `admin` role. Use the `ctx.isCallerInRole`
          method.

Note – If you do not have a context reference you can obtain one by adding
`@Resource private SessionContext ctx;` as an instance level variable.

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

Task 3 – Deploying and Testing the Application

Complete the following steps:

1. Save any modified files. If Deploy on Save is not enabled then deploy the **BrokerTool** web application manually. Resolve any errors before you continue.

2. Point your browser at
    `http://localhost:8080/BrokerTool/AllCustomers`

3. 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.

4. Close all instances of your web browser to log out.

5. Launch a new web browser and point it at
    `http://localhost:8080/BrokerTool/AllCustomers`

6. 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.

Exercise 4: Describing Java EE Security

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

Preparation

No preparation is needed for this exercise.

Task

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

1. To check the calling user, an EJB would use its _____.

2. The web tier equivalent of `isUserInRole(...)` is
    _____.

3. Two common security annotations used in an EJB are _____ and
    _____.

4. Web-tier components configure their security settings in the
    _____ file.

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

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences

- Interpretations

- Conclusions

- Applications

Exercise Solutions

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

Solutions for Exercises 1 Through 3

You can find example solutions for the exercises in this lab in the following directory: `solutions\Security`.

Solution for Exercise 4: Describing Java EE Security

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

1. To check the calling user, an EJB would use its *EJBContext or SessionContext*.

2. The web-tier equivalent of `isUserInRole(...)` is *isCallerInRole(...)*.

3. Two common security annotations used in an EJB are *@DeclareRoles* and *@RolesAllowed*.

4. Web-tier components configure their security settings in the *web.xml* file.

5. The *JavaEE6* version of enterprise Java first allowed the `@ServletSecurity` annotation to be used in a servlet.