ORACLE UNIVERSITY

Java EE 7: Back-end Server Application Development

Activity Guide
D85116GC20
Edition 2.0 | June 2016 | D95989

Learn more from Oracle University at **oracle.com/education/**

ORACLE

Authors

Anjana Shenoy
Eduardo Moranchel
Edgar Martinez
Tom McGinn
Michael Williams

Technical Contributors and Reviewers

Maria Asuzena Ortiz Castro
Juan Carlos Vargas Garcia
Cristian Joshiro Perez
Cristobal Ramirez
Angel Rodolfo Macias Buendia
Victor Daniel Jimenez Camacho
Alejandro Pena
Juan Francisco Garcia Navarro
Jose De Jesus Sanchez Rico
Helena Guadalupe Mares Tijerina
Silvia Elizabeth Alonso Sedano
Erick Jose Antonio Tovar Marmolejo

Editor

Vijayalakshmi Narasimhan
Raj Kumar
Chandrika Kennedy

Graphic Designers

Seema Bopaiah
Maheshwari Krishnamurthy

Publishers

Joseph Fernandez
Jayanthy Keshavamurthy
Jobi Varghese
Sumesh Koshy
Giri Venugopal

Table of Contents

Course Practice Environment: Security Credentials

Course Practice Environment: Security Credentials

For OS usernames and passwords, see the following:

- If you are attending a classroom-based or live virtual class (LVC), ask your instructor or LVC producer for OS credential information.
- If you are using a self-study format, refer to the communication that you received from Oracle University for this course.

For product-specific credentials used in this course, see the following table:

Product-Specific Credentials		
Product/Application	**Username**	**Password**
Administrator access to log in to Oracle WebLogic Server administration console	`weblogic`	`welcome1`
JavaDB	`oracle`	`oracle`
Security role for a user created in WebLogic Server	`admin`	`welcome1`
Default keyring password for NetBeans		`oracle`

Practices for Lesson 1: Java Platform, Enterprise Edition

Chapter 1

Practices for Lesson 1

There are no practices for this lesson.

Practices for Lesson 2: Enterprise Development Tools and Application Servers

Chapter 2

Practices for Lesson 2: Overview

Practices Overview

In these practices, you configure your environment to develop Java EE 7 applications.

Practice 2-1: Installing and Starting WebLogic Server

Overview

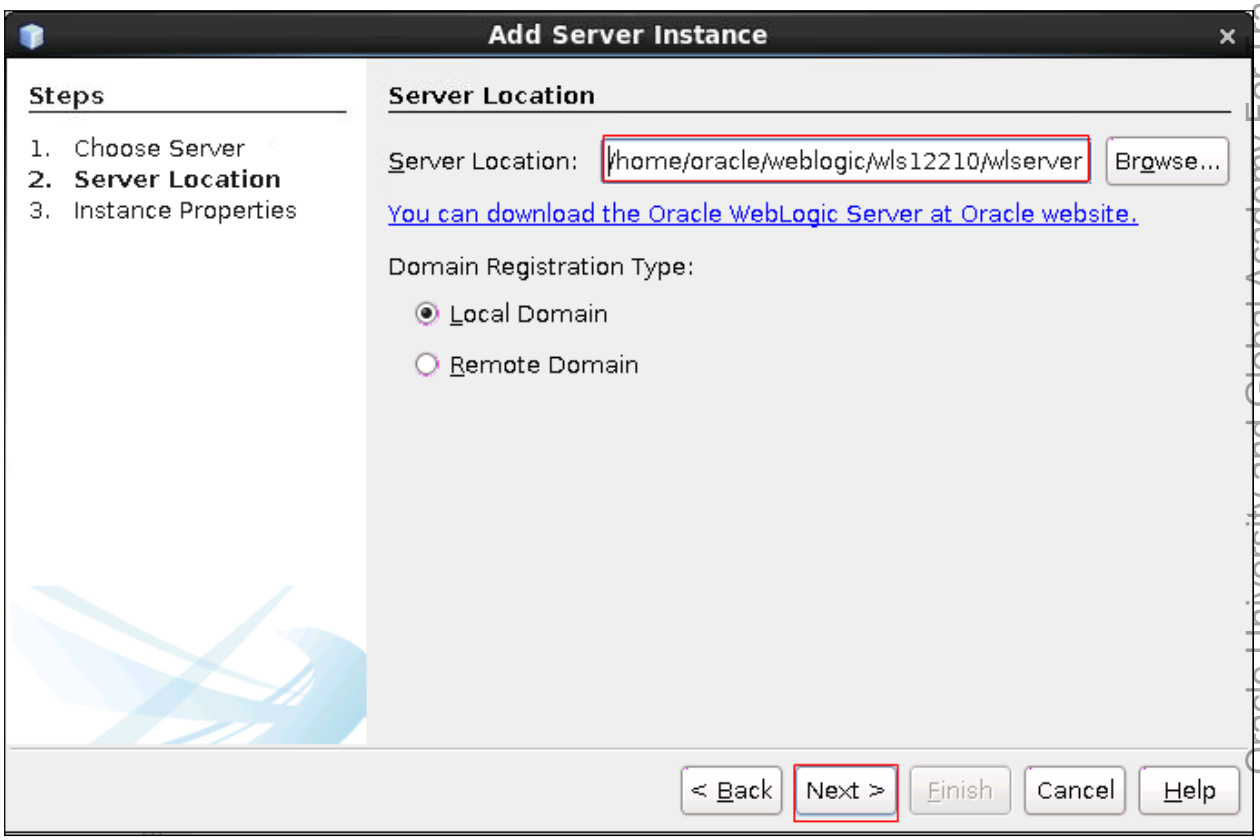
In this practice, you configure WebLogic Server as a Service in NetBeans and start the server.

Assumptions

- JDK 8 is installed.
- NetBeans 8.1 EE is installed.
- WebLogic Server is installed.

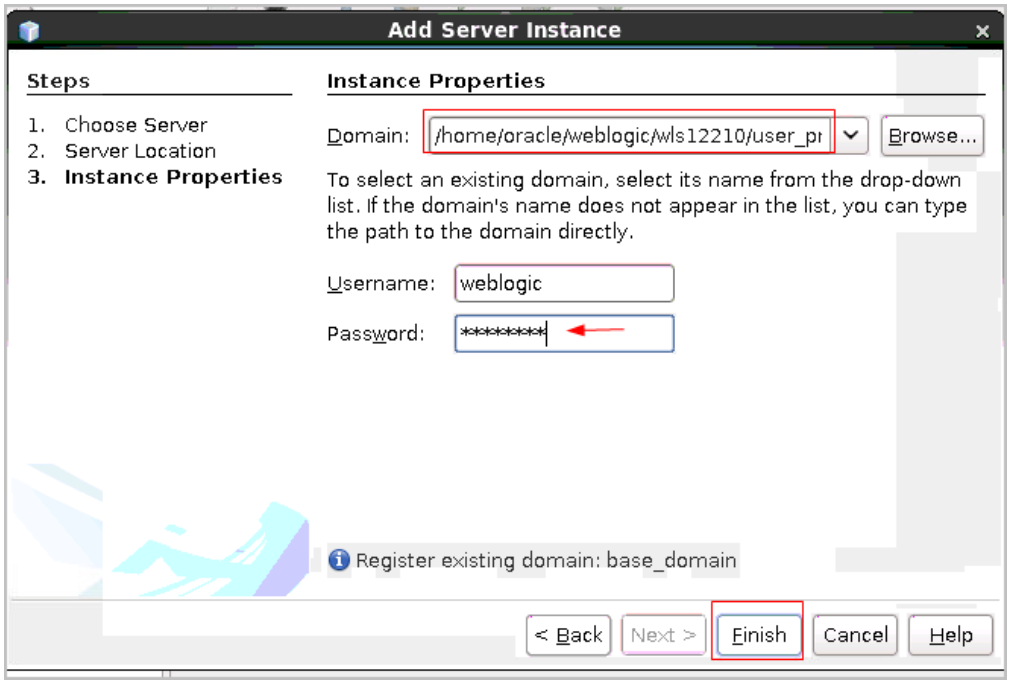
Tasks

1. Start NetBeans.
2. Click Window > Services to open the Services window in NetBeans.
3. Install WebLogic as a Server in NetBeans.
   a. Right-click Servers and select Add Server.
   b. Select Oracle WebLogic Server and click Next.
   c. Enter the Server Location as `/home/oracle/weblogic/wls12210/wlserver`.

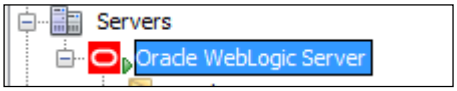


   d. Ensure Local Domain is selected as Domain Registration Type.
   e. Click Next.
   f. Enter the domain:
      `/home/oracle/weblogic/wls12210/user_projects/domains/base_domain`
   g. Enter the password as specified in the security credentials page.

h. Click Finish.



4. Start WebLogic Server.
   a. Expand the Servers folder.
   b. Right-click WebLogic Server and select Start.
   c. After a few seconds, you should see a green triangle beside the server name, indicating that the server started successfully.



Note: If you are prompted for a default keyring password, refer to the password in the security credentials page. Enter the password twice and click OK.

5. Open the Server Console.
   a. Expand WebLogic Server by clicking the triangle icon.
   b. Right-click WebLogic Server and select View Admin Console.

      The console opens in a new browser window. If the console does not open, open a browser and navigate to `http://localhost:7001/console`.
   c. Enter the username and password as specified in the security credentials page and click Login.

Practice 2-2: Writing a Simple Test Web Application

Overview

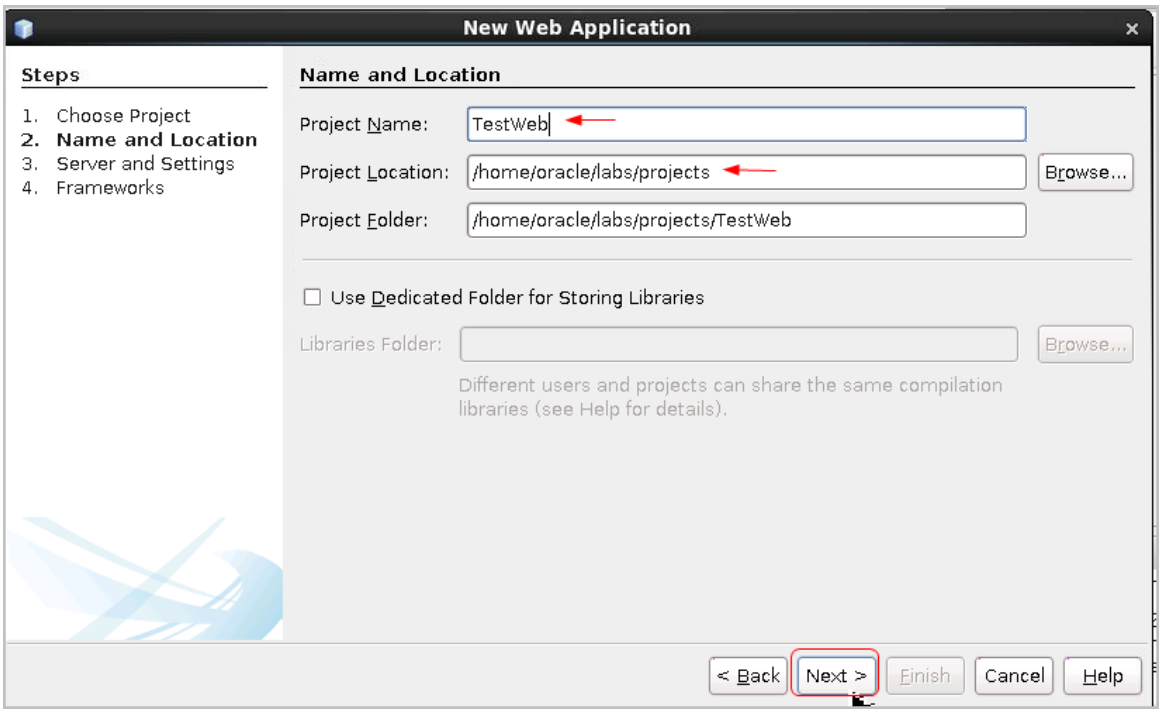
In this practice, you create a test project in NetBeans and deploy it to WebLogic Server.

Assumptions

NetBeans is running.

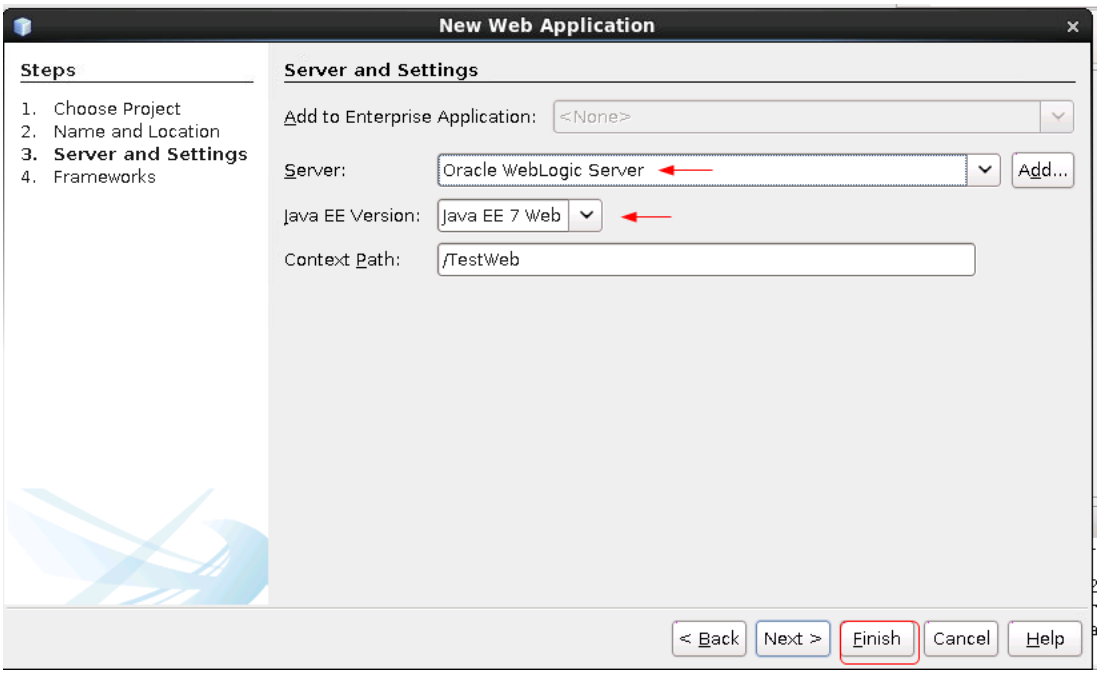
Tasks

1. Create a new web application project.
   a. Select File > New Project.
   b. Select Java Web from Categories and Web Application from Projects, and click Next.
   c. Enter `TestWeb` as the Project Name.
   d. Set the Project Location to `/home/oracle/labs/projects`.
   e. Click Next.



   f. Make sure that WebLogic Server is selected as the Server.
   g. Select Java EE 7 Web as the Java EE version.

h. Click Finish.



2. Modify `index.html`.

a. The `index.html` file opens in the editor in NetBeans.

b. Modify the text `TODO supply a title` to `Test Web App`.

```
<title>Test Web App</title>
```

Modify the text `TODO write content` to `Hello World`.

```
<div>Hello World</div>
```

c. Save the file.

3. Run the project in WebLogic.

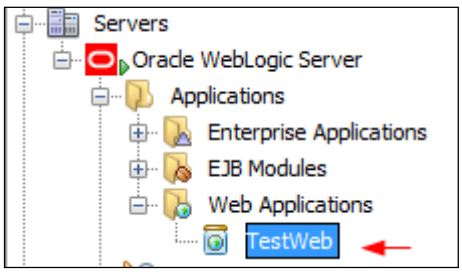
- Right-click the `TestWeb` project on the Projects tab and select Run.

  A browser window opens to the URL `http://localhost:7001/TestWeb/` and displays Hello World.

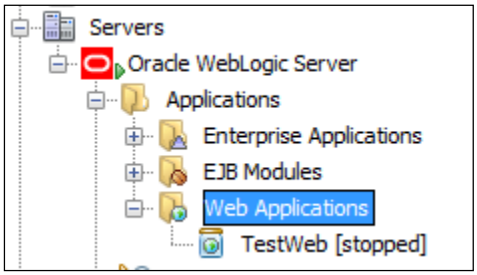
4. View and control the applications that are deployed in WebLogic Server.

a. On the Services tab, expand Servers > Oracle WebLogic Server > Applications > Web Applications.

You should see the `TestWeb` application:



b. Right-click `TestWeb` and select Stop.



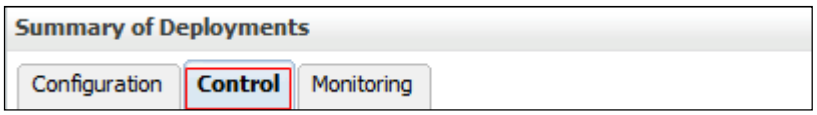
c. Try reloading the URL of the application in a browser (`localhost:7001/TestWeb/`). Because the server is no longer accepting connections, a 404 error is displayed.

5. Control the applications through the console.

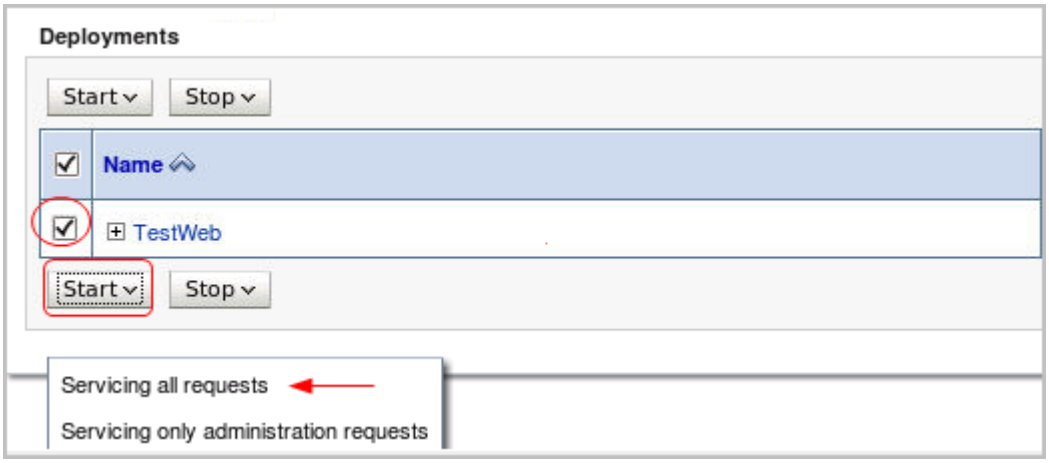
a. Open the WebLogic Server console.

b. Click Deployments.
You should see that `TestWeb` is deployed, but is not in Active state.

c. At the top of the page, click the Control tab.



d. Select `TestWeb`, click Start, and select Servicing all requests to restart the application.



e. You should see that `TestWeb` is in Active state.

f. Try reloading the URL of the application. This time, it should load properly.

Practices for Lesson 3: JavaBeans, Annotations, and Logging

Chapter 3

Practices for Lesson 3: Overview

Practices Overview

In this practice, you create an instance of a `java.util.Logger` class to log messages for your application.

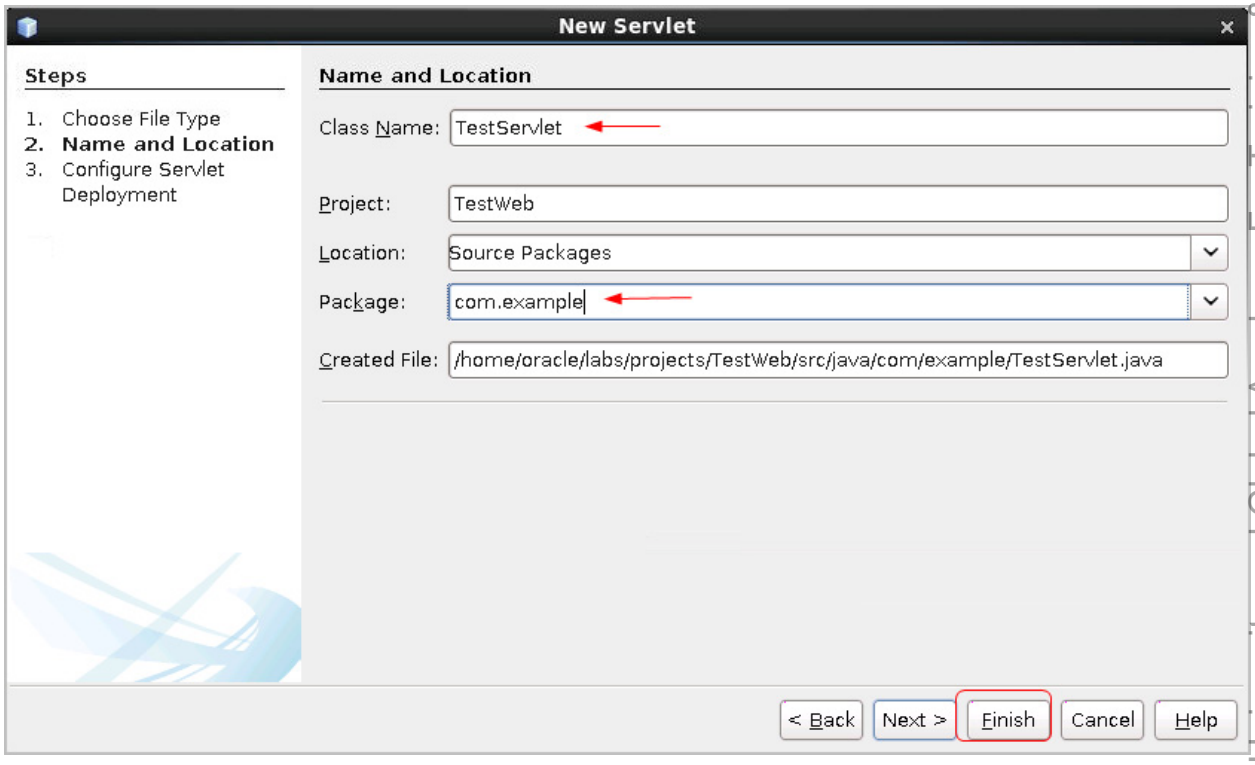
Practice 3-1: Adding a Logger to an Application

Overview

In this practice, you create an instance of a `Logger` class and use that class to log messages in the `TestWeb` application.

Tasks

1. Start NetBeans if it is not already running.
2. Open the `TestWeb` application from `/home/oracle/labs/projects` if it is not already open in NetBeans.
3. Add a servlet to the project.
   - Right-click the project and select New > Other.
   - Select Web from Categories and Servlet from File Types, and click Next.
   - Enter `TestServlet` as Class Name and `com.example` as Package, and click Finish.



4. Modify the servlet class to use a `Logger` class.
   - Add a `static final Logger` to the class and use the static `Logger.getLogger` method to create an instance called `logger`. Pass the string name of the fully qualified class name.

```
private static final Logger logger =
Logger.getLogger("com.example.TestServlet");
```

   - In the `processRequest` method, use the `logger` instance to log all of the request headers.

```
Enumeration<String> headers = request.getHeaderNames();
String header;
```

```
while(headers.hasMoreElements() != false) {
    header = headers.nextElement();
    String message = header + " : " + request.getHeader(header);
    logger.log(Level.INFO, message);
}
```

- Fix any missing import statements by right-clicking in the editor window and selecting "Fix Imports" or by pressing the Ctrl + Shift + I keys.
- Save the file.

5. Test the application.
   - Run the servlet by entering the following URL in a browser:

```
http://localhost:7001/TestWeb/TestServlet
```

   - In the WebLogic Server console window (on the Output tab), you should see the following header information printed (or something similar; note that you may not see all of the header information shown):

```
INFO: Host : localhost:7001
INFO: User-Agent : Mozilla/5.0(X11; Linux X86_64;rv:24.0)
Gecko/20100101 Firefox/24.0
INFO: Accept :
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
INFO: Accept-Language : en-US,en;q=0.5
Info:   accept-encoding : gzip, deflate
Info:   cookie : JSESSIONID=e01597191c47a3c16c8d9fd98a76; ....
Info:   connection : keep-alive
```

   Note: The default logging level for WebLogic Server is INFO.

   - More detailed information is also recorded in the `AdminServer.log` file, which can be accessed by using the following path:

```
/home/oracle/weblogic/wls12210/user_projects/domains/base_domain
/servers/AdminServer/logs/AdminServer.log
```

6. For all future projects, add logging statements as needed to aid in any troubleshooting.
7. Close the `TestWeb` project.

   In NetBeans, click the Projects tab, right-click the `TestWeb` project, and select Close.

Practices for Lesson 4: XML Programming with JAXB

Chapter 4

Practices for Lesson 4: Overview

Practices Overview

In these practices, you use JAXB to generate XML files from Java objects.

Practice 4-1: Marshalling a Java Class to an XML File

Overview

In this practice, you annotate your Java class with XML annotations and generate an XML file.

Assumptions

You have completed the lecture portion of the course.

Tasks

1. In NetBeans, open the `Prac04-01-XML` project for this lesson in the `/home/oracle/labs/projects/04_XML` directory.
2. Edit the `Product.java` file in the `com.example.xml` package:
   a. Add constructors to the file.
      - Add a zero argument constructor to the class. This is required by JAXB.
      - Add a constructor with the following signature to initialize an individual object:

```
public Product(String name, double price, String description)
{
    this.name = name;
    this.price = price;
    this.description = description;
}
```

Note: NetBeans can create the getters and setters for you. In the source file that you are updating, right-click the background and select **Refactor > Encapsulate Fields > Create Getters**. Fill the form to create the required methods.

   b. Add methods that return the details of the class.
      - Add a method that returns the `name`.
      - Add a method that returns the `price`.
      - Add a method that returns the `description`.
   c. Add XML annotations to the class.
      - Add an annotation to the `Product` class to make it a root element. Add this annotation just before the class declaration.

```
@XmlRootElement(name="product")
public class Product { ...
```

      - Set the accessor type for the class. This indicates that the fields should be included as elements in the XML document.

```
@XmlRootElement(name="product")
@XmlAccessorType(XmlAccessType.FIELD)
public class Product { ...
```

   d. Fix any missing import statements by right-clicking in the editor window and selecting “Fix Imports” or by pressing the Ctrl + Shift + I keys.

   e. Save the file.

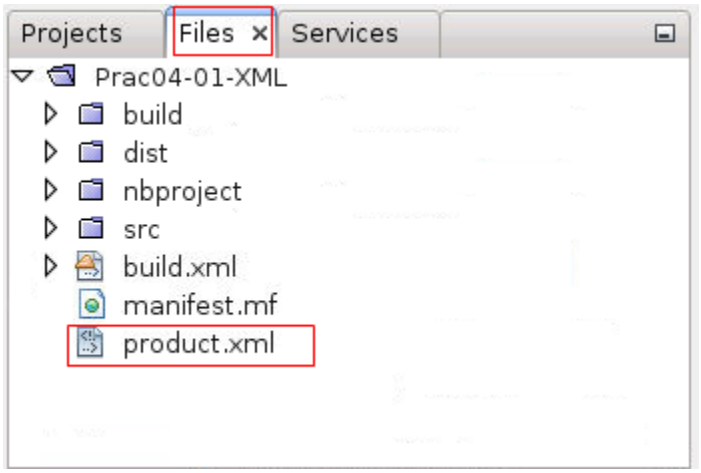
3. Edit the `WriteProductXML.java` file:
   a. In the `writeXML` method, create a new `Product` instance that you can marshal into an XML file.

```
Product p = new Product("Widget", 25.00, "The all new multifunction
widget");
```

   b. Call the `marshal` method to create an XML file and to also print the results to the console.

```
m.marshal(p, new File("product.xml"));
m.marshal(p, System.out);
```

   c. Save the file.
4. Run the project. Ensure that the XML output is written to the console. Also, open the `product.xml` file to examine the output in the file.
   Click the **Files** tab to see the generated file in the root directory for the project.



Practice 4-2: Marshalling a Java Collection to an XML File

Overview

In this practice, you convert an `ArrayList` of `Product` objects into an XML file.

Assumptions

You have completed the previous practice.

Tasks

1. In NetBeans, open the `Prac04-02-XML` project for this lesson in the `/home/oracle/labs/projects/04_XML` directory.
2. Edit the `ProductList.java` file in the `com.example.xml` package:
   a. Add constructors to the file.
      - Add a zero argument constructor to the class. This is required by JAXB.
      - Add a constructor with the following signature to initialize the `List` object:

```
public ProductList(List<Product> pl) {

  this.pList = pl;

}
```

   b. Add a `getList` method that returns the list.

```
  public List<Product> getList(){
    return pList;
  }
```

   c. Add XML annotations to the class:
      - Add an annotation to make the `productList` class a root element.

```
@XmlRootElement(name="productList")
```

      - Set the accessor type for the class. This indicates that the fields should be included as elements in the XML document.

```
@XmlAccessorType(XmlAccessType.FIELD)
```

   d. Add an XML annotation to the `pList` variable, which holds the product `ArrayList`. This annotation sets the element name for each product object.

```
public class ProductList {

  @XmlElement(name="product")
  private List<Product> pList = new ArrayList<>();
...
}
```

   e. Save the file.

3. Edit the `WriteProductList.java` file:

   a. In the `writeXML` method, create three new `Product` instances that you can marshal into an XML file.

```
    newList.add(new Product("Widget", 25.00, "The all new
multifunction widget"));
    newList.add(new Product("Widget Pro", 35.00, "The all new
multifunction widget pro"));
    newList.add(new Product("Widget Pro XL", 45.00, "The all new
multifunction widget pro xl"));
```

   b. Assign the newly created list to the `ProductList` class.

```
ProductList pl = new ProductList(newList);
```

   c. Set a JAXB property to format the XML output. This property formats the XML output such that each element is indented under its parent.

```
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

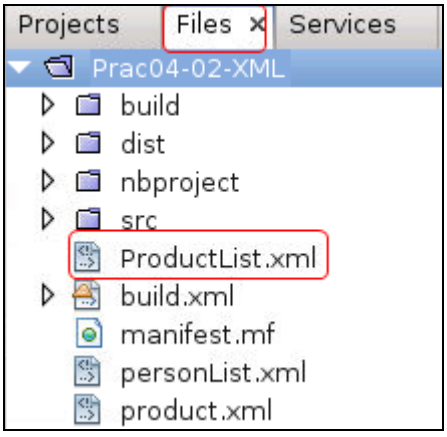
   The fields that can be set are described in the `Marshaller` API documentation.

   d. Call the `marshal` method to create an XML file and to also print the results to the console.

```
m.marshal(pl, new File("ProductList.xml"));
m.marshal(pl, System.out);
```

4. Run the project. Ensure that the XML output is written to the console. Also, open the `ProductList.xml` file to examine the output in the file.

   Click the **Files** tab to see the generated file in the root directory for the project.



Practice 4-3: Generating a Schema from Java Source Files

Overview

In this practice, you generate an XML schema from the Java classes that you created.

Assumptions

You have completed the previous practice.

Tasks

1. Continue using the same project as the previous practice.
2. Open a terminal window.
3. Use the `cd` command to change to the directory that contains the source files.

```
cd /home/oracle/labs/projects/04_XML/Prac04-02-
XML/src/com/example/xml
```

4. Run the `schemagen` program to generate an XML schema XSD file based on the two product classes.

```
schemagen Product.java ProductList.java
```

   The result of the command is a schema file with XML schema definitions for both classes. The resulting schema file could be used to validate either `product` or `productList` elements.

5. Open the `schema1.xsd` file from NetBeans Projects tab to view the results of the command.

   Note: The tool automatically names the file with a number. You cannot specify the name of the schema file.

Practices for Lesson 5: SOAP Web Services with JAX-WS

Chapter 5

Practices for Lesson 5: Overview

Practices Overview

In these practices, you create and test SOAP web services. You create both servers and clients.

Practice 5-1A: Creating and Testing a SOAP Web Service and Client

Overview

In this practice, you create a `Calc` web service that calculates the square of a number. You test the web service by using NetBeans. Then, you create a Java SE web service client to connect to your new web service.

Assumptions

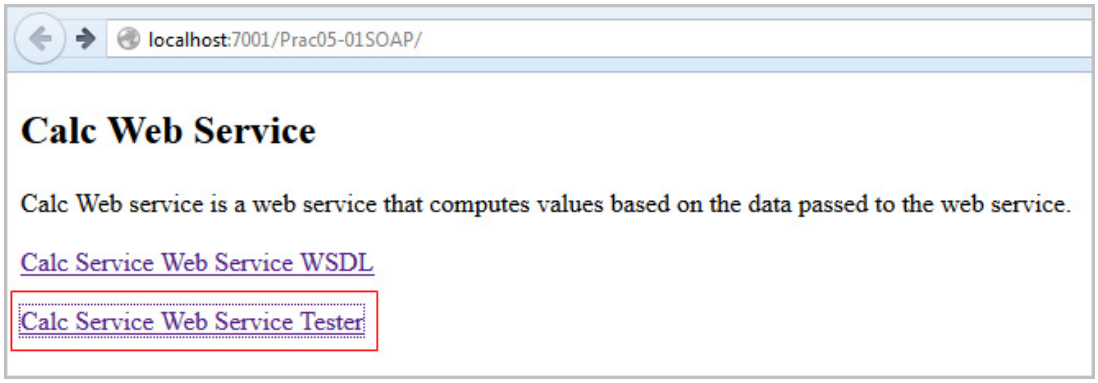
You have completed the lecture portion of the course.

Tasks

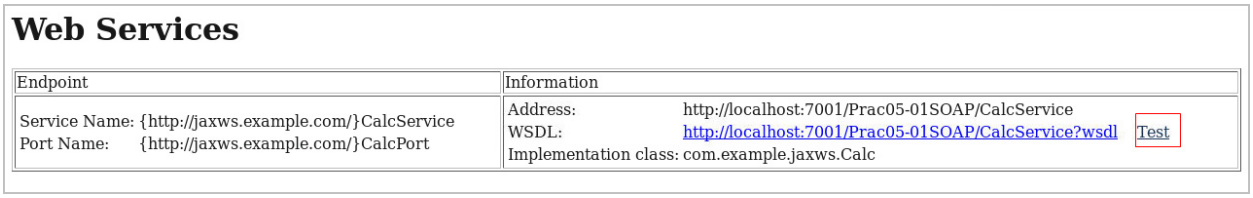
1. In NetBeans, open the `Prac05-01SOAP` project for this lesson in the `/home/oracle/labs/projects/05_SOAP` directory. Note that this project is a Java EE web project.
2. Expand the `com.example.jaxws` package.
3. Open the `Calc` class.
4. Complete the source code for the `Calc` class.
   a. Add the code to calculate the result in the `squared` method.

```
double num = Double.valueOf(number);
return Double.toString(num * num);
```

   b. Annotate the `squared` method as a `@WebMethod`.
5. Complete the `index.html` file for the project.
   - Add the URL for the WSDL file to the `<a href="">` link for the Web Service WSDL text: `http://localhost:7001/Prac05-01SOAP/CalcService?wsdl`.
   - Add the URL for the Web Service Tester to the `<a href="">` link for the Web Service Tester text: `http://localhost:7001/Prac05-01SOAP/CalcService?Tester`.
6. Run the application.
   - Test the WSDL by clicking the link and viewing the WSDL file.
7. Test the web service by using the `Tester` interface.
   - Click the link for the `Tester` interface.

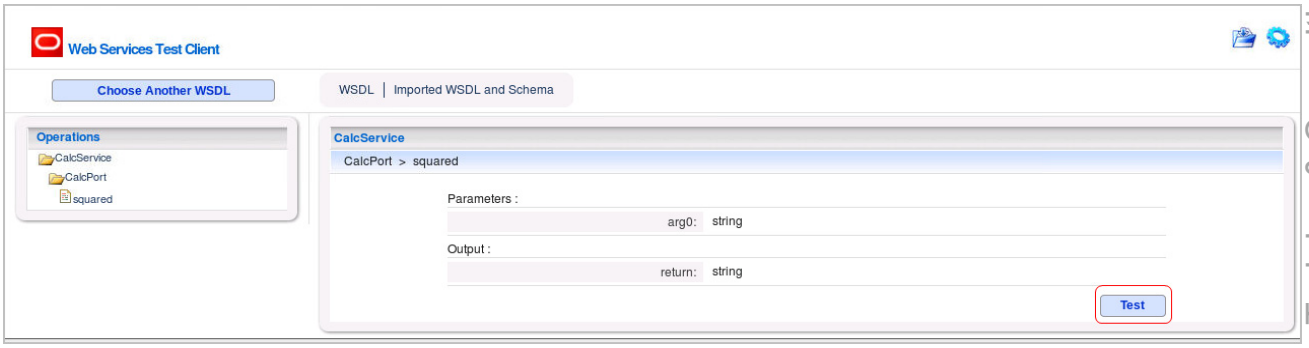


   The Web Services page is displayed.

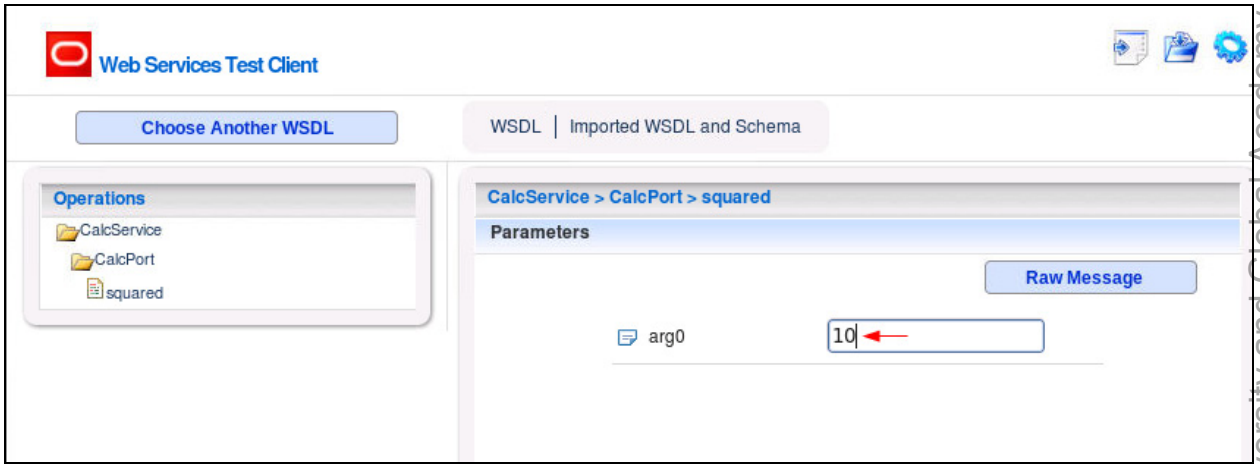


- Click Test. The Web Services Test Client page is displayed.

- Click Test.

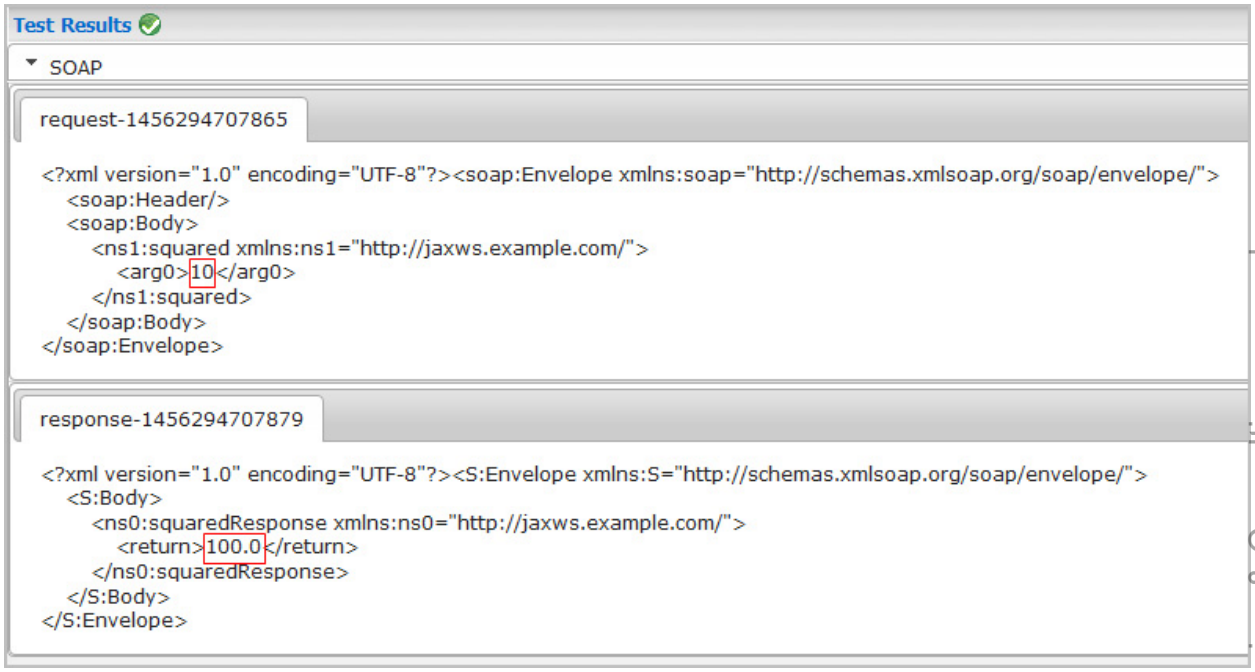


- Enter a value to test.



- Click `Invoke` at the bottom of the page to submit the value to the web service.

- Ensure that the result is the square of the initial value. The test results are displayed at the bottom of the page.
- This example shows the result if `10` is entered.



Practice 5-1B: Creating a Web Service Client

Overview

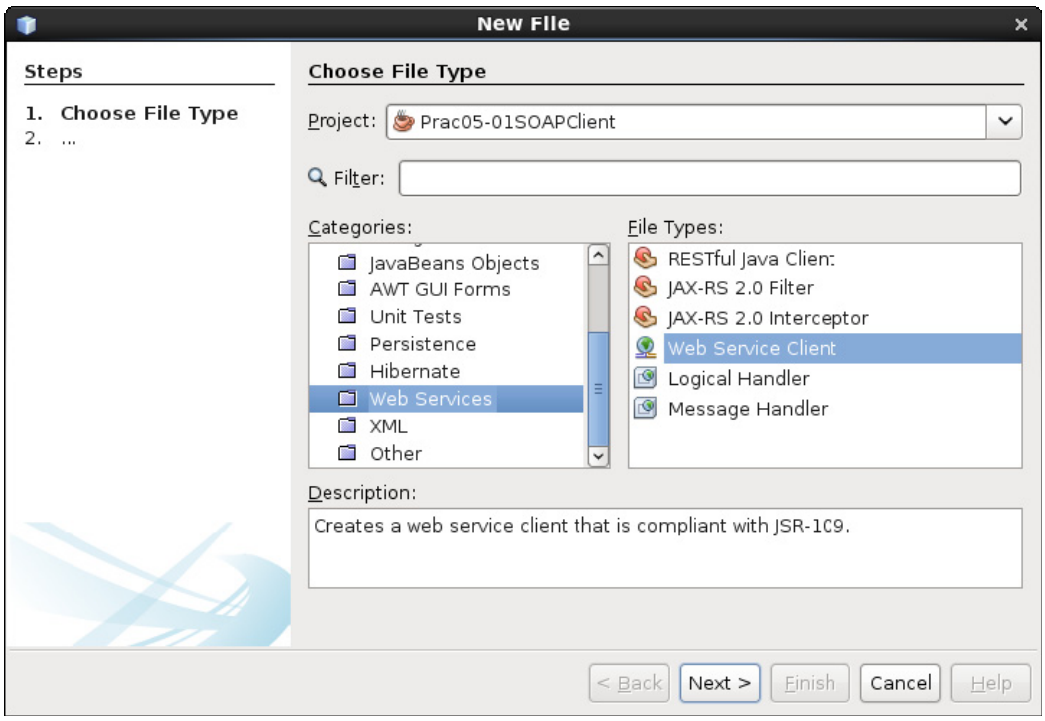
In this practice, you create a Java SE web service client from a WSDL file.

Assumptions

You have completed the preceding practice.

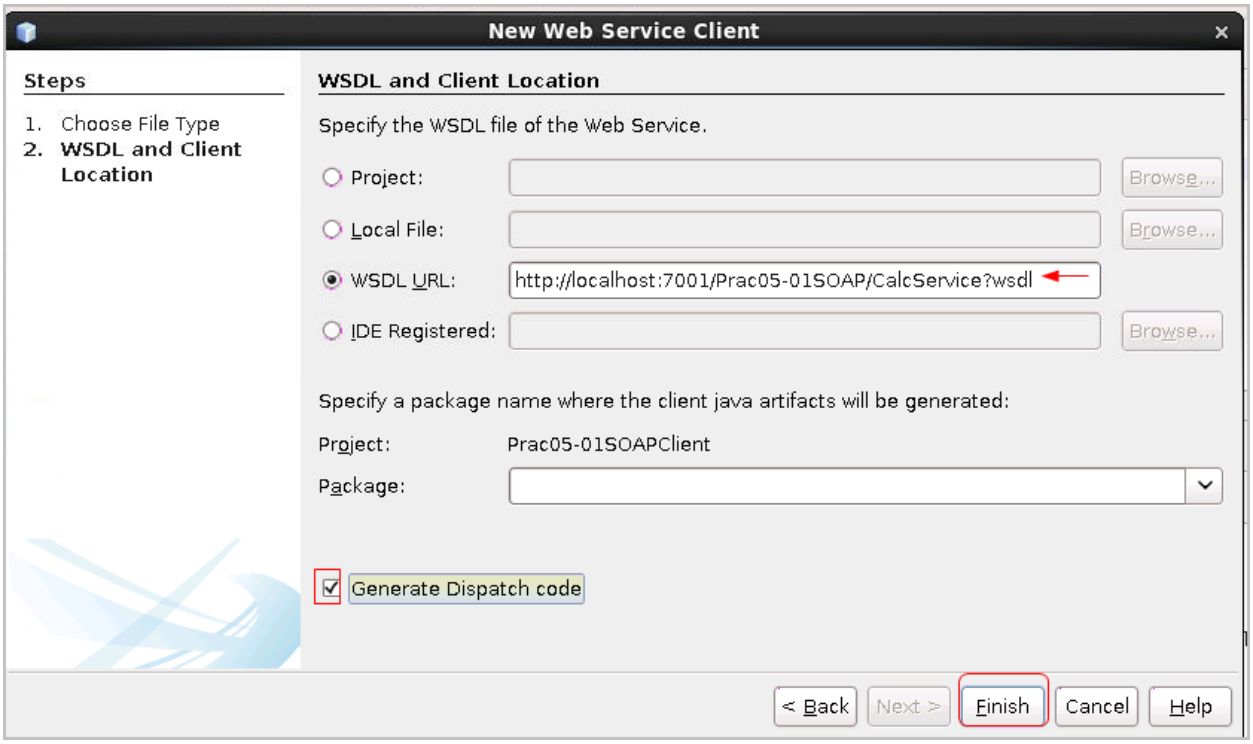
Tasks

1. Leave WebLogic running with the `Prac05-01SOAP` project deployed. You reference the WSDL file URL when you create the web service client.
2. Create a new Java SE project. Name the project `Prac05-01SOAPClient` and specify the project location as `/home/oracle/labs/projects/05_SOAP`.
   - Enter the `Main` class as `com.example.jaxws.Main`.
   - Let NetBeans create the project.
3. Open the `com.example.jaxws` package.
4. Right-click the `com.example.jaxws` package and, from the context menu, select **New** > **Other** > **Web Services** > **Web Service Client**.

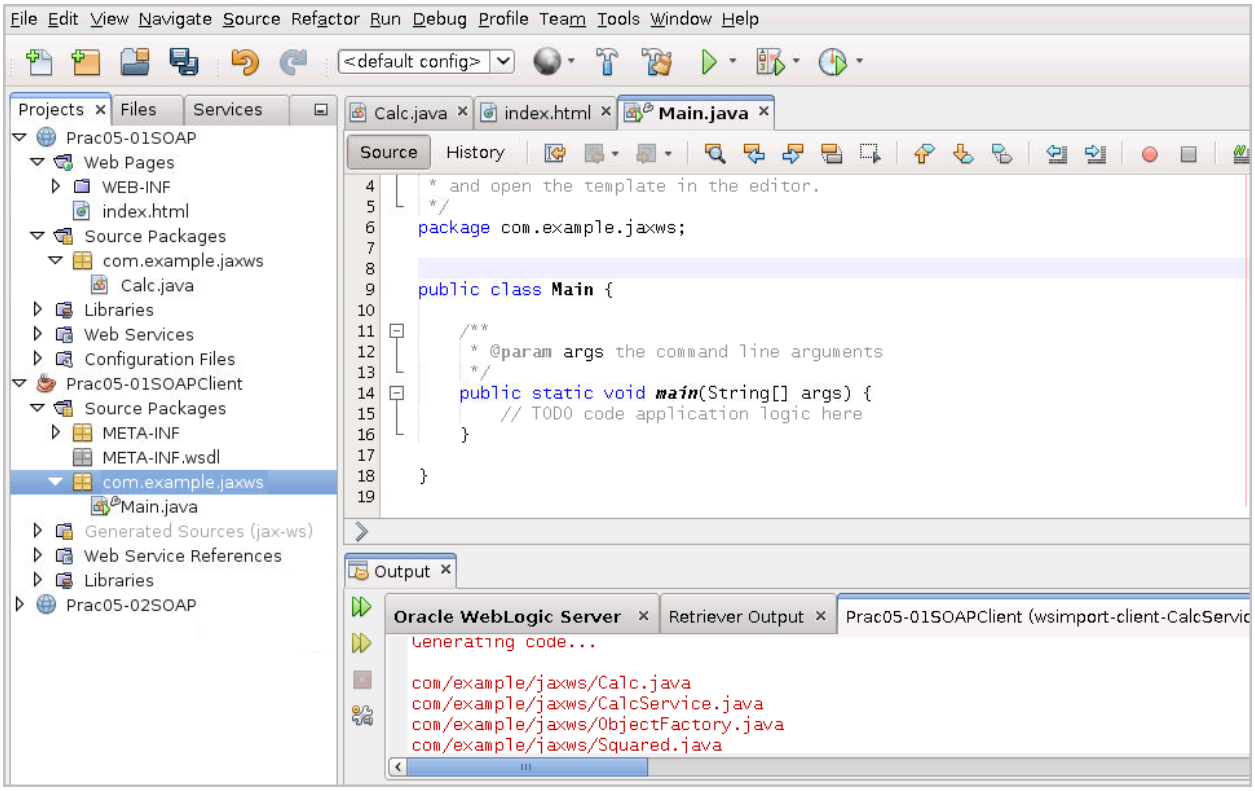


5. Click **Next**.

6. Copy the WSDL URL from the project index page in Firefox. Paste that URL as shown:
   http://localhost:7001/Prac05-01SOAP/CalcService?wsdl
7. Select Generate Dispatch code.
8. Click **Finish**.



9. You notice that NetBeans generates a number of source files for you. A Generated Sources section is added to the project. In addition, configuration files and sections are added as shown.



You can now edit the `Main` file and rely on the generated source code to make a web service call.

10. Enter the following code for the `Main` method:

```
    CalcService serv = new CalcService();
    Calc servPort = serv.getCalcPort();

    System.out.println("The square of 9 is: " + servPort.squared("9"));
    System.out.println("The square of 10 is: " + servPort.squared("10"));
```

11. Run your new web service client. The result should now be shown in the console output.

```
run:
The square of 9 is: 81.0
The square of 10 is: 100.0
```

Note: Because some source files are auto-generated from a URL, NetBeans may report some errors if you perform a **Clean Project** because the auto-generated source files no longer exist. In addition, when this method is used, the SOAP application must be deployed on a running WebLogic Server for the source files to be auto-generated.

Practice 5-2A: Annotating the SOAP Web Service Methods

Overview

In this practice, you add two additional methods to your web service, along with some annotation options.

Assumptions

You have completed practice 5-1A.

Tasks

1. You can use the `Prac05-02SOAP` project for this lesson. Or you could continue with the previous project if you wish.
2. Modify the two methods in the `Calc` class with the following signatures:

```
public String addTwo(String aStr, String bStr)
public String subtractTwo(String aStr, String bStr)
```

   - Create the following code that is necessary to return the desired results for the two methods, respectively. You will need to convert the input strings into double values and back to strings.
     - `return aStr + bStr;`
     - `return aStr - bStr;`
3. Create two annotations for the two methods.
   - For `addTwo: @WebMethod(operationName="add")`
   - For `subtractTwo: @WebMethod(operationName="subtract")`
4. Deploy and test the web service as you did in practice 5-1A.

Practice 5-2B: Creating a Web Service Client

Overview

In this practice, you create a new web service client to test your new web service.

Assumptions

You have completed practice 5-1B.

Tasks

1. Leave WebLogic running with the `Prac05-02SOAP` project deployed.
2. Create a new Java SE project. Name the project `Prac05-02SOAPClient`.
3. Use your knowledge from practice 5-1B to create a new web service client that tests the `Prac05-02SOAP` project.
4. When complete, the new client should produce the following output:

```
run:
10 + 9 is: 19.0
10 - 9 is: 1.0
10 squared is: 100.0
```

Practices for Lesson 6: Using Java Naming and Directory Interface in a Java EE Environment

Chapter 6

Practices for Lesson 6: Overview

Practices Overview

In these practices, you set up a Java DB database. Then you connect a Java program to the database by using JNDI.

Practice 6-1: Setting Up a Java DB Database

Overview

In this practice, you set up a database by using Java DB. The database is for managing logins and has a single `USERS` table.

Assumptions

You have completed the lecture portion of the course.

Tasks

1. In NetBeans, open the `Prac06-01JNDI` project for this lesson from the `/home/oracle/labs/projects/06_JNDI` directory.
2. Create the `LoginDb` database.
   - Open the Services tab.
   - Open the Databases section of the tab.
   - Right-click JavaDB and select **Start Server** if available.
     - The Java DB database server might already be running.
   - Right-click JavaDB and select **Create Database**.
   - Fill in the following information for the database and click **Ok**:

`Database name:` **`LoginDb`** `User name:` **`oracle`** `Password:` Enter the password as specified in the security credentials page. `Confirm password:` Enter the password as specified in the security credentials page.

   An empty database named `LoginDb` is created. In addition, the interface adds a connection for the database. Double-click the connection to connect to the database server.
3. Set up the `USERS` table.
   - If you have not done so, double-click the `LoginDb` connection to connect NetBeans to the database server.
   - Open the `LoginDb` connection.
   - Expand the information under the `oracle` user.
   - Open the `LoginShortTbl.sql` file from the `/home/oracle/labs/resources` folder.
   - Create the database table by clicking the **Run SQL** icon (looks like a database).
   - Select `LoginDb` from the list of databases.
   - Click **Ok**.
   - Your database is now created. If you open the Tables toggle, you should be able to view the table now.

Practice 6-2: Connecting to the Database Without JNDI

Overview

In this practice, you use Java code to verify the installation of your database.

Assumptions

You have completed the previous practice.

Tasks

1. Continue using the `Prac06-01JNDI` project for this lesson.
2. Run the `Prac06-01JNDI` project to test the database from Java.
3. Click the `DataExample` link to see the results of a database query. The details from the `USERS` table are displayed.
4. Look at the source code for this project. This servlet uses the `DriverManager` class to connect to and query the database. At this time, no JNDI has been used.

Practice 6-3: Connecting to the Database by Using JNDI

Overview

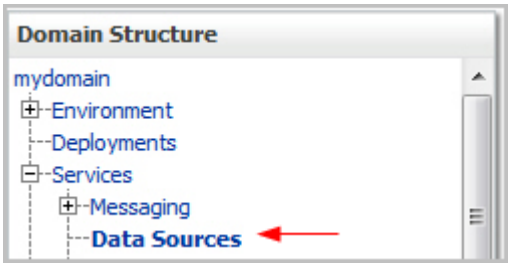
In this practice, you create a JDBC connection pool and a JDBC resource with a JNDI name for the database.

Assumptions

You have completed the previous practice.

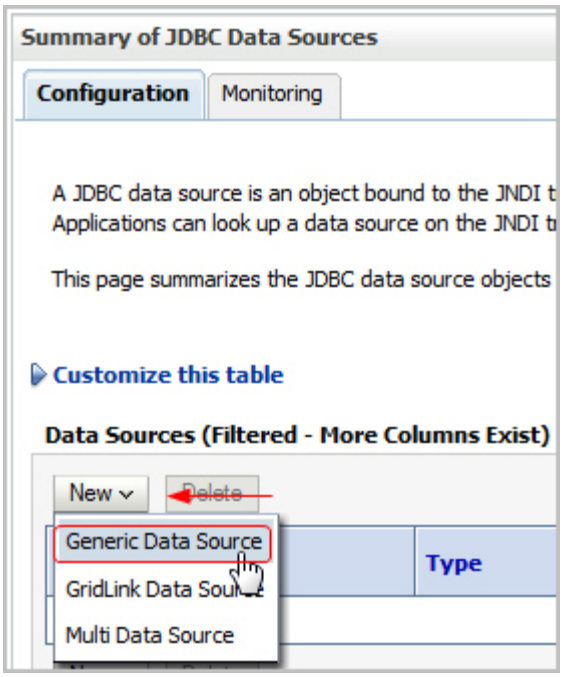
Tasks

1. Continue using the `Prac06-01JNDI` project for this lesson.
2. Add a connection pool for the new database; you need to configure a JDBC data source using the WebLogic Server administration console. Perform the following steps:
   a. Access the WebLogic Server administration console and log in to the console.
   b. Under Domain Structure, expand **Services**, and then click **Data Sources**.



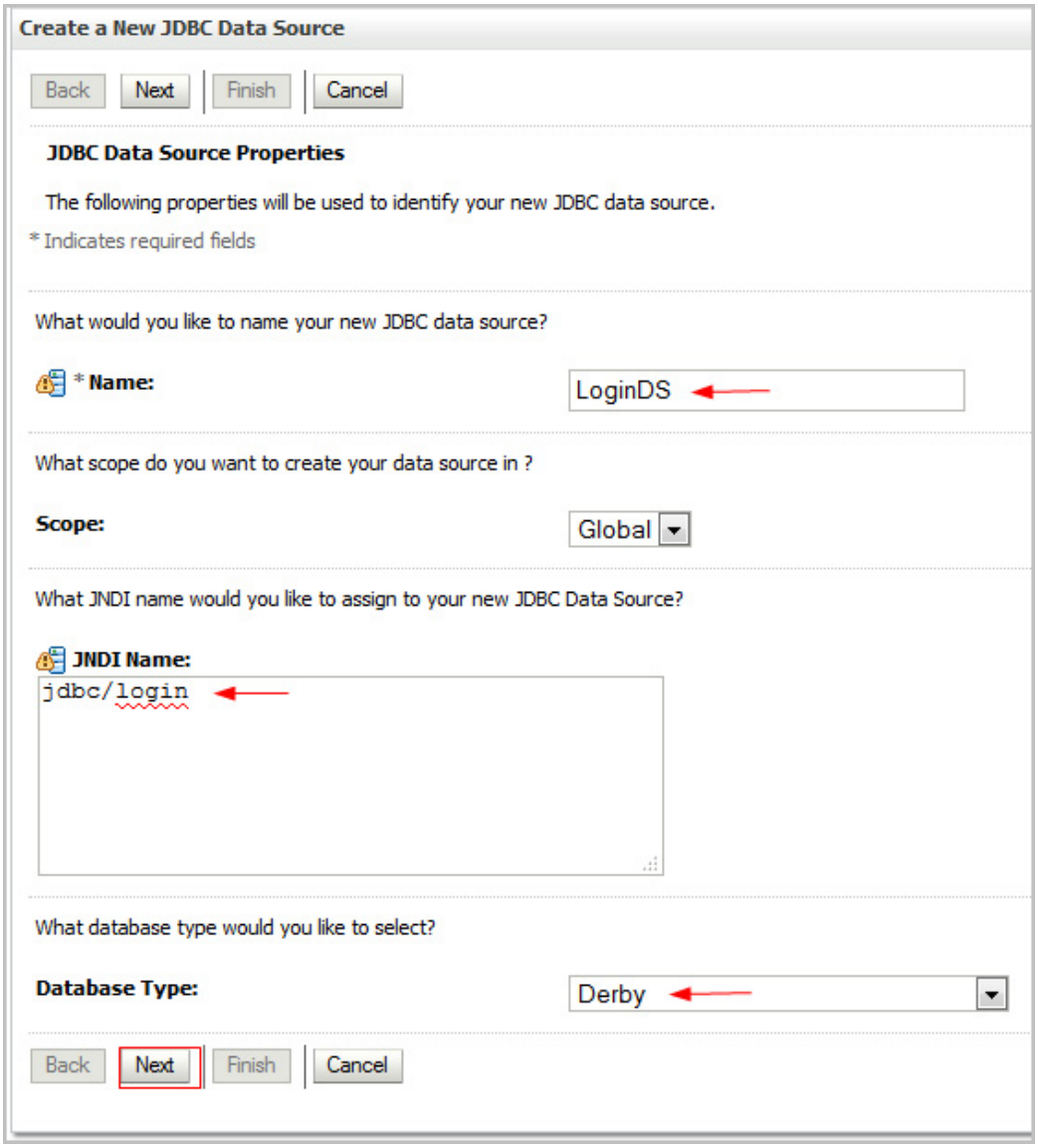
On the right, notice that the Summary of JDBC Data Sources section appears.

   c. Under the Data Sources table heading, click the **New** drop-down list and select **Generic Data Source**.

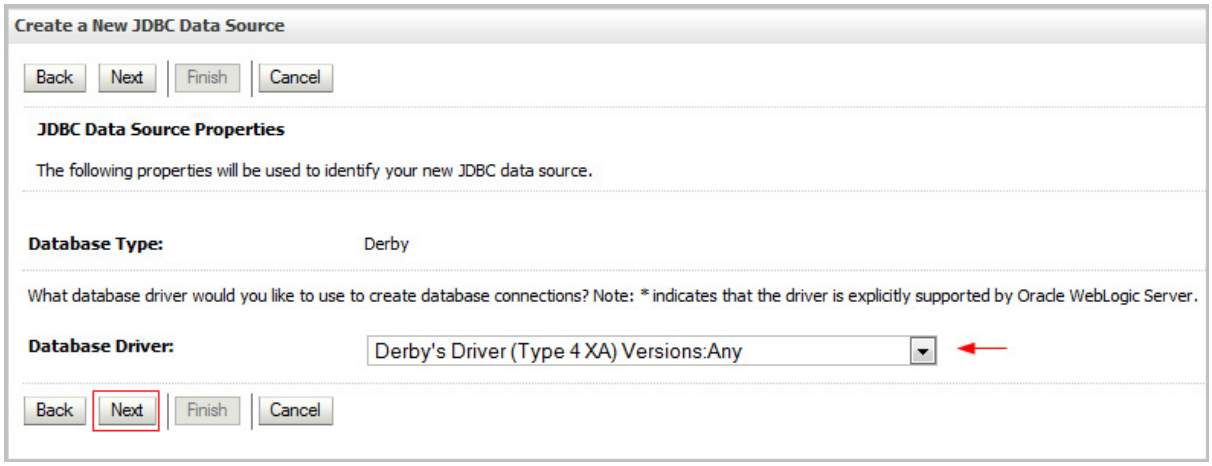


   d. On the first page of the Create a New JDBC Data Source Wizard:
      - Enter `LoginDS` as the data source name.

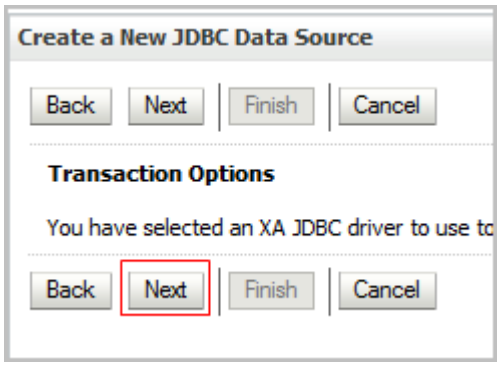
- Enter `jdbc/login` as the JNDI name.
- Select `Derby` from the Database Type drop-down list, and click **Next**.



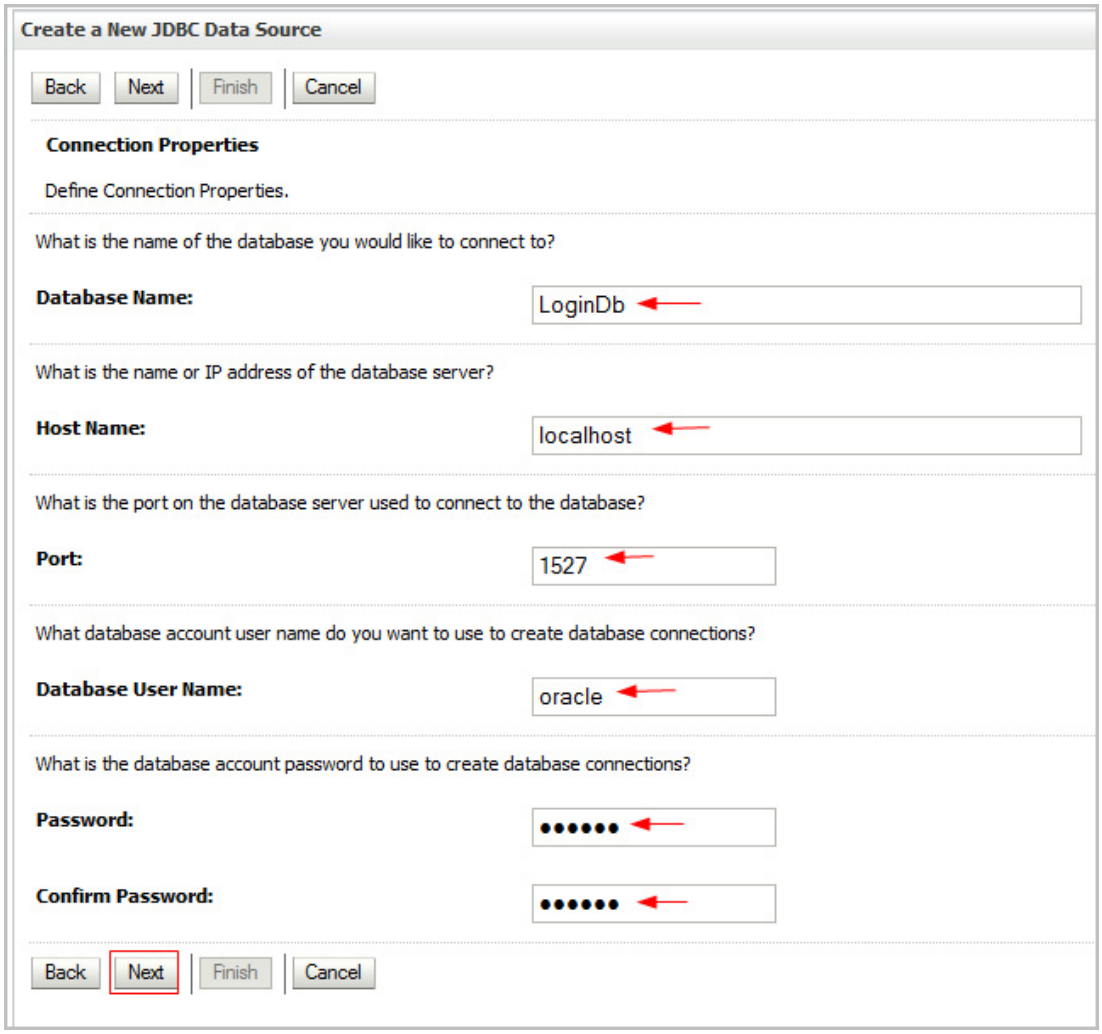
e. Select `Derby's Driver(Type 4XA) Versions:Any` from the Database Driver drop-down list, and click **Next**.



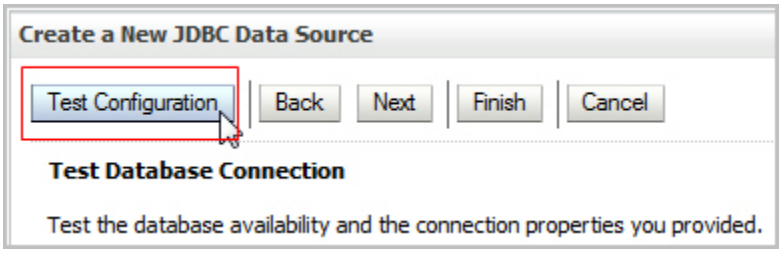
f. On the next page of the wizard, click **Next**.



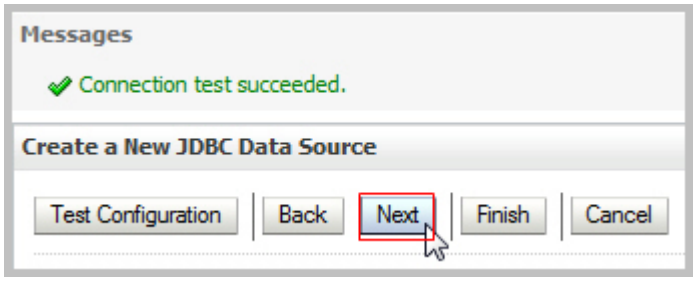
g. On the next page of the wizard, enter the following information and click **Next**:
   - Database Name: `LoginDb`
   - Host Name: `localhost`
   - Port: `1527`
   - Database User Name: `oracle`
   - Password: Enter the password as specified in the security credentials page.
   - Confirm Password: Enter the password as specified in the security credentials page.



h. On the next page of the wizard, click **Test Configuration** to check if you can make a connection to the database based on the information that you entered.

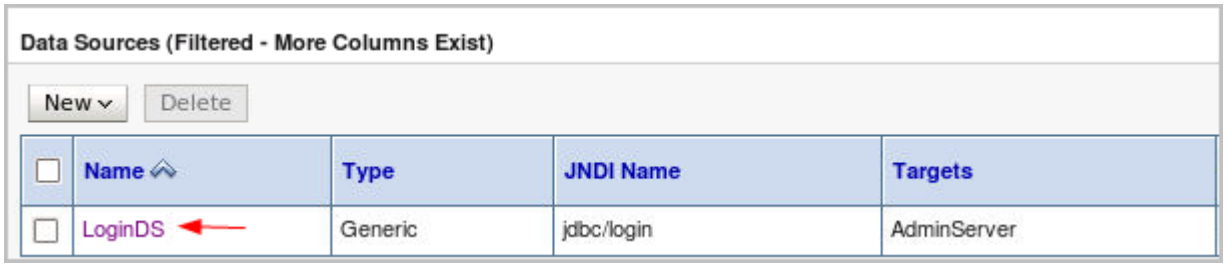


i. The message "Connection test succeeded." is displayed. Click **Next**.

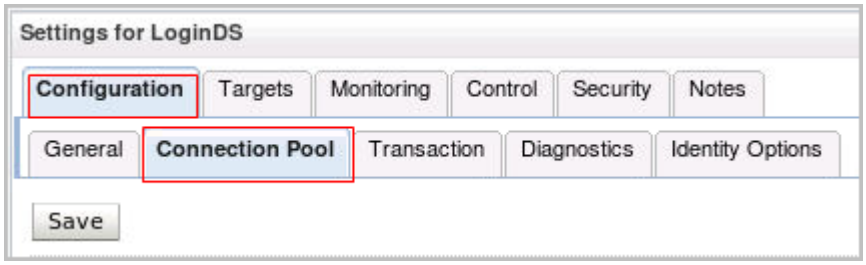


j. On the last page of the wizard, the data source is targeted. Targeting a data source to a server means that the server manages its own instance of the data source. The data source is available as one of the resources of that server. Select **AdminServer** and click **Finish**.

k. In the Data Sources table, click **`LoginDS`** to modify its configuration.



l. Under Settings for `LoginDS`, click the **Configuration** tab and the **Connection Pool** subtab.



m. Scroll down to the capacity fields, enter the following values, and then click **Save**.

- Initial Capacity: **2**
- Maximum Capacity: **15**
- Minimum Capacity: **2**

n. Under Domain Structure, expand **Environment** and click **Servers**.

o. In the Servers table, click **AdminServer**.

p. Click the **View JNDI Tree** link.

- The JNDI tree opens in a new browser window (or tab).
- Expand `jdbc`.
- `login` is displayed.

Note: Other entries in the JNDI tree of your server may be very different from what is shown here. It depends on your server's resources.

3. Right-click `DataExampleJndi.java` and select **Run File** from the context menu. A browser window should be launched with a link to the servlet.
4. Click `DataExampleJNDI` to test the servlet. The details from the `USERS` table are displayed.
5. Look at the source code for this project. Notice that in this case, a JNDI context is used to look up a name for the database.

Practices for Lesson 7: The Full EJB Component Model

Chapter 7

Practices for Lesson 7: Overview

Practices Overview

In the first practice, you develop a stateless session bean and expose it to create a JAX-WS web service.

In the second practice, you create a remote interface with an EJB stateless session bean. You also create an application remote client that accesses the session bean.

Practice 7-1: Creating a JAX-WS Web Service with an EJB Stateless Session Bean

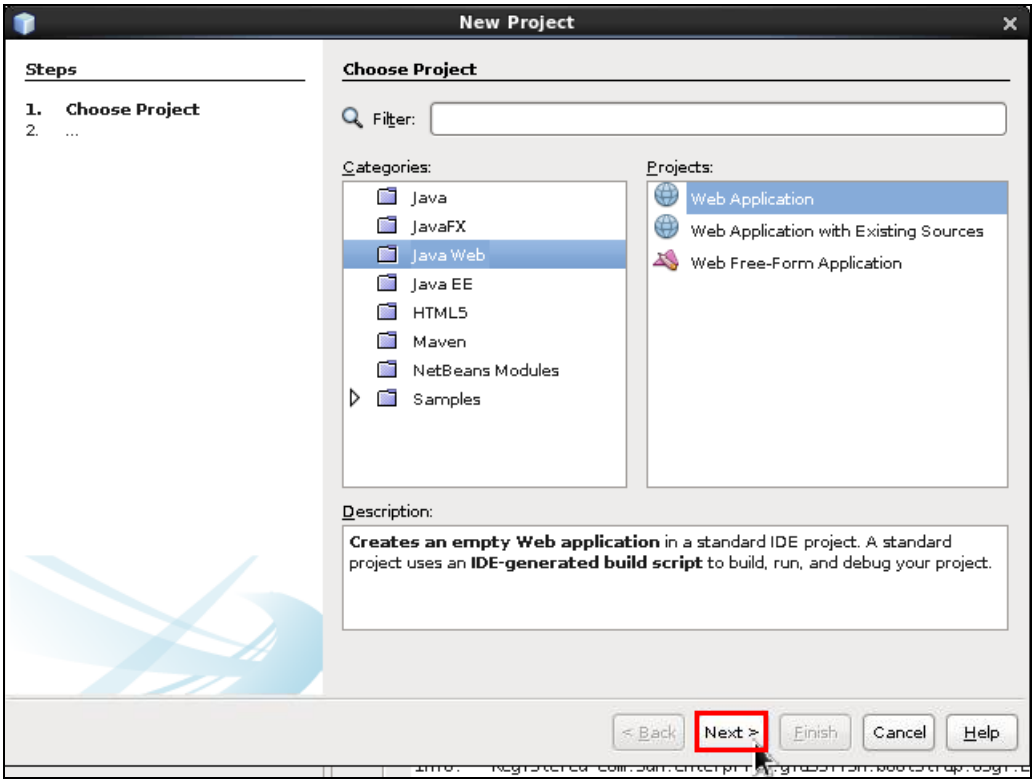
Overview

In this practice, you create a web application where you deploy your EJB directly. You create a web service from a stateless session bean and finally deploy and test the web service.

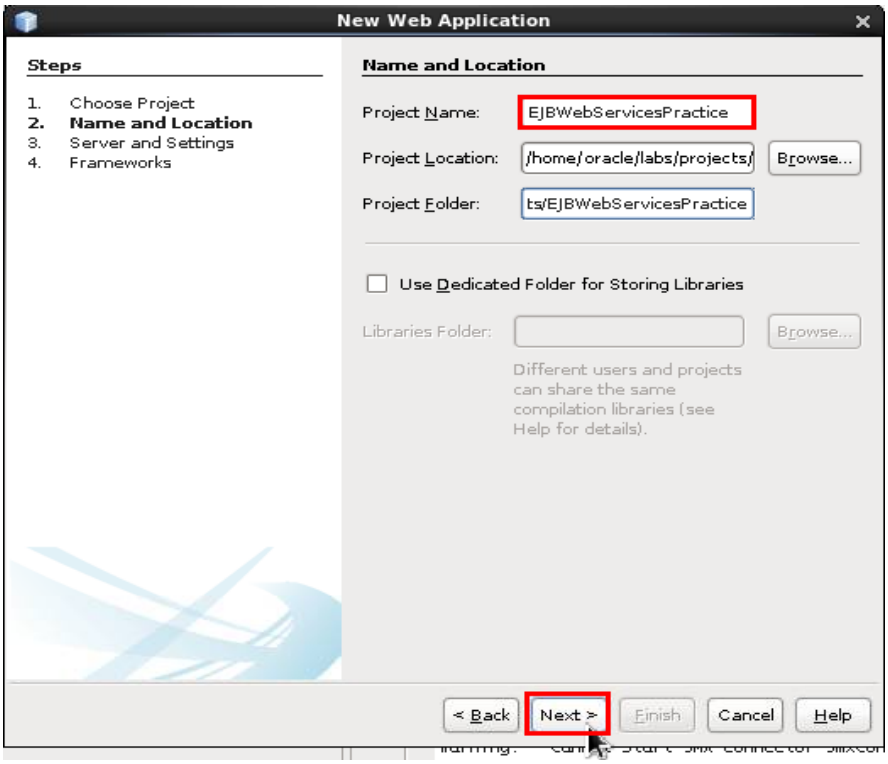
Tasks

Creating a Web Application in NetBeans

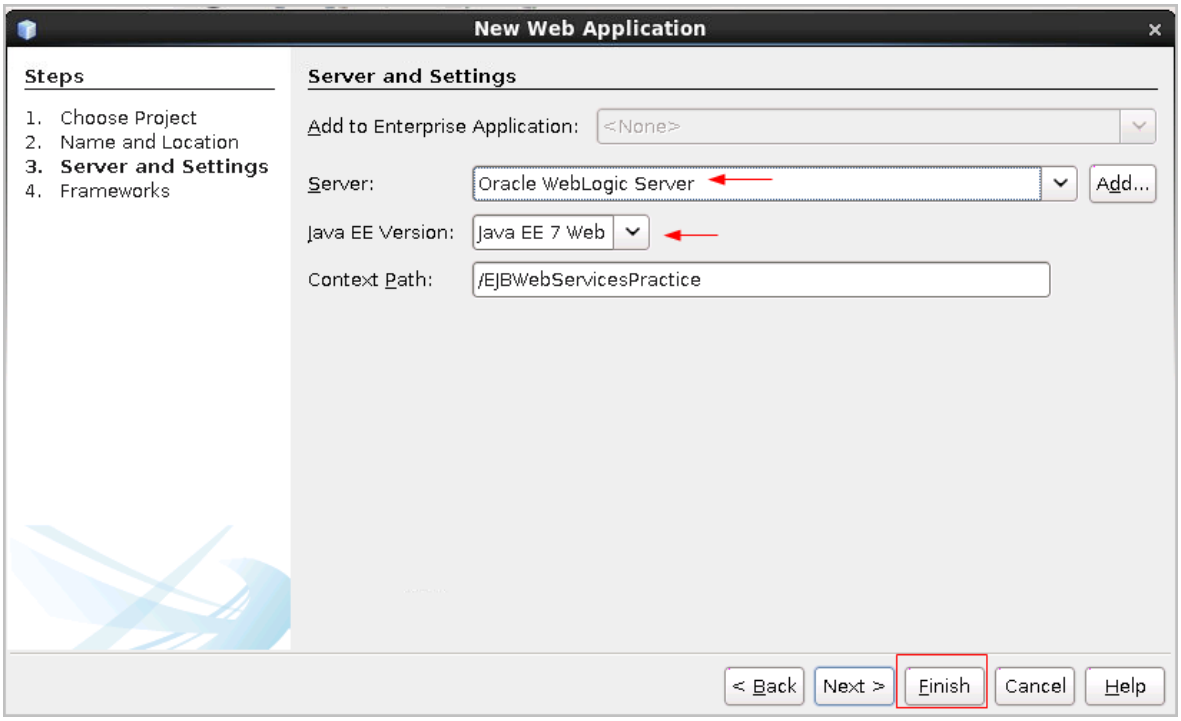
1. Create a new web application by performing the following steps:
   - Select **File > New Project** from the NetBeans menu.
   - Select **Java Web** from Categories and **Web Application** from Projects.
   - Click **Next**.



2. Enter `EJBWebServicesPractice` as Project Name. Click **Next**.



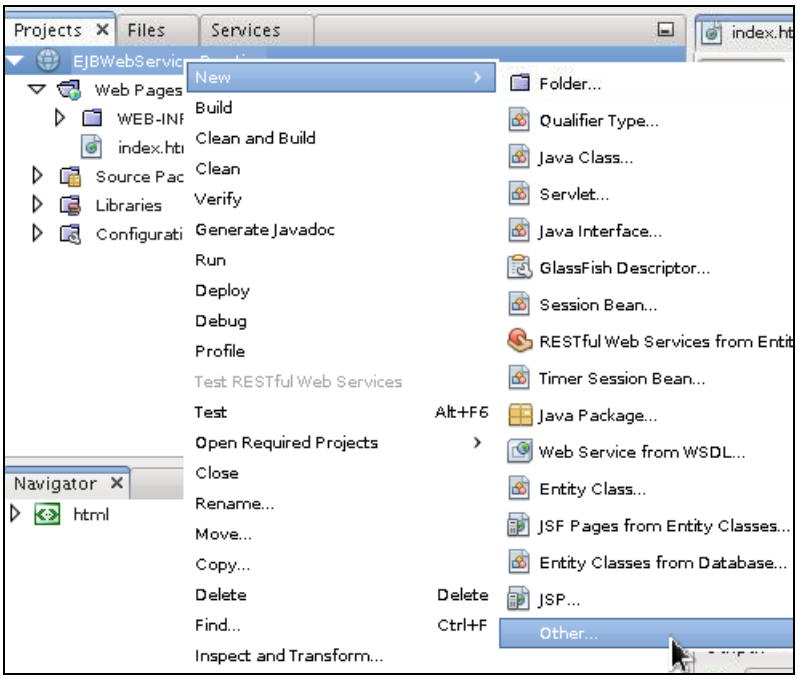
3. Verify that the server is WebLogic Server and that the Java EE version is Java EE 7 Web. Click **Finish**.



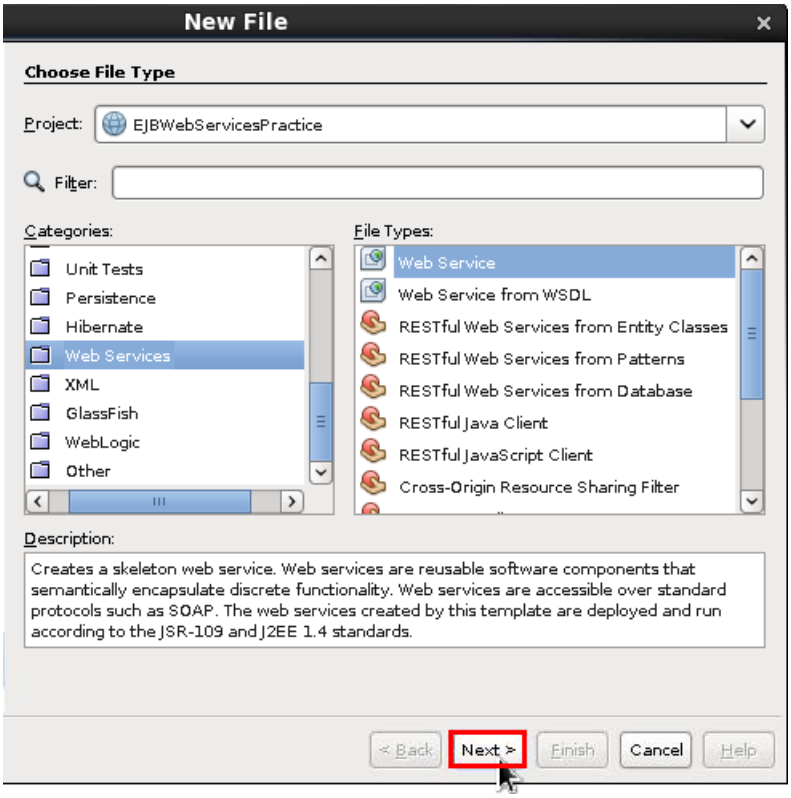
Creating a Web Service from a Stateless Session Bean

In this section, you create a web service from an EJB stateless session bean.

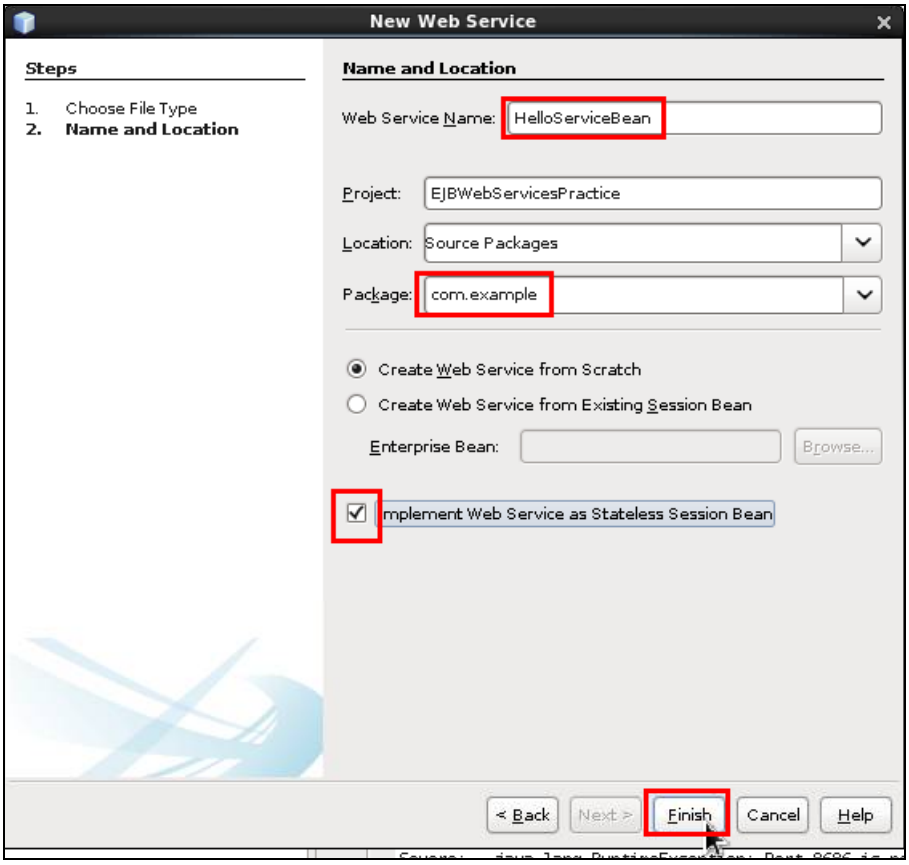
4. Right-click the **`EJBWebServicesPractice`** project and select New > Other.



5. Select Web Services from Categories and Web Service from File Types. Click **Next**.



6. Enter the Web Service information as follows:
   - Web Service Name: `HelloServiceBean`
   - Package Name: `com.example`
   - Select **Implement Web Service as Stateless Session Bean**.
   - Click **Finish**.

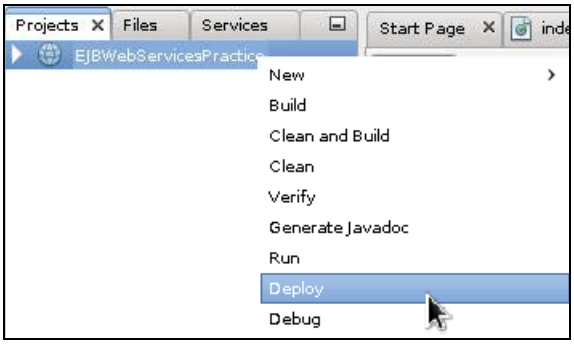


Deploying and Testing the Web Service

To deploy and test the web service, perform the following steps in the NetBeans IDE. You can follow the progress of these operations on the **`EJBWebServicesPractice`** tab and the **WebLogic Server** tab in the Output window.

7. Right-click the **`EJBWebServicesPractice`** project in the Projects window and select **Build**.

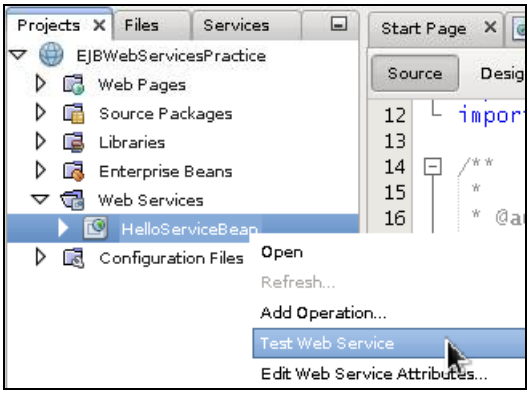
8. In the Projects window, right-click **`EJBWebServicesPractice`** and select **Deploy**.



The command builds and packages the application into **`EJBWebServicesPractice.war`**, and deploys this WAR file to the WebLogic Server.

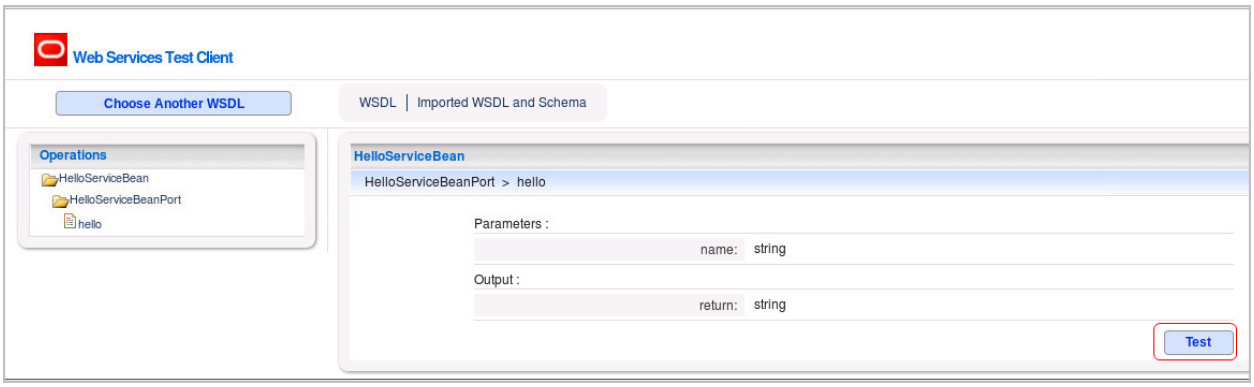
9. To test the `hello` method of `HelloServiceBean`, complete the following steps:

- On the Projects tab, expand the Web Services node of the **`EJBWebServicesPractice`** project.
- Right-click **`HelloServiceBean`** and select Test Web Service.

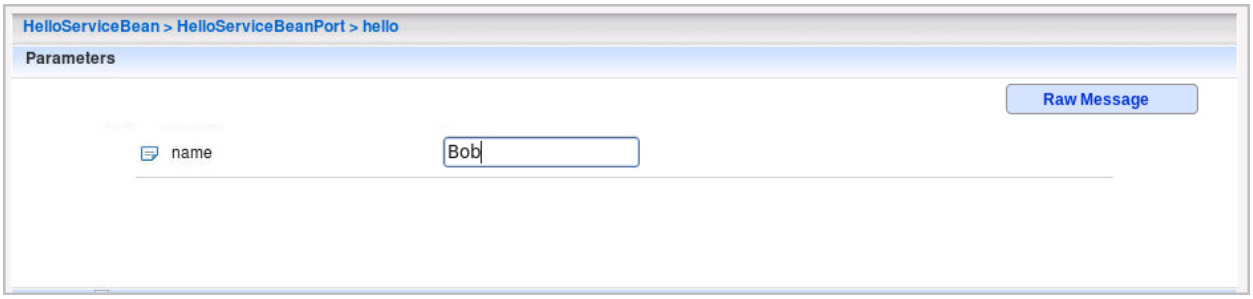


The tester page opens in the browser.

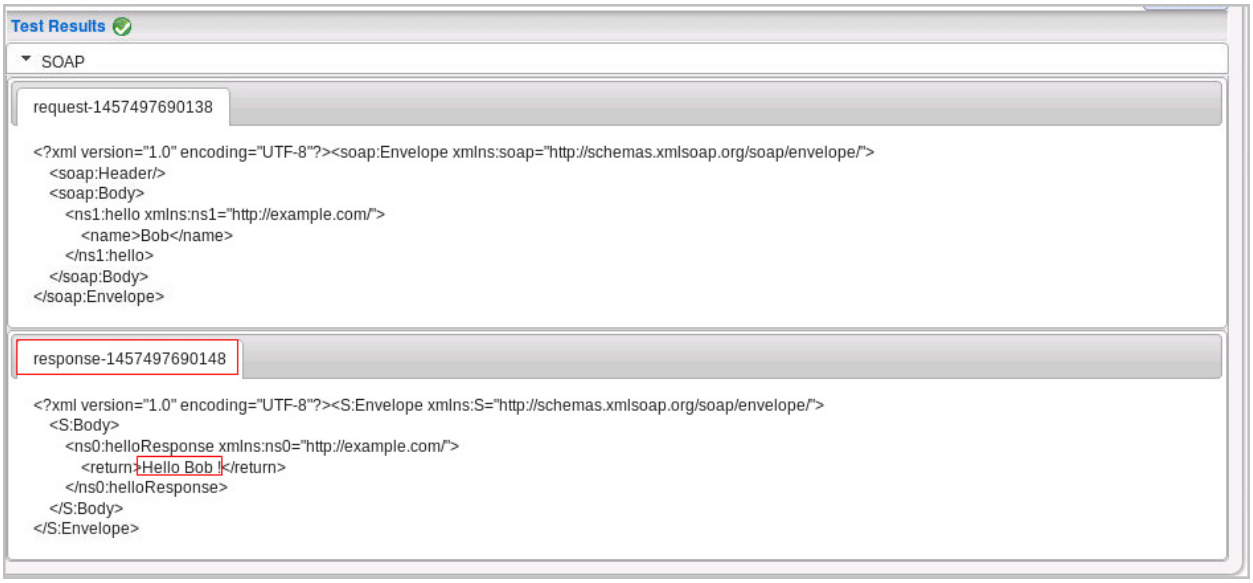
10. Click Test.



- Enter a String value for the name field and click Invoke at the bottom of the page.



The Test Results is displayed.



Practice 7-2: Creating a Remote Interface with an EJB Stateless Session Bean

Overview

In this practice, you start by creating a Java class library project that contains the remote interfaces for the session bean. Then you create a session bean and its methods in an enterprise application. Finally you create an application remote client that accesses the session bean.

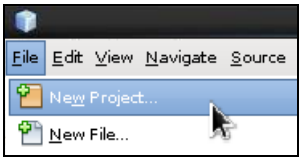
Tasks

Creating a Java Class for the Remote Interface

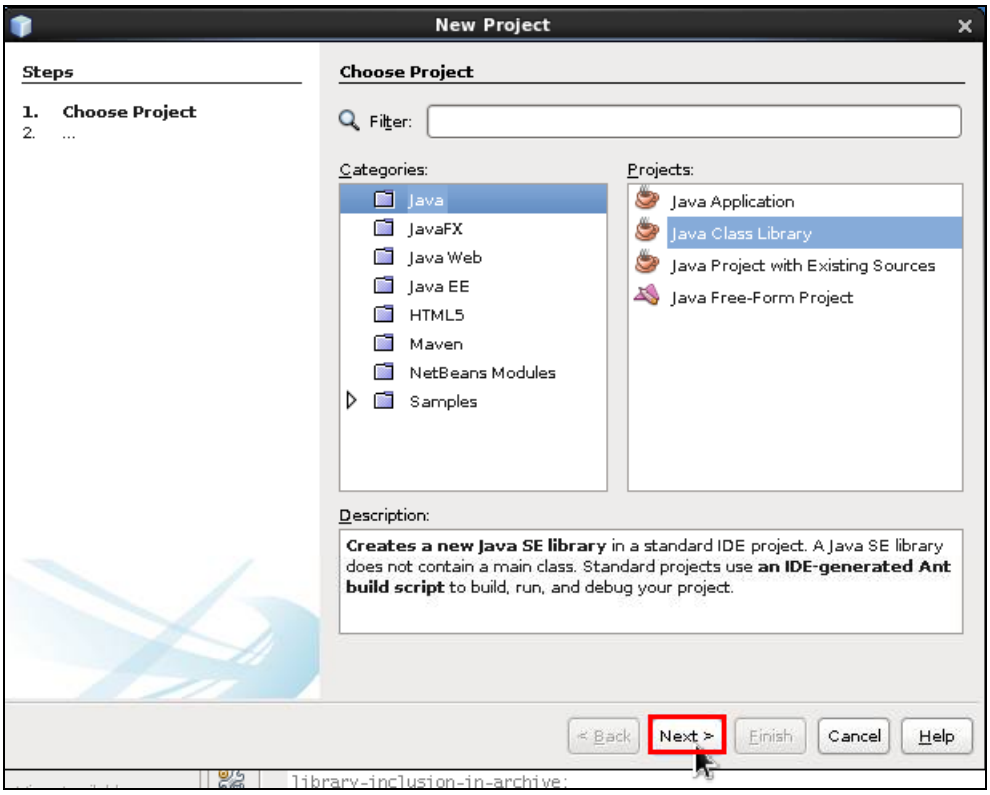
In this section, you create a Java class library project that contains the remote interfaces for the session bean.

The compiled class library JAR is added to the classpath of the EJB module and the application client that is used to call the session bean.

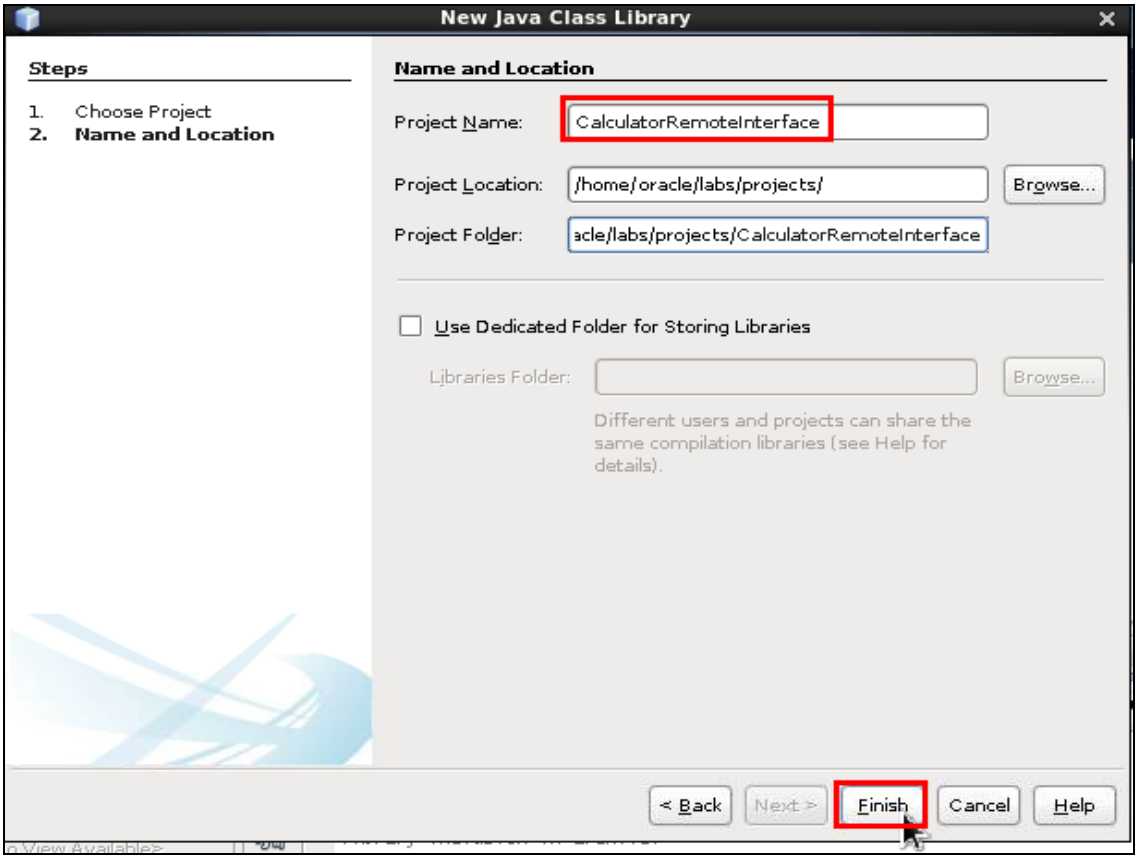
1. Select File > New Project.



2. Select the Java category and Java Class Library under Projects. Click **Next**.



3. Enter `CalculatorRemoteInterface` as Project Name. Click **Finish**.



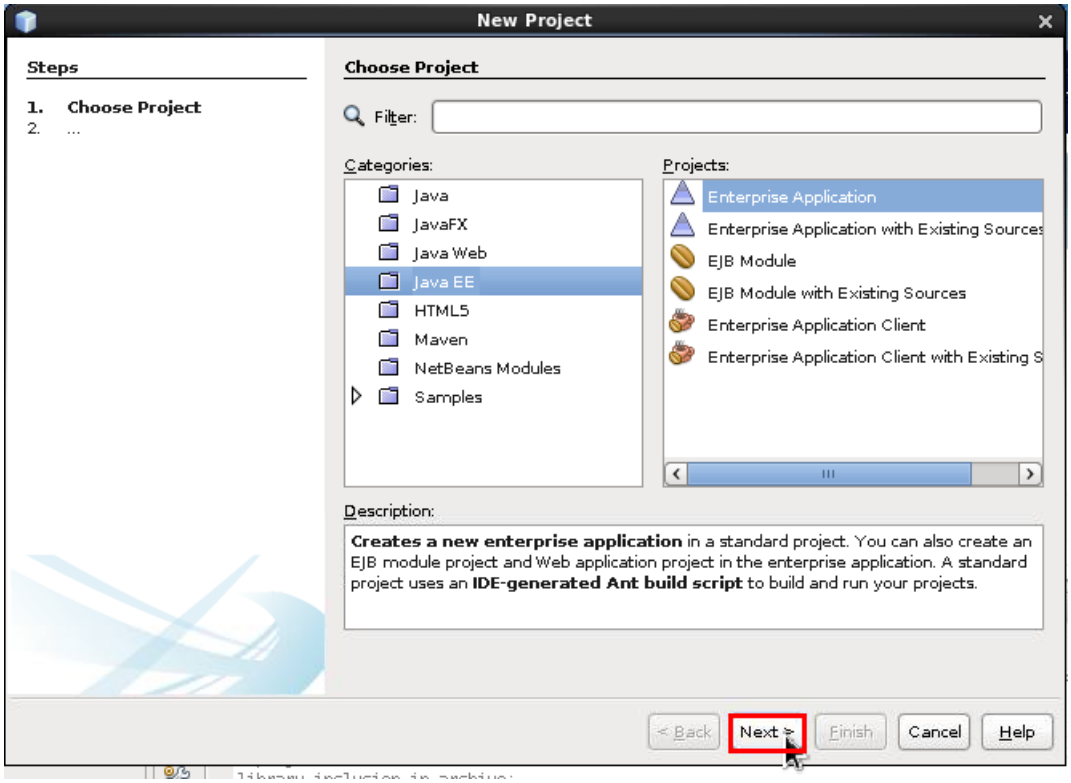
The **`CalculatorRemoteInterface`** project is added in the Projects window.

In the next section, you create a session bean in an enterprise application. The session bean is accessed via a remote interface. When you create the session bean, the IDE automatically creates the remote interface in the class library and adds the class library JAR to the classpath of the enterprise application.

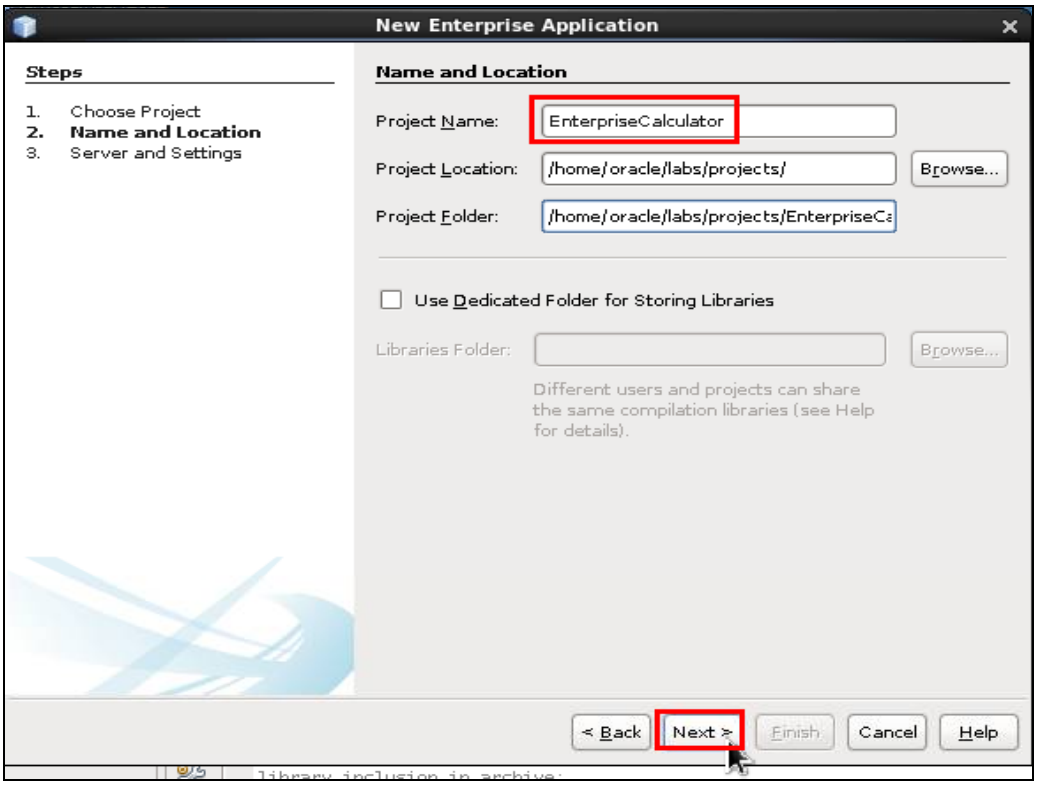
Creating the Enterprise Application Project

In this section, you create an enterprise application that contains an EJB module.

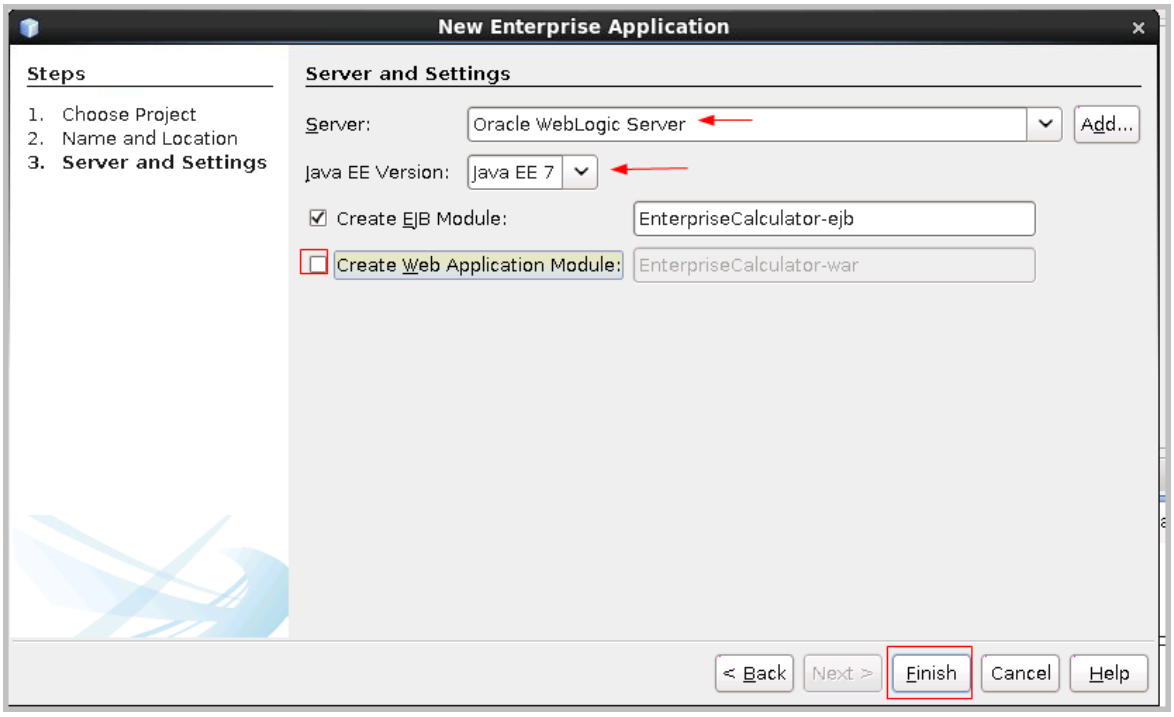
1. Select File > New Project.
2. Select Java EE from Categories and Enterprise Application from Projects. Click **Next**.



3. Enter `EnterpriseCalculator` as Project Name. Click **Next**.



4. Make sure that WebLogic Server is selected as the server and that Java EE 7 is the Java EE Version.
   - Select Create EJB Module.
   - Deselect Create Web Application Module.
   - Click **Finish**.

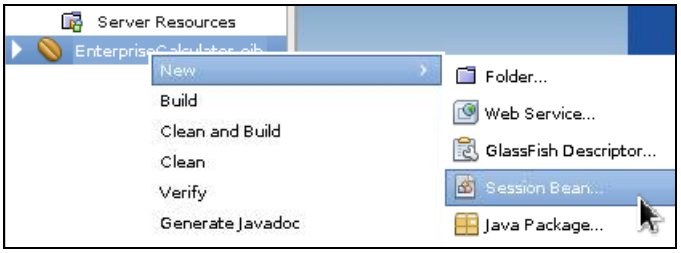


After this, the Enterprise Application project is created. Now you create a session bean in the EJB module project.

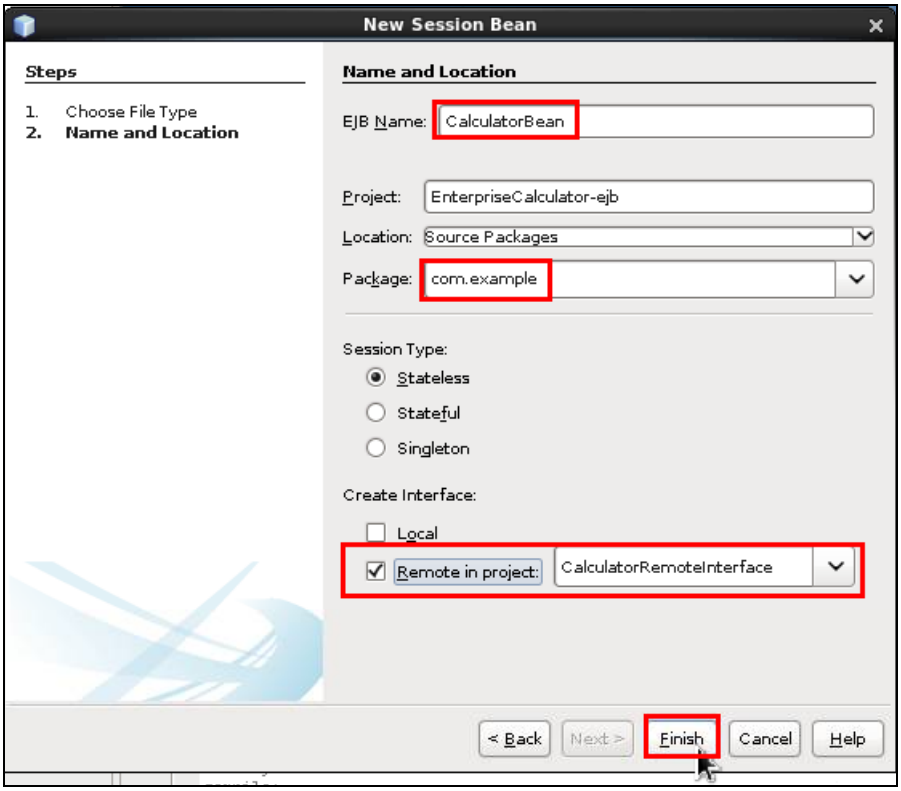
Creating Methods in a Session Bean

In this section of the practice, you create a session bean in the EJB module project and create the `add` and `subtract` methods.

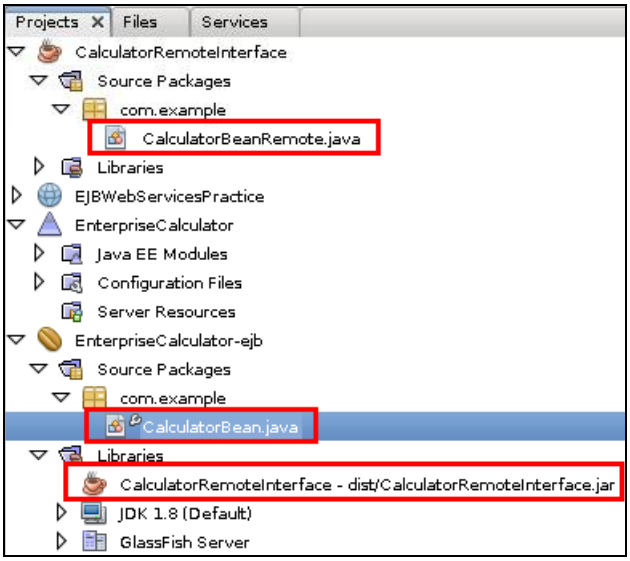
1. Right-click the **`EnterpriseCalculator-ejb`** module in the Projects window. Select New > Session Bean:



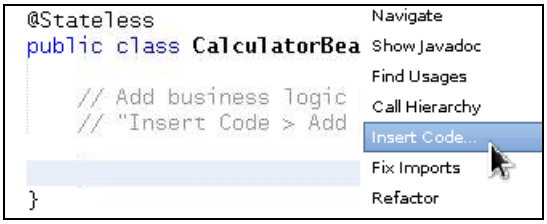
2. Perform the following actions in the New Session Bean window:
   - Enter `CalculatorBean` as EJB Name.
   - Enter `com.example` as Package.
   - Select “Remote” and select **`CalculatorRemoteInterface`** from the drop-down list.
   - Click **Finish**.



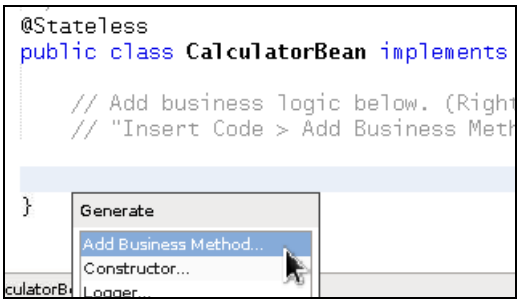
After you click Finish, the IDE creates the `CalculatorBean` class and opens the file in the source editor. It also creates the `CalculatorBeanRemote` remote interface for the bean in the bean package in the `CalculatorRemoteInterface` project. In addition to this, it adds the `CalculatorRemoteInterface` JAR to the classpath of the `EnterpriseCalculator-ejb` project. These three files are highlighted in red in the following screenshot:



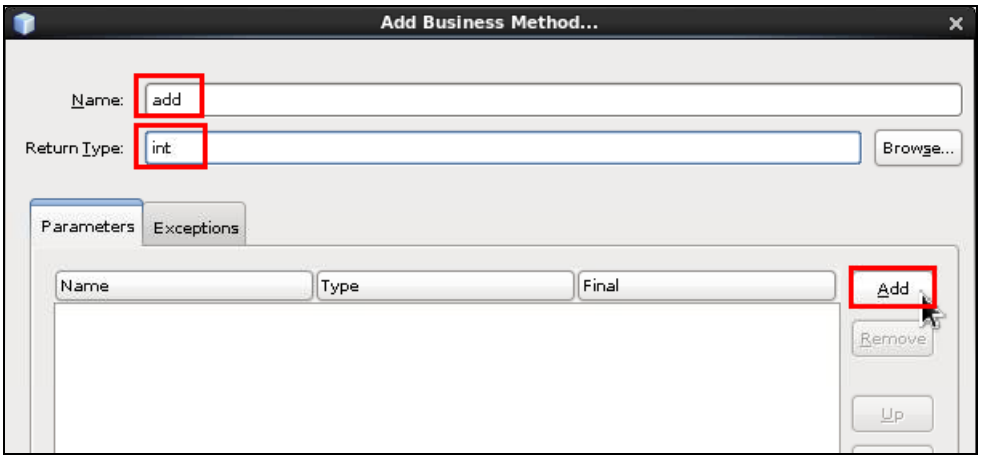
3. In the source editor, right-click in the class `CalculatorBean`. Select Insert Code.



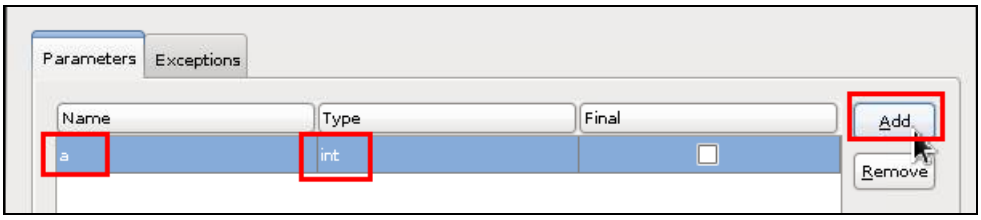
4. Select Add Business Method.



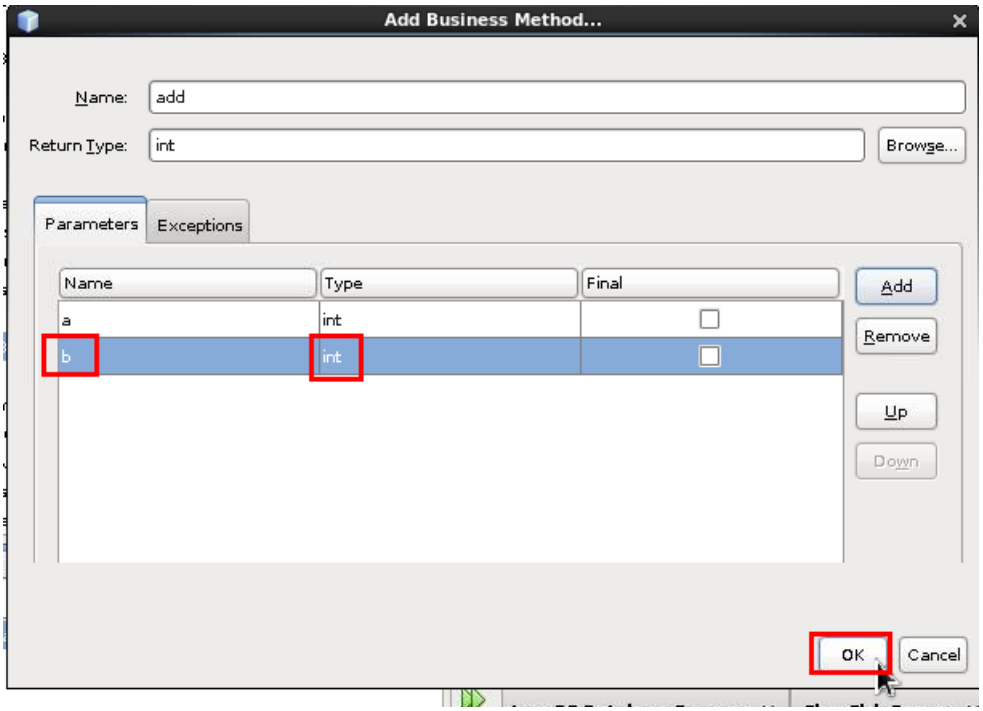
5. Enter `add` for Name and `int` for Return Type. Click **Add**.



6. Enter `a` for parameter Name and `int` for Type. Click **Add**.



7. Enter `b` for parameter Name and `int` for Type. Click **OK**.



In addition to adding the code in the `CalculatorBean` class, the IDE automatically exposes the business method in the remote interface:

```
@Remote
public interface CalculatorBeanRemote {

    int add(int a, int b);

}
```

8. In the source editor, for the `CalculatorBean` class, replace the return line of code of the `add` method with the following line:

```
return a + b;
```

```
@Stateless
public class CalculatorBean implements CalculatorBeanRemote {

    @Override
    public int add(int a, int b) {
        return a + b;
    }
```

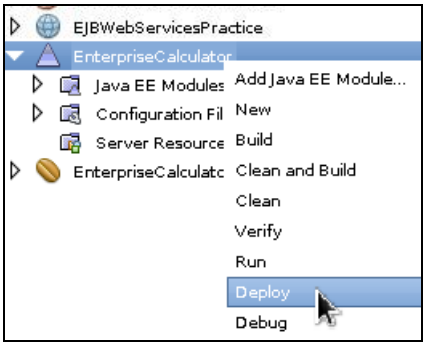
9. Repeat steps 3 through 8 to add another method to the `CalculatorBean` class called `subtract`, which returns `a - b`.

```
@Stateless
public class CalculatorBean implements CalculatorBeanRemote

    @Override
    public int add(int a, int b) {
        return a + b;
    }

    // Add business logic below. (Right-click in editor and
    // "Insert Code > Add Business Method")

    @Override
    public int subtract(int a, int b) {
        return a - b;
    }

}
```

You now have an enterprise application with an EJB that is exposed through a remote interface. You also have an independent class library (`CalculatorRemoteInterface`) that contains the EJB interface that can be distributed. You can add the library to your projects if you want to communicate with the EJB that is exposed by the remote interface and do not need to have the sources for the EJB. When you modify the code for the EJB, you only need to distribute a JAR of the updated class library if any of the interfaces change.

Deploying the Enterprise Application

10. Right-click the `EnterpriseCalculator` enterprise application and select **Deploy**.



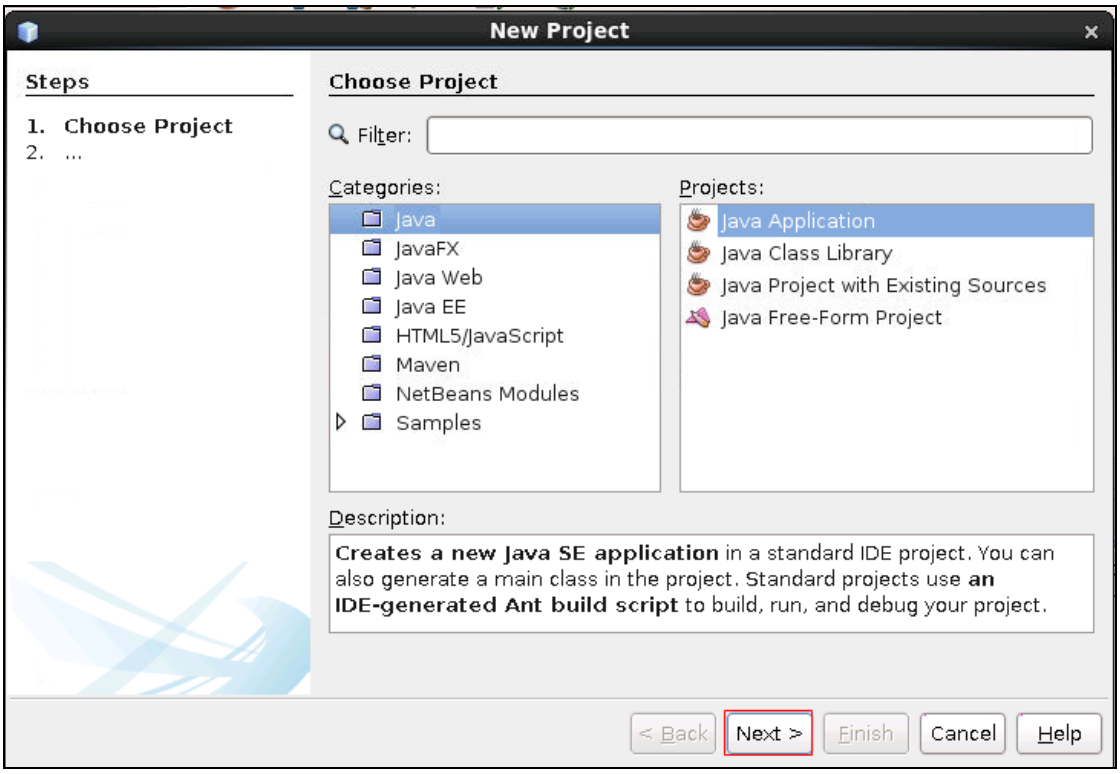
Creating an Application Remote Client That Accesses the Session Bean

In this section, you create an enterprise application client that accesses the session bean through the remote interface in the class library. The project needs the `CalculatorRemoteInterface` Java class library as a library in order to reference the EJB.

1. Create a stand-alone Java SE application

a. Select File > New Project.

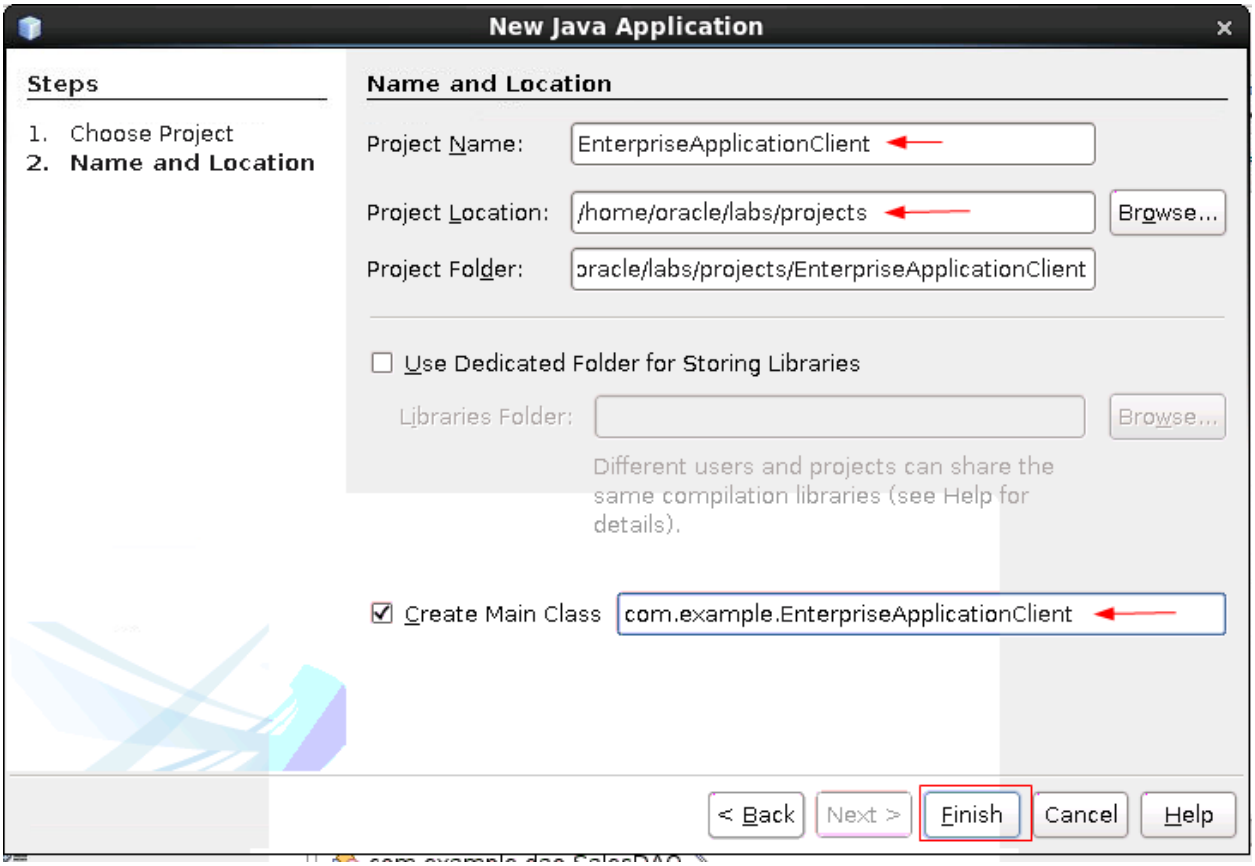
b. Select Java from Categories and Java Application from Projects. Click **Next**.



c. Enter `EnterpriseCalculatorClient` as Project Name.

d. Modify the `Main` class name to `com.example.EnterpriseCalculatorClient`.

e. Click **Finish**.



Install client libraries

You will need to configure the `EnterpriseCalculatorClient` project to reference the client libraries of the `EnterpriseCalculator` project. Additionally, you will need to configure the `EnterpriseCalculatorClient` project to reference the WebLogic runtime libraries. This will enable you to use the WebLogic JNDI provider implementation.

1. Add the `CalculatorRemoteInterface` library to `EnterpriseCalculatorClient`.
   a. Right-click `EnterpriseCalculatorClient` and select Properties > Libraries.
   b. Click Add Project.
   c. In the Add Project dialog box, browse to /home/oracle/labs/projects.
   d. Select the `CalculatorRemoteInterface` project.
   e. Click Add Project Jar Files.
   f. Click OK.
2. Add the WebLogic runtime libraries to `EnterpriseCalculatorClient`.
   a. Right-click `EnterpriseCalculatorClient` and select Properties > Libraries.
   b. Click Add Jar/Folder.
   c. In the file chooser, browse to `/home/oracle/weblogic/wls12210/wlserver/server/lib`.
   d. Select `weblogic.jar`.
   e. Click OK.

3. To perform the JNDI global lookup of the `CalculatorBean` and invoke the `add` and `sub` methods, complete the following steps:
   a. Edit `EnterpriseCalculatorClient.java`.
      - Include the following import statements:

```
import com.example.CalculatorBeanRemote;
import java.util.Hashtable;
import javax.naming.*;
```

      - Modify the `main` method to invoke the EJB.

```
String jndiPath =
"java:global/EnterpriseCalculator/EnterpriseCalculator-
ejb/CalculatorBean";
try {
    final Context cxt = getInitialContext();
    System.out.println("standaloneapp.main: looking up bean at:
" + jndiPath);
    CalculatorBeanRemote CalculatorBean = (CalculatorBeanRemote)
cxt.lookup(jndiPath);
   System.out.println("5 + 5 = " + CalculatorBean.add(5, 5));
   System.out.println("5 - 5 = " +CalculatorBean.subtract(5,5));
}
catch (Exception e) {
    System.err.println("Could not find CalculatorBeanRemote");
    e.printStackTrace();
    }
}
```

      - Add the `getInitialContext` method to obtain a reference to JNDI.

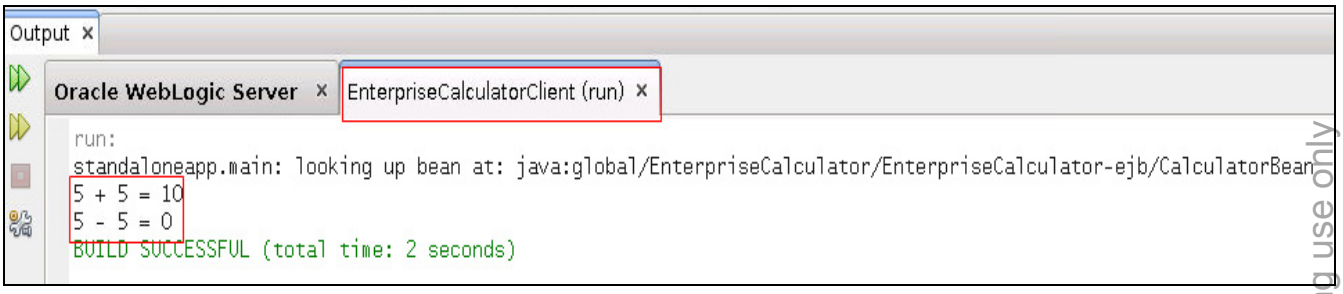
```
private static InitialContext getInitialContext() throws
NamingException {
        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
        env.put(Context.PROVIDER_URL, "t3://localhost:7001");
        return new InitialContext(env);
    }
```

4. Run the `EnterpriseCalculatorClient` application.

To execute the `EnterpriseCalculatorClient` application, complete the following steps:

   a. Verify that the `EnterpriseCalculator` application is deployed and running in WebLogic Server.
   b. On the Projects tab, right-click the `EnterpriseCalculatorClient` project and select Run.

c. Examine the output on the EnterpriseCalculatorClient(run) tab. The output will be as shown below.



Practices for Lesson 8: Contexts and Dependency Injection

Chapter 8

Practices for Lesson 8: Overview

Practices Overview

In these practices, you use the NetBeans IDE to set up a Java web project with CDI.

Practice 8-1: Injecting a Bean with CDI

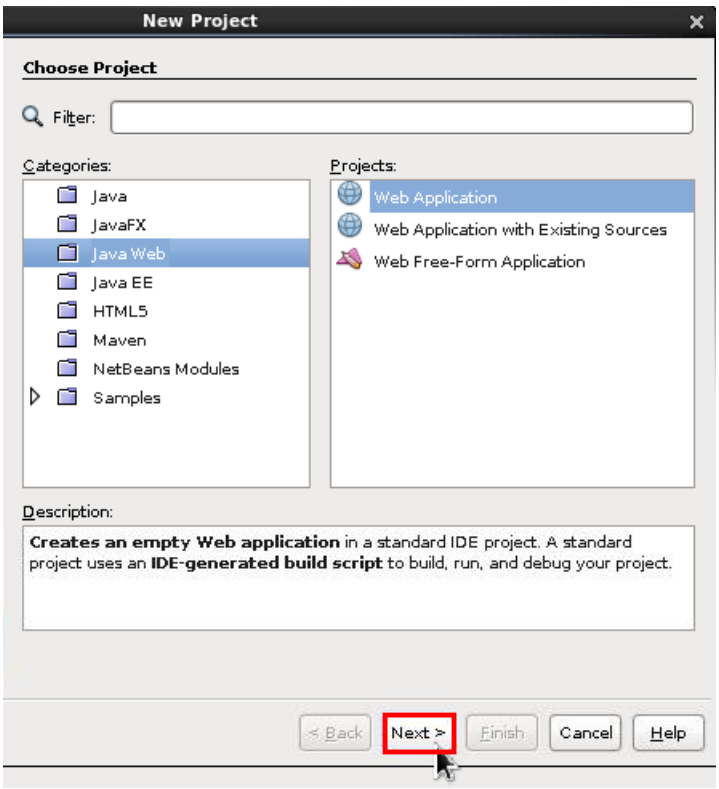
Overview

In this practice, you use the NetBeans IDE to create a web application. In the `index.html` file of the web application, you inject a bean that implements an interface that you also built previously.

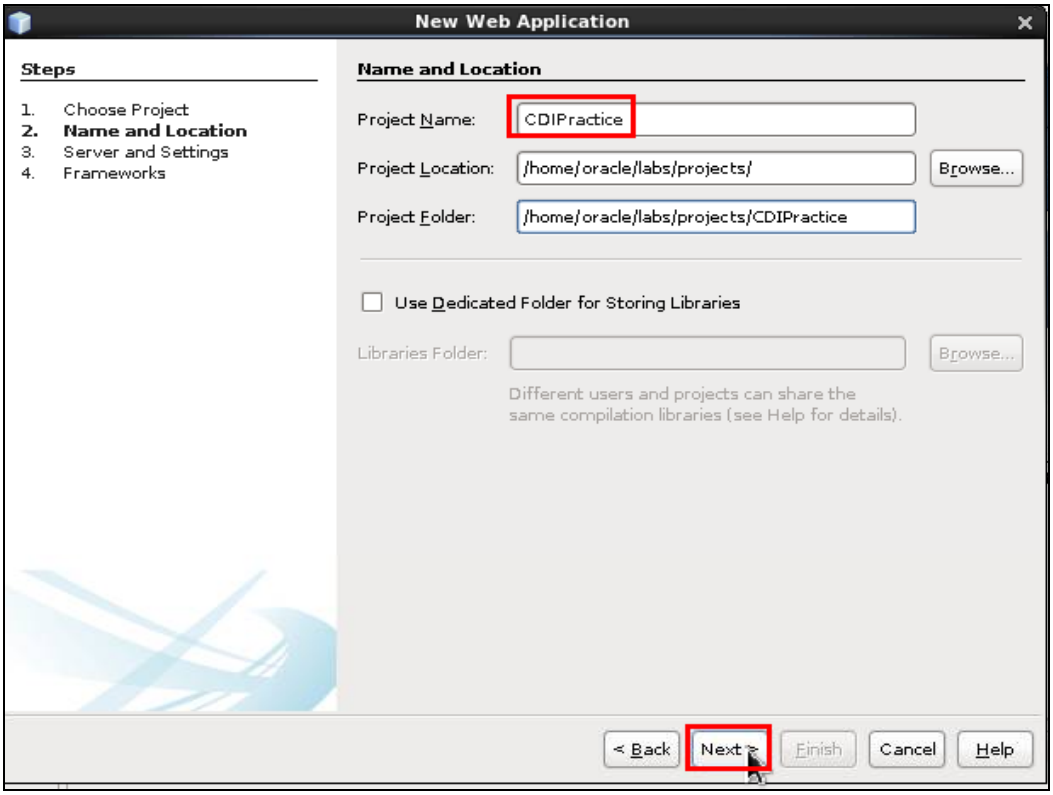
Tasks

Creating a Web Application

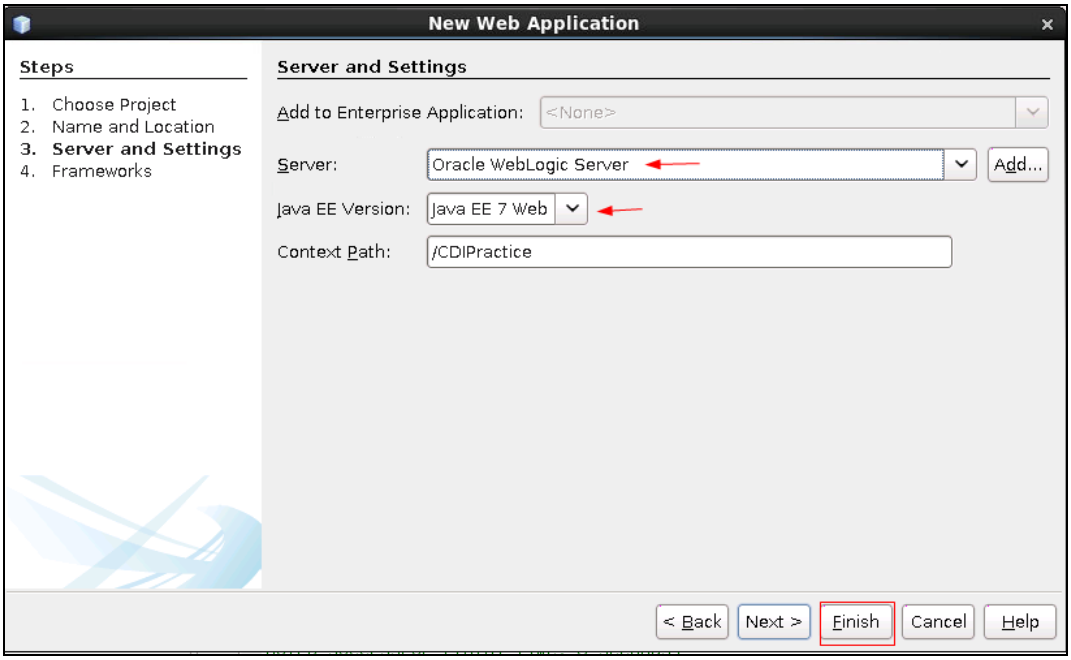
1. Open NetBeans 8.
2. Create a new web application by performing the following steps:
   - Select **File > New Project** from the NetBeans menu.
   - Select **Java Web** from Categories and **Web Application** from Projects.
   - Click **Next**.



3. Enter `CDIPractice` as Project Name. Click **Next**.



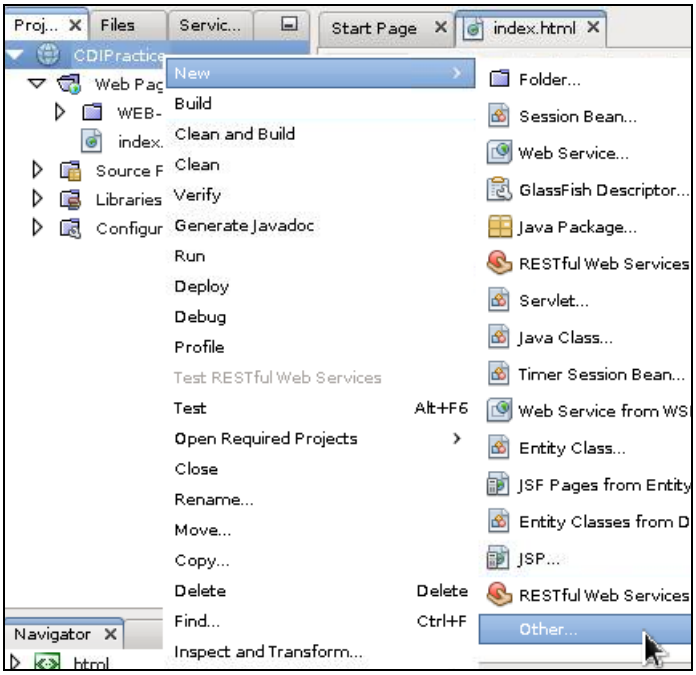
4. Verify that the server is WebLogic Server and that the Java EE version is Java EE 7 Web. Click **Finish**.



A new project called `CDIPractice` is created. The `index.html` file appears in the Source pane.

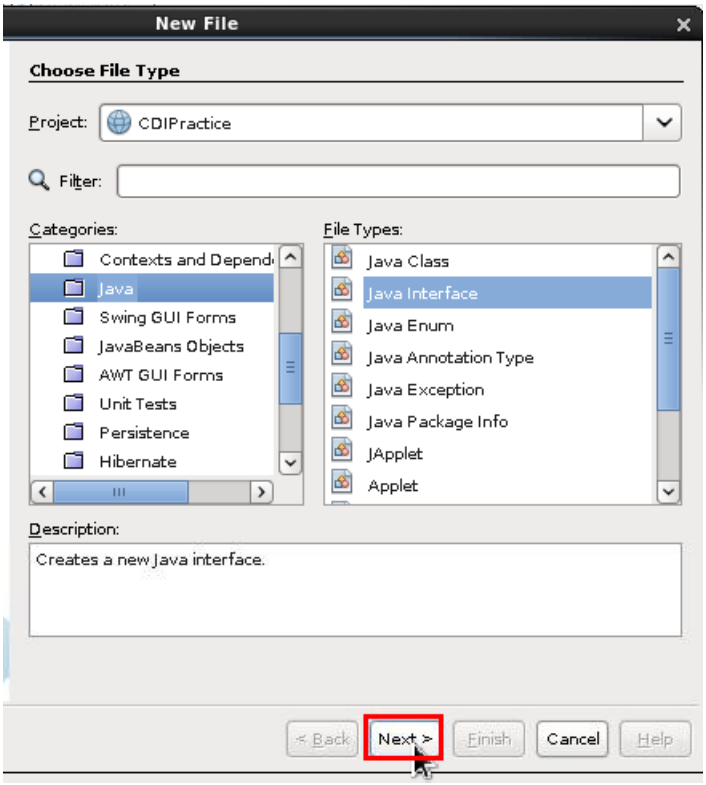
Creating the Message Interface

5. Right-click the **CDIPractice** project. Select New > Other.



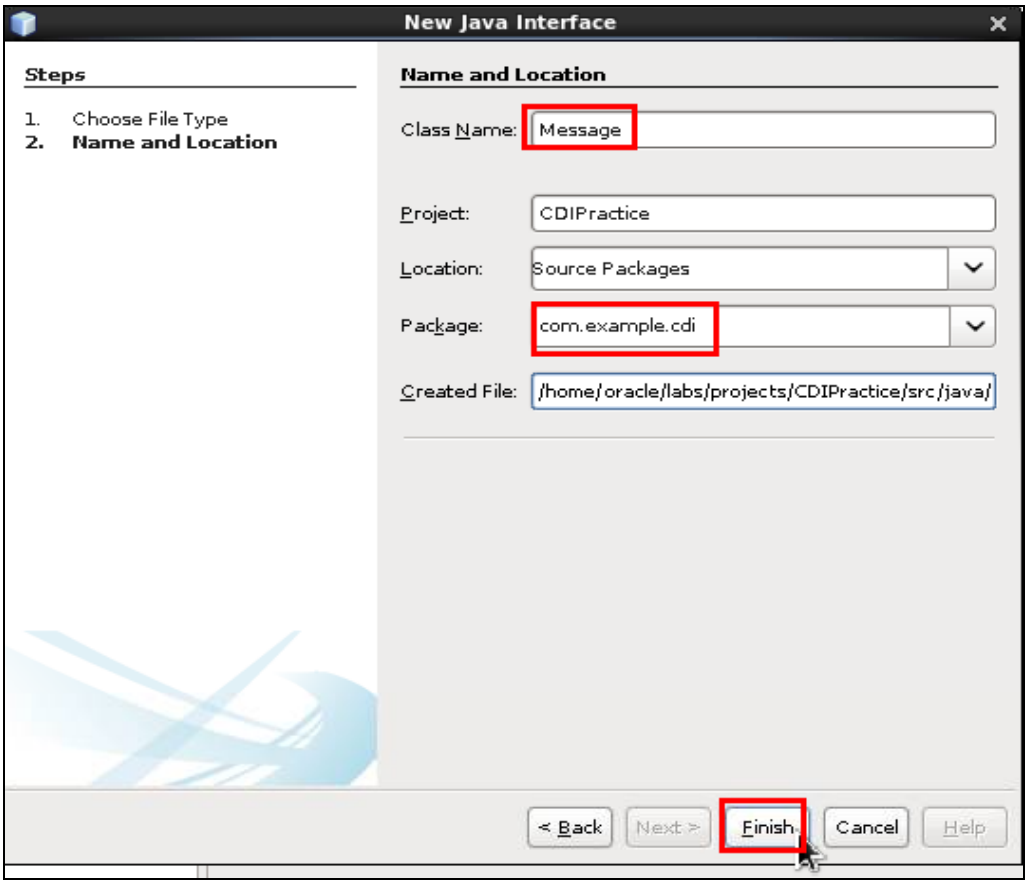
6. Create a new Java interface by performing the following steps:
   - Select **Java** from Categories.
   - Select **Java Interface** from File Types.

- Click **Next**.



7. Enter the Java interface information as follows:
   - Class Name: `Message`
   - Package: `com.example.cdi`

- Click **Finish**.



The `Message.java` file in the editor area shows the source code of the new interface:

```
public interface Message {

}
```

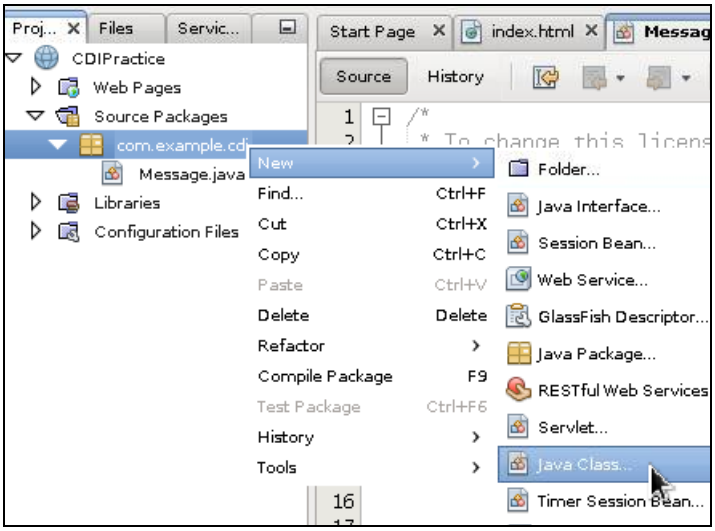
8. Add the following line of code to the Message interface:

```
public String get();
```

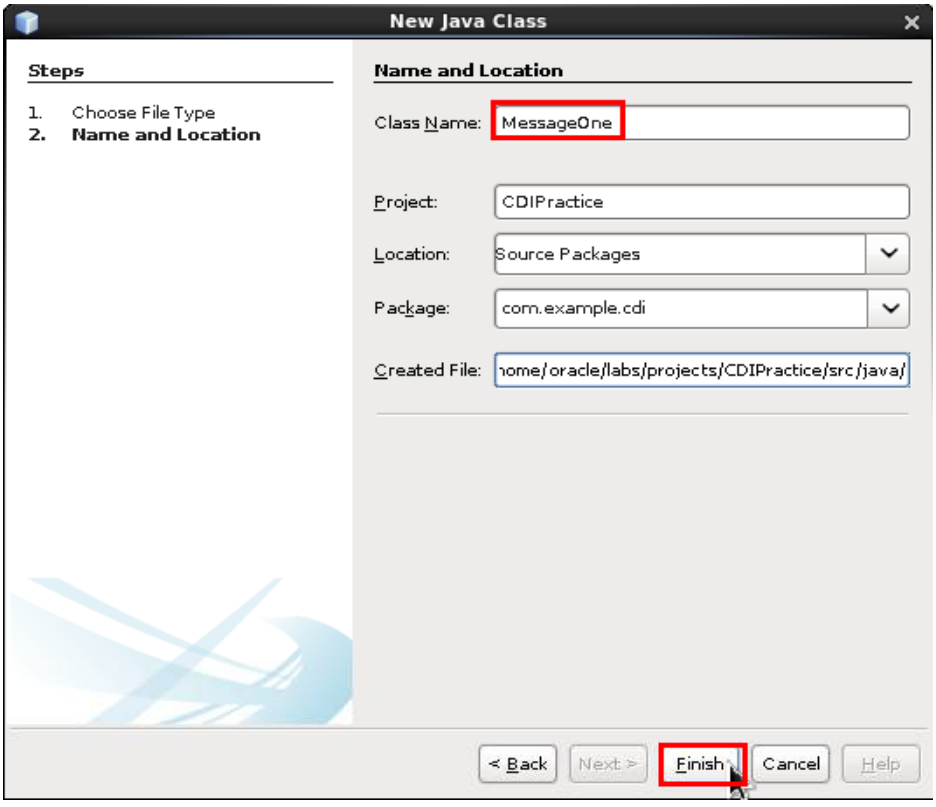
```
public interface Message {
    public String get();
}
```

Implementing the Message Interface

9. Right-click the **`com.example.cdi`** package under Source Packages. Select New > Java Class.



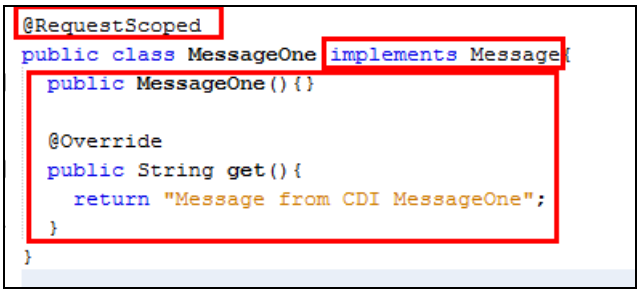
10. Enter `MessageOne` as the class name. Click **Finish**.



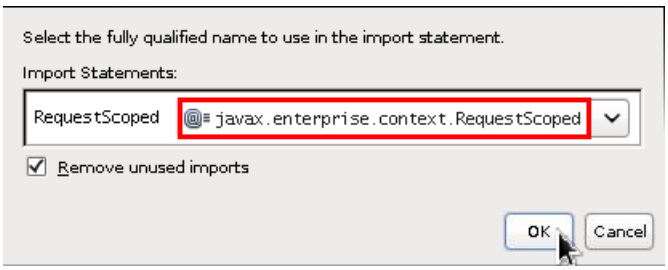
The `MessageOne.java` file in the editor area shows the source code of the new Java class.

11. Add the following lines of code (in bold) to `MessageOne.java`:

```
@RequestScoped
public class MessageOne implements Message{
  public MessageOne(){}

  @Override
  public String get(){
    return "Message from CDI MessageOne";
  }
}
```



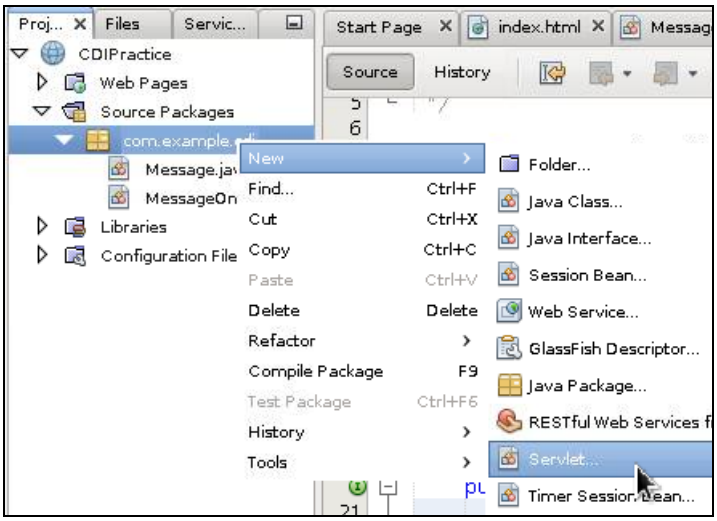
12. Press **Ctrl + Shift + I** to import the missing classes. Make sure that you select the right class when you are prompted for RequestScoped:



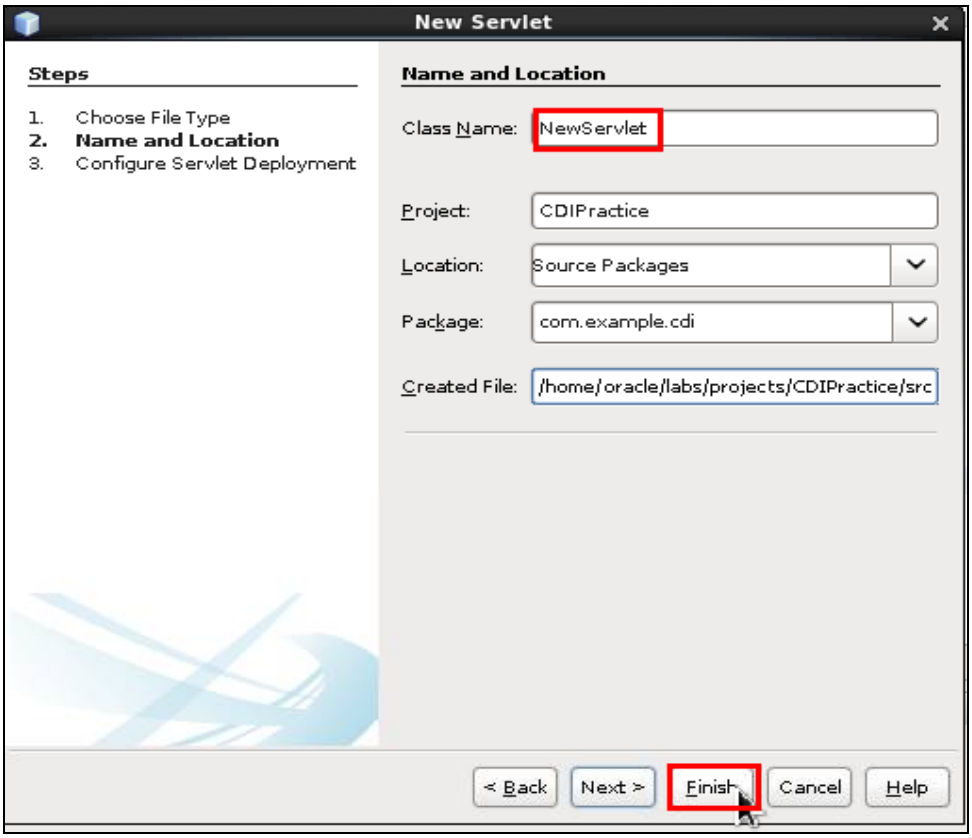
Creating the Servlet

In this section, you create a servlet where you inject the bean.

13. Right-click the **com.example.cdi** package under Source Packages. Select New > Servlet.



14. Keep the default class name. Click **Finish**.



The `NewServlet.java` file in the editor area shows the source code of the new servlet.

15. Add the following lines of code to the servlet:

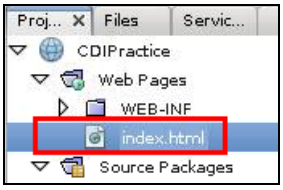
```
@Inject private Message myMessage;

out.println("<p>Hello!!!</p>");
out.println("<p>" + myMessage.get() + "</p>");
```

```
public class NewServlet extends HttpServlet {

    @Inject private Message myMessage;
    /**
     * Processes requests for both HTTP <code>GET</code> and <code>POST</code>
     * methods.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request, HttpServletRespons
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet NewServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet NewServlet at " + request.getContextPath()
            out.println("<p>Hello!!!</p>");
            out.println("<p>" + myMessage.get() + "</p>");
            out.println("</body>");
            out.println("</html>");
        }
    }
```

16. Press **Ctrl + Shift + I** to import the missing classes and save your changes.

Updating the `index.html` File

17. Double-click the `index.html` file in the Web Pages folder:



The `index.html` file in the editor area shows the source code of the index page.

18. Replace the `<div>TODO write content</div>` line with the following line of code:

```
<p><a href="NewServlet">CDI Servlet</a></p>
```

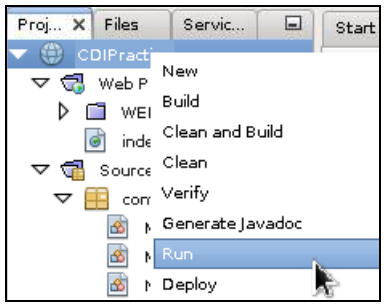
```
<html>
    <head>
        <title>TODO supply a title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,
    </head>
    <body>
        <p><a href="NewServlet">CDI Servlet</a></p>
    </body>
</html>
```

19. Save your changes.

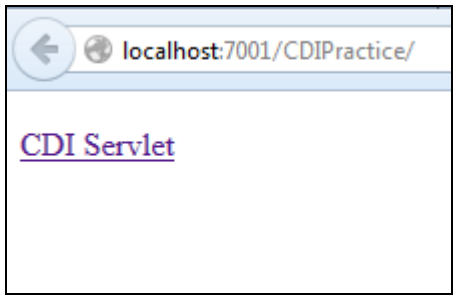
Running the Project

To deploy and test the web application, perform the following steps in the NetBeans IDE:

20. Right-click the **CDIPractice** project in the Projects window and select **Run**.



The index page opens in the browser.



21. Click the `CDI Servlet` link.

You can see the message “Message from CDI MessageOne”:

Practice 8-2: Using Qualifiers

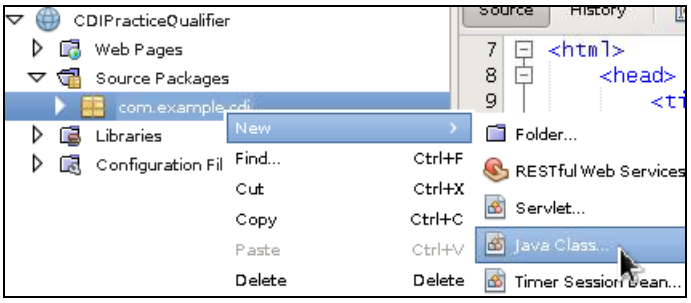
Overview

In this practice, you create a second Java class that implements the `Message` interface. Then, by using a qualifier, you specify which of the two classes should be injected into the field.

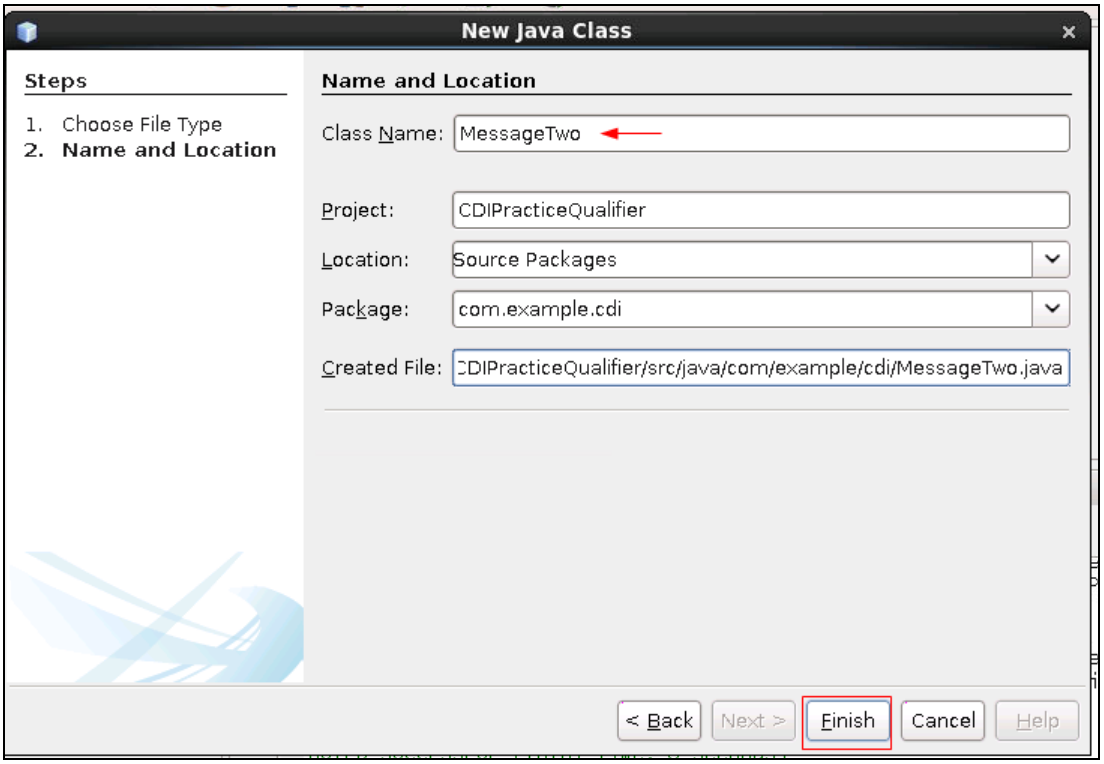
Tasks

Implementing a Second Message Interface

1. In NetBeans, open the `CDIPracticeQualifier` project in the `/home/oracle/labs/projects/08_CDI` directory.
2. Right-click the **`com.example.cdi`** package under Source Packages of the `CDIPracticeQualifier` project. Select New > Java Class.



3. Enter `MessageTwo` as Class Name. Click **Finish**.



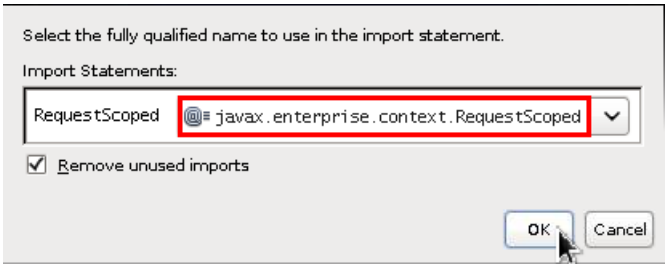
The `MessageTwo.java` file in the editor area shows the source code of the new Java class.

4. Add the following lines of code (in bold) to `MessageTwo.java`:

```
@RequestScoped
public class MessageTwo implements Message{
  public MessageTwo(){}

  @Override
  public String get(){
    return "Message from CDI MessageTwo";
  }
}
```

```
@RequestScoped
public class MessageTwo implements Message {
    public MessageTwo(){}

    @Override
    public String get() {
        return "Message from CDI MessageTwo";
    }
}
```

5. Press **Ctrl + Shift + I** to import the missing classes. Make sure that you select the right class when you are prompted for RequestScoped:

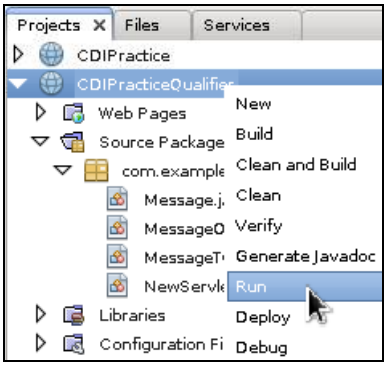


6. Save your work.

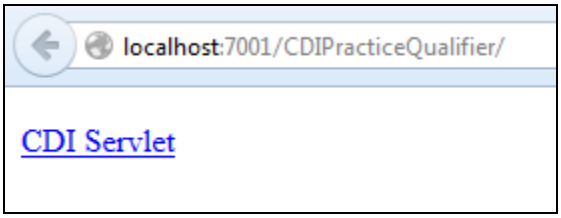
Running the Project to see the Ambiguous Dependency Error

At this point, if you run the servlet, you get a deploy-time error because both `MessageOne` and `MessageTwo` are implementing the `Message` interface and both have the `@Default` qualifier.

7. Right-click the **`CDIPracticeQualifier`** project in the Projects window and select **Run**.

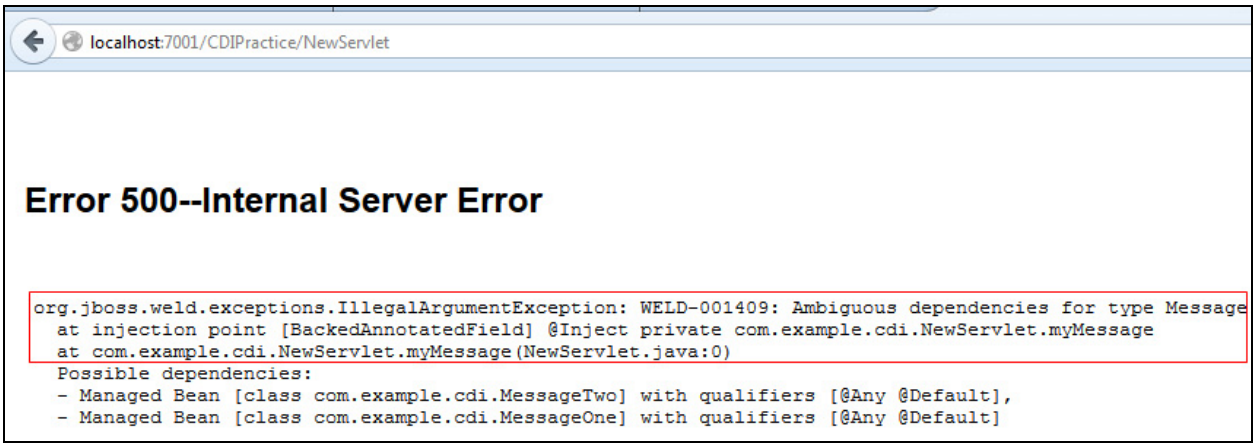


The index page opens in the browser:



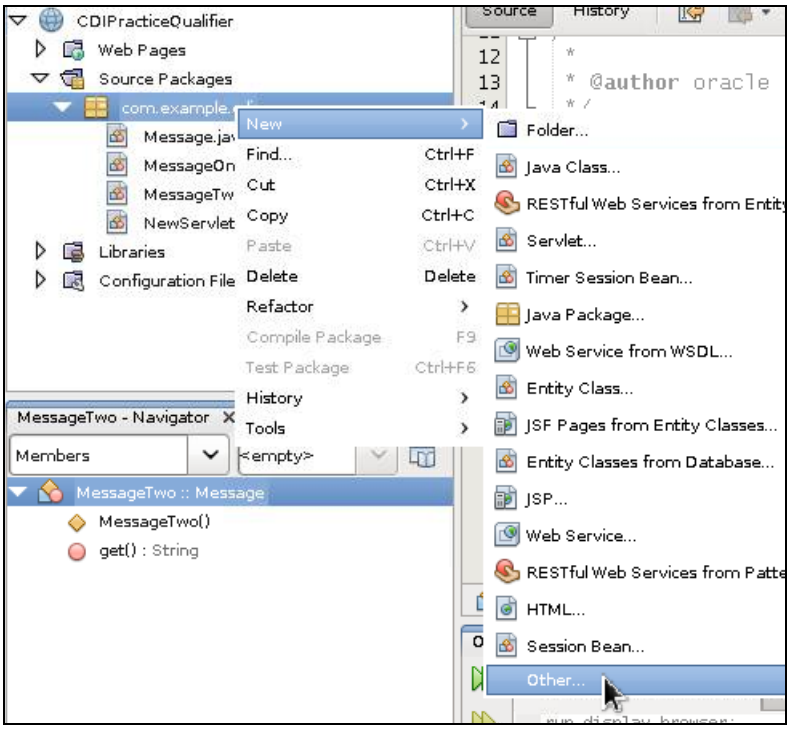
8. Click the `CDI Servlet` link.

   You can see the runtime error message due to the ambiguous dependencies for the `Message` interface:

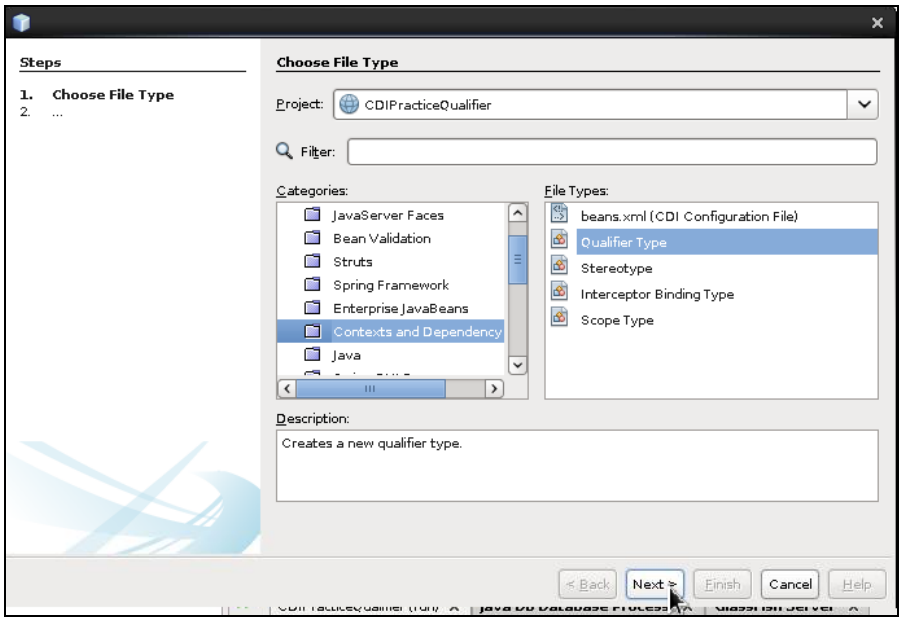


Creating a Qualifier

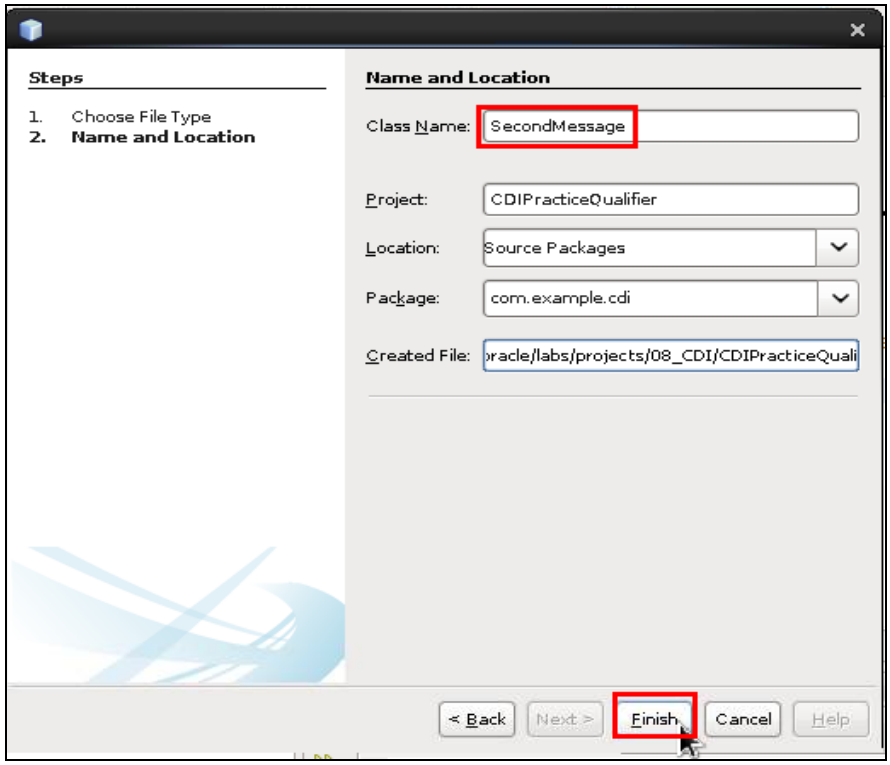
9. Right-click the **`com.example.cdi`** package under Source Packages of the `CDIPracticeQualifier` project. Select New > Other.



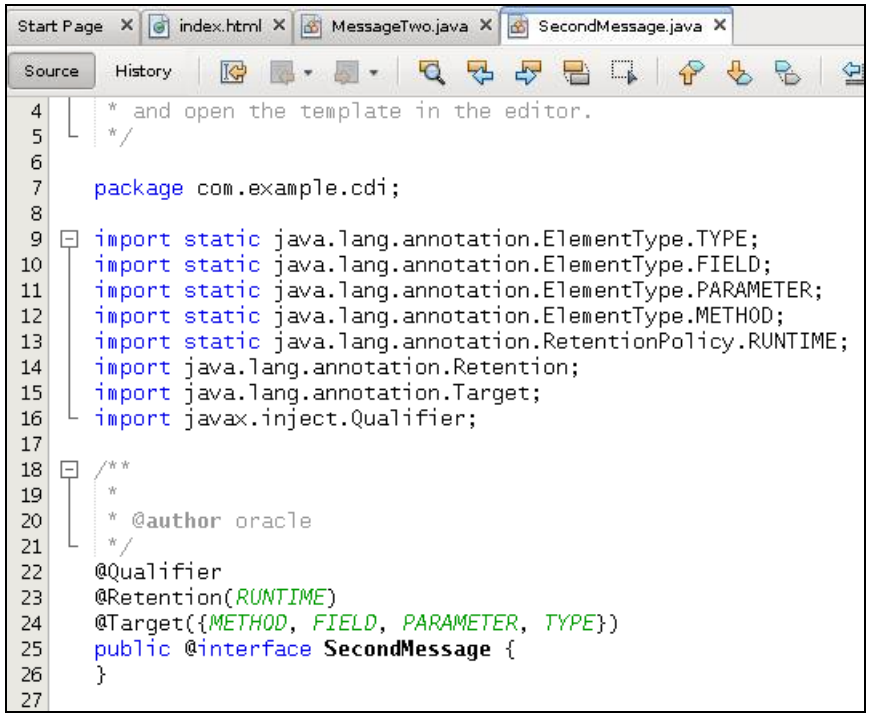
10. Create a new qualifier by performing the following steps:
    - Select **Contexts and Dependency Injection** from Categories.
    - Select **Qualifier Type** from File Types.
    - Click **Next**.



11. Enter `SecondMessage` as Class Name. Click **Finish**.



The `SecondMessage.java` file in the editor area shows the source code of the new qualifier:



Updating the Injection Point

12. Double-click the `MessageTwo` class in the Projects window.
13. Add the following annotation to the `MessageTwo` class:

    `@SecondMessage`

```
@RequestScoped @SecondMessage
public class MessageTwo implements Message {
    public MessageTwo(){}

    @Override
    public String get() {
        return "Message from CDI MessageTwo";
    }
}
```

14. Double-click the `NewServlet` servlet in the Projects window.
15. Add the following annotation to the `NewServlet` class:

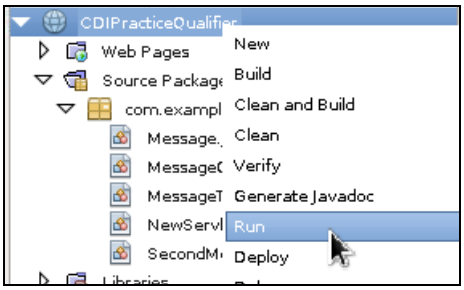
    `@SecondMessage`

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

    @Inject @SecondMessage private Message myMessage;
    /**
     * Processes requests for both HTTP <code>GET</code> and <code>
     * methods.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
```

Running the Project

To deploy and test the web application, perform the following steps in the NetBeans IDE:

16. Right-click the **CDIPracticeQualifier** project in the Projects window and select **Run**.



    The index page opens in the browser.

17. Click the `CDI Servlet` link.

    You can now see the message “Message from CDI MessageTwo” from `MessageTwo` displayed on the page:

Practices for Lesson 9: Developing Java EE Applications Using Messaging

Chapter 9

Practices for Lesson 9: Overview

Practices Overview

In these practices, you use the JMS API to create and send a message to queues.

Practice 9-1: Creating a JMS Queue by Using WebLogic Server

Overview

In this practice, you create the necessary JMS connections and destinations that are required to build a JMS solution. You use the NetBeans IDE to create the necessary server resources. Specifically, you create a JMS destination, `orderQueue`, in the WebLogic Server.

Every JMS application requires a connection factory (an object that implements `javax.jms.ConnectionFactory`) and at least one destination (an object that implements either `javax.jms.Queue` or `javax.jms.Topic`).

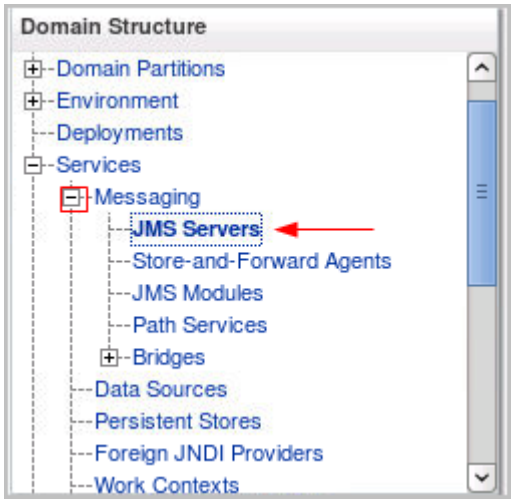
In JMS, `ConnectionFactory` is the object that is used to create the connection to the JMS provider and `Queue` or `Topic` is the object that identifies the physical queue or topic that messages are being sent to or received from. How these objects are configured varies from one JMS provider to another.

Assumptions

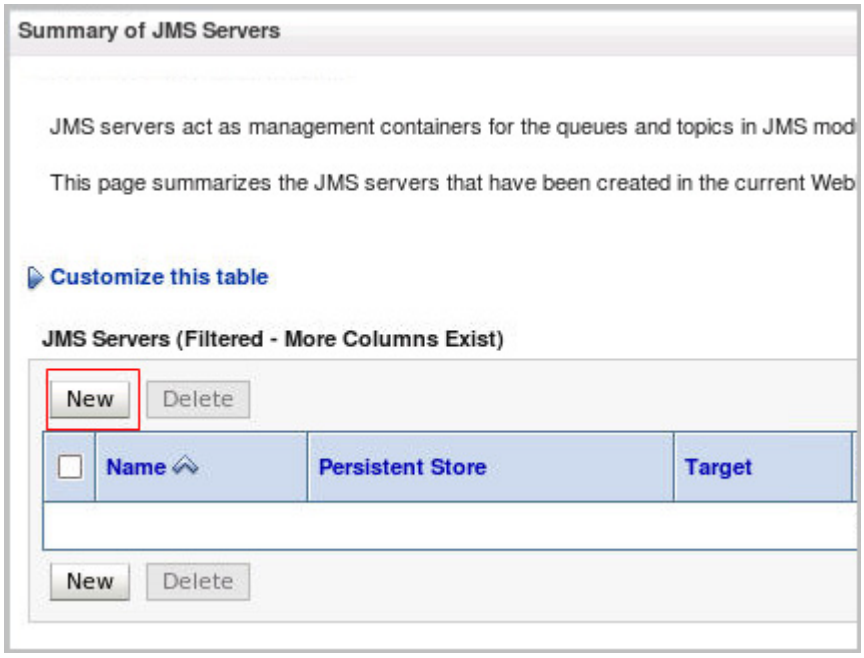
WebLogic Server is configured in NetBeans and is running.

Tasks

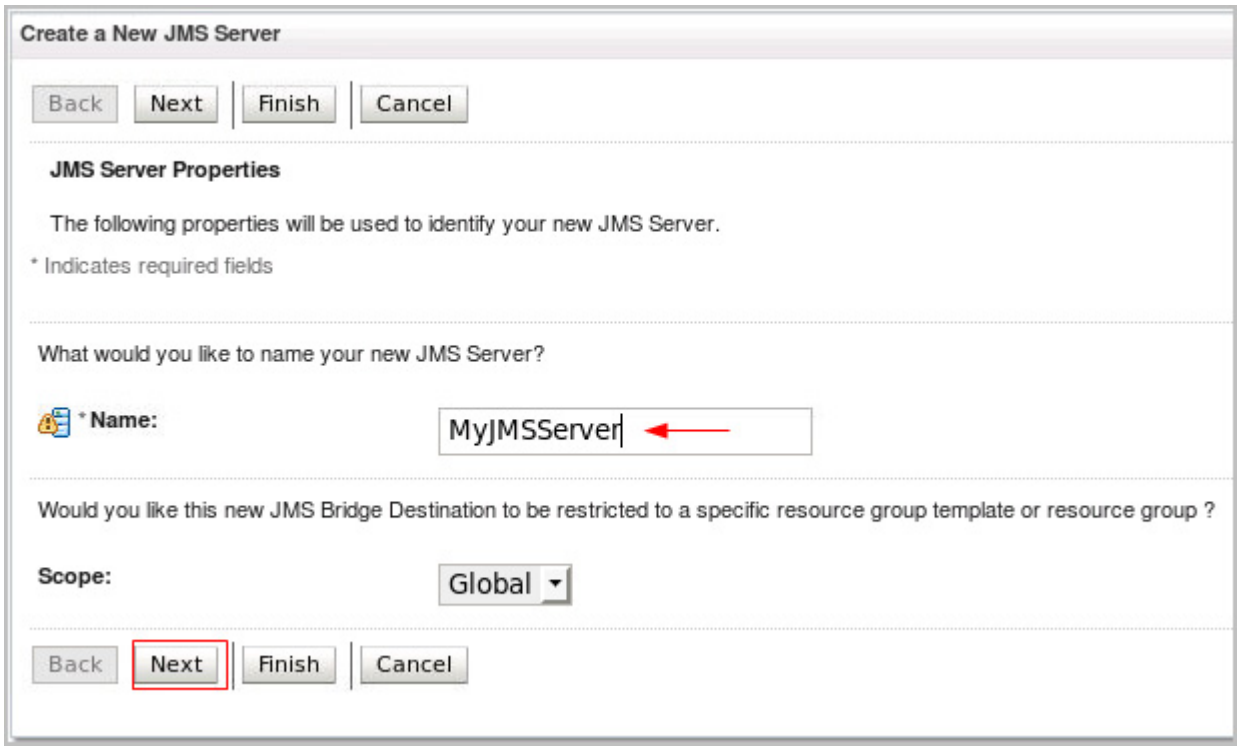
1. Create a JMS server using the WebLogic Server Admin Console.
   a. A JMS server is the container that manages JMS queue and topic destinations. A JMS server can be configured to persist messages, so they can be delivered even if the server instance they were received at went down.
   b. Launch the Admin Console by right-clicking **Oracle WebLogic Server** in Services and selecting **View Admin Console**.
   c. Enter the username and password and click Login.
   d. In the **Domain Structure** panel on the left, expand **Services**, expand **Messaging**, and then select **JMS Servers.**



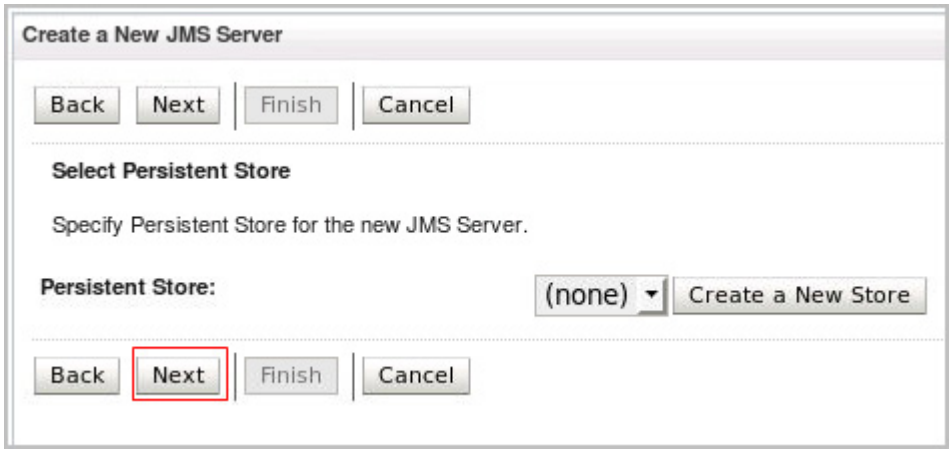
   e. Click **New** to create a new JMS server.



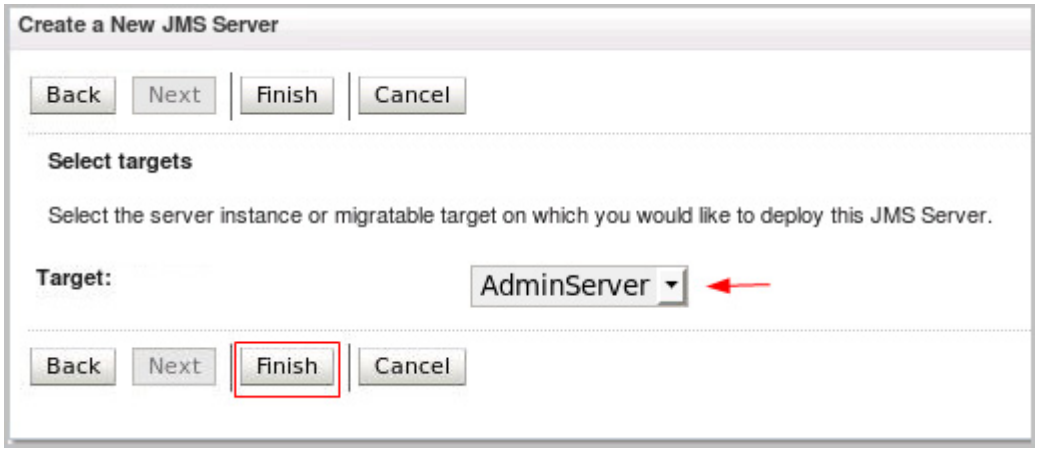
f. Enter **`MyJMSServer`** as the server name and click **Next**.



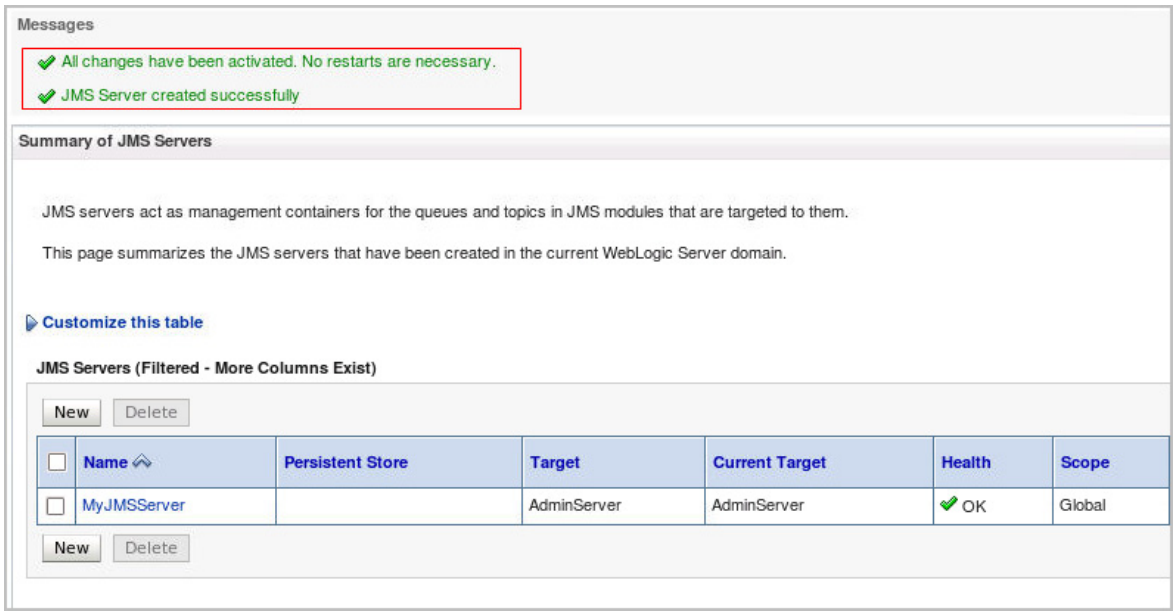
g. Click **Next** on the **Create a New JMS Server** page.



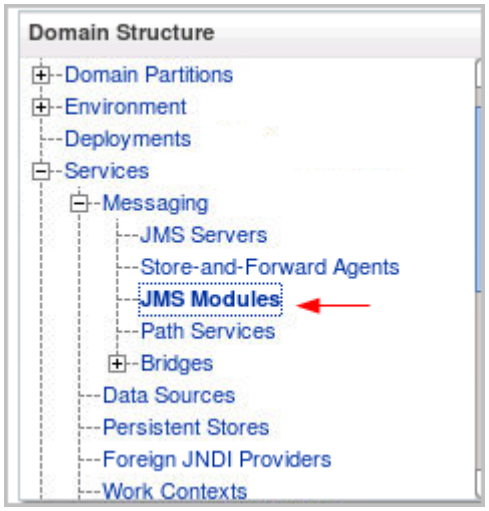
h. Select **AdminServer** as the target for the JMS server and click **Finish**.



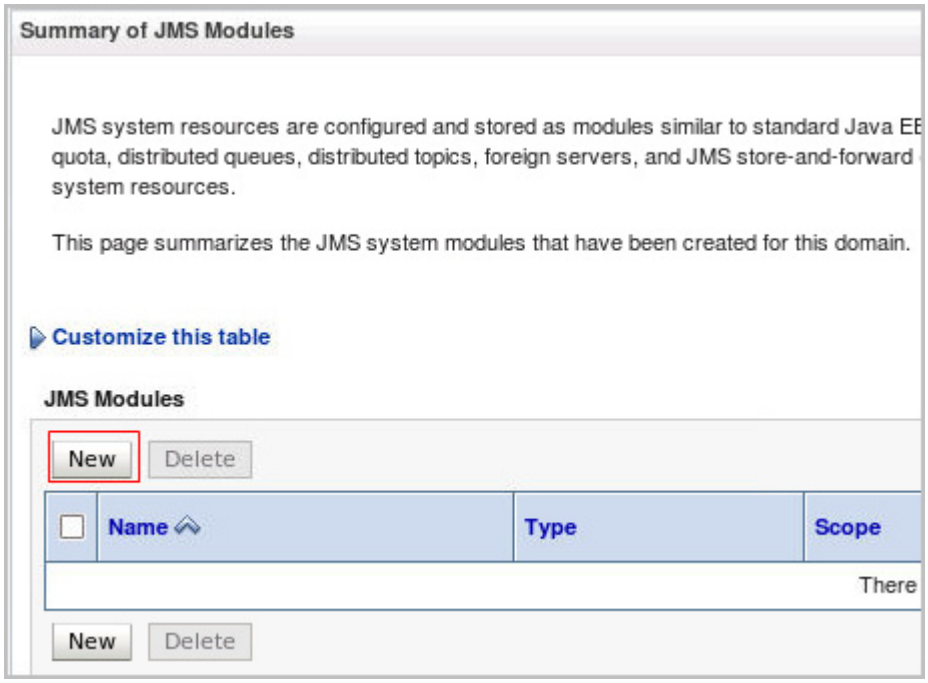
i. You will see a message at the top of the page, indicating that the JMS server has been successfully created. This server targets AdminServer, the WLS instance you have running.



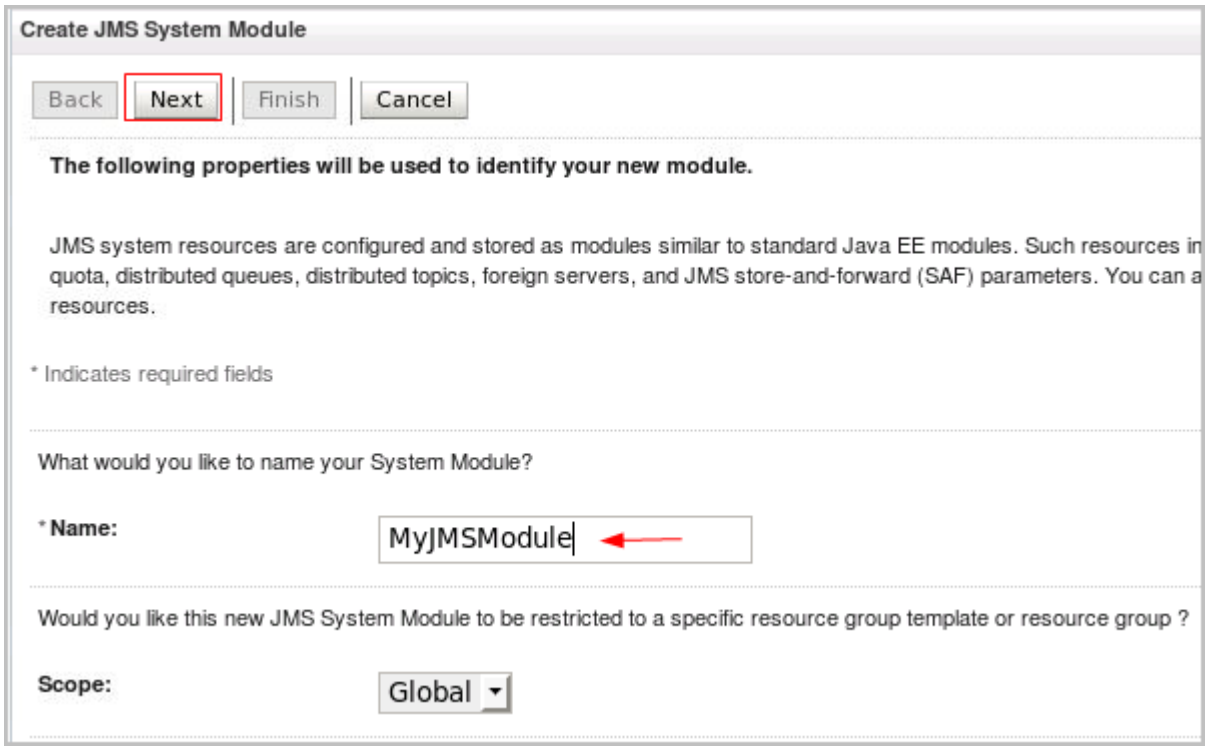
2. Next you will create a **JMS System Module** that contains your queue and connection factory.
   a. In the **Domain Structure** panel on the left, click **JMS Modules** under **Messaging**.



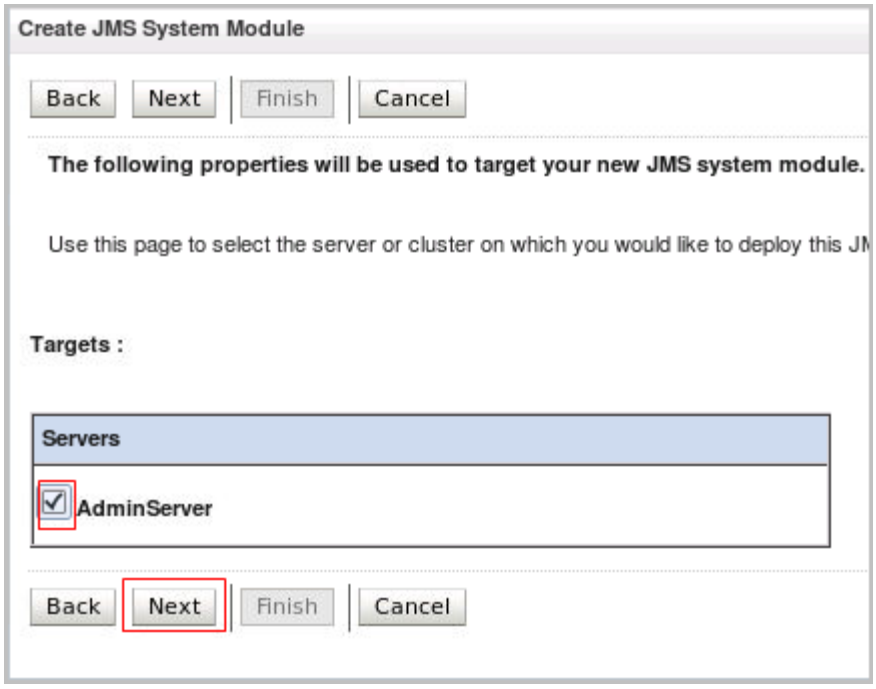
   b. Click **New** to create a new JMS System Module.



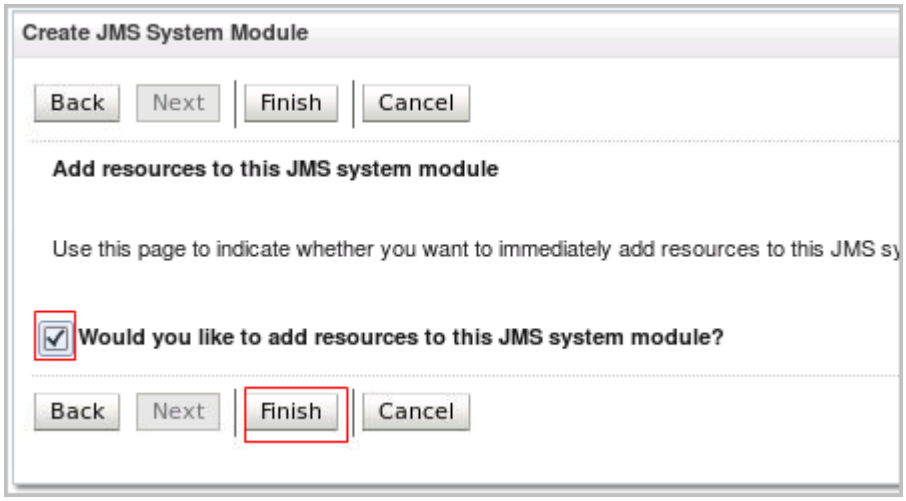
c. Enter the name of the JMS System Module as **`MyJMSModule`** and click **Next**.



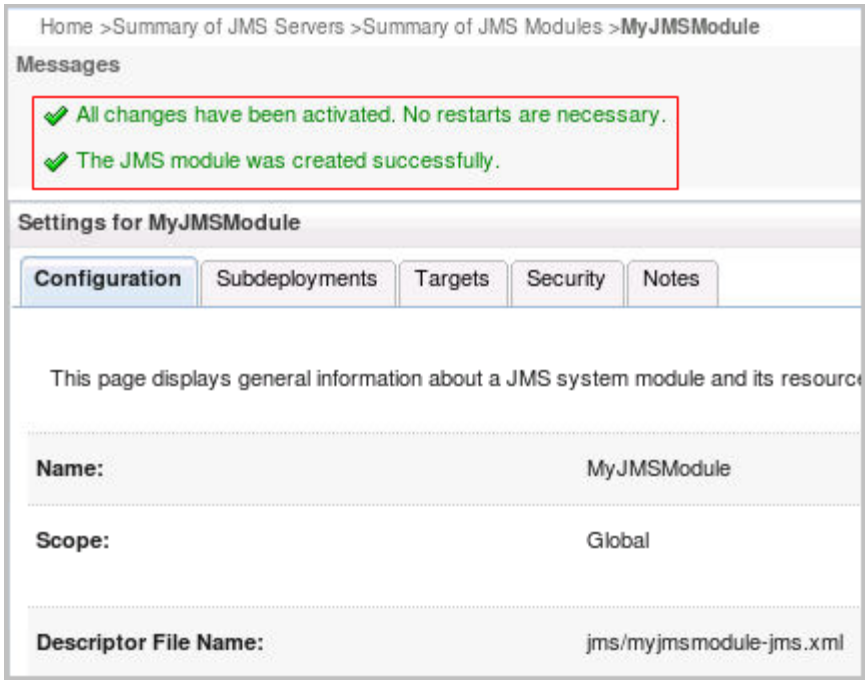
d. Select **AdminServer** to choose your current WLS instance as the target for this module and click **Next**.



e. Select the check box to add resources to the new JMS System Module (you will add the queue and connection factory) and click **Finish**.

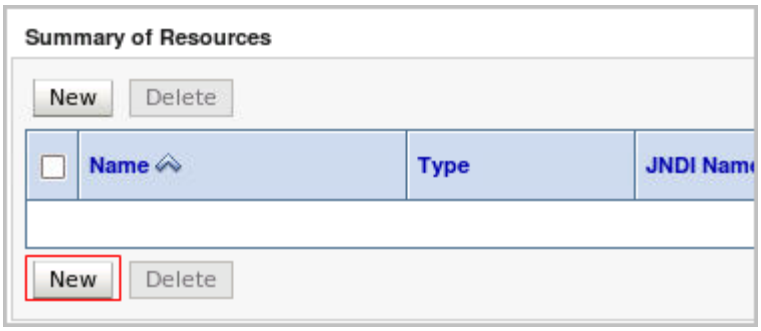


f. At the top of the page, messages will appear, indicating that you successfully created the JMS System Module.

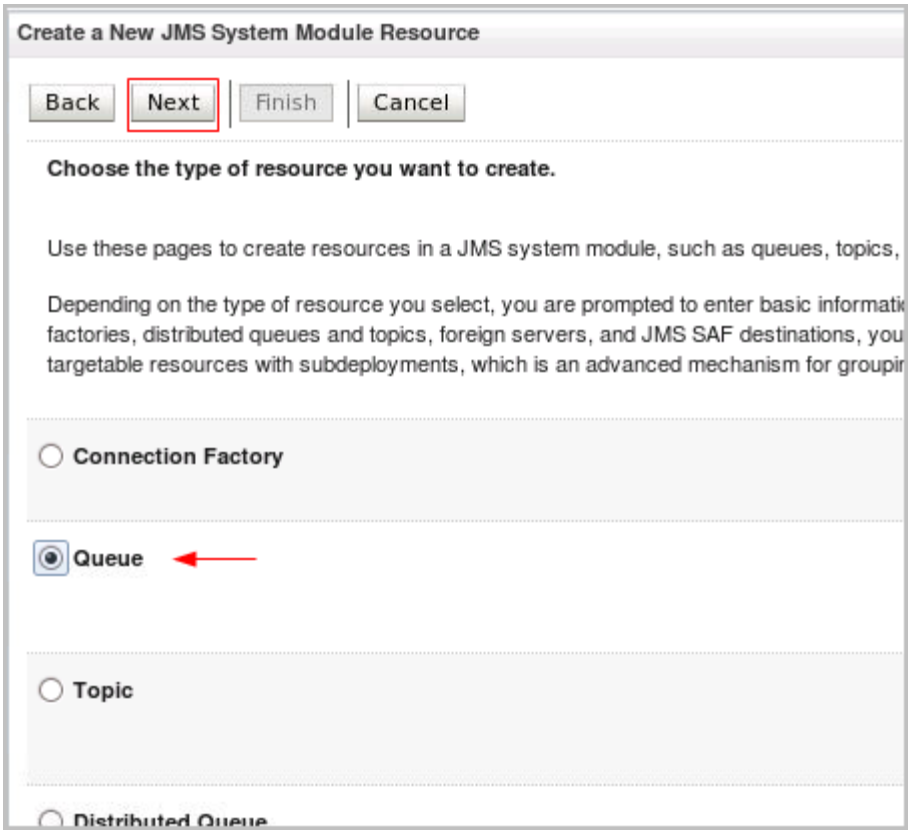


3. Create a Queue Destination.

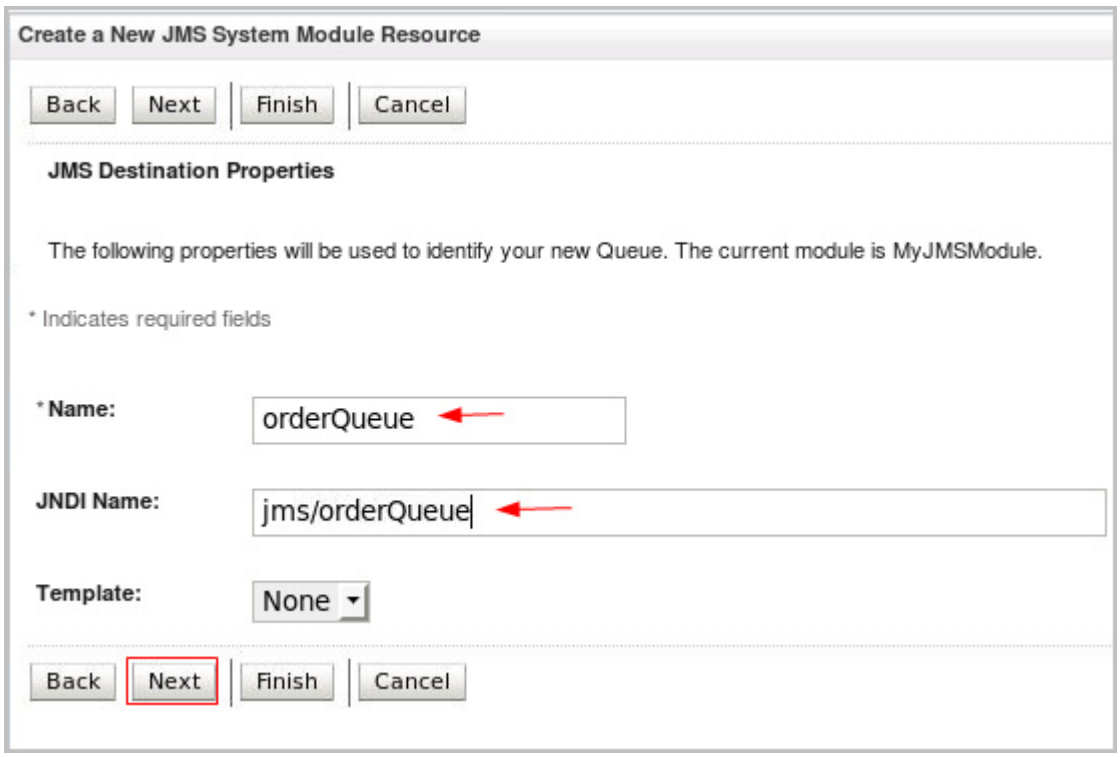
   a. Click **New** to add a new resource to the system module.



   b. Create a new Queue destination by selecting **Queue** from the options. Click **Next**.

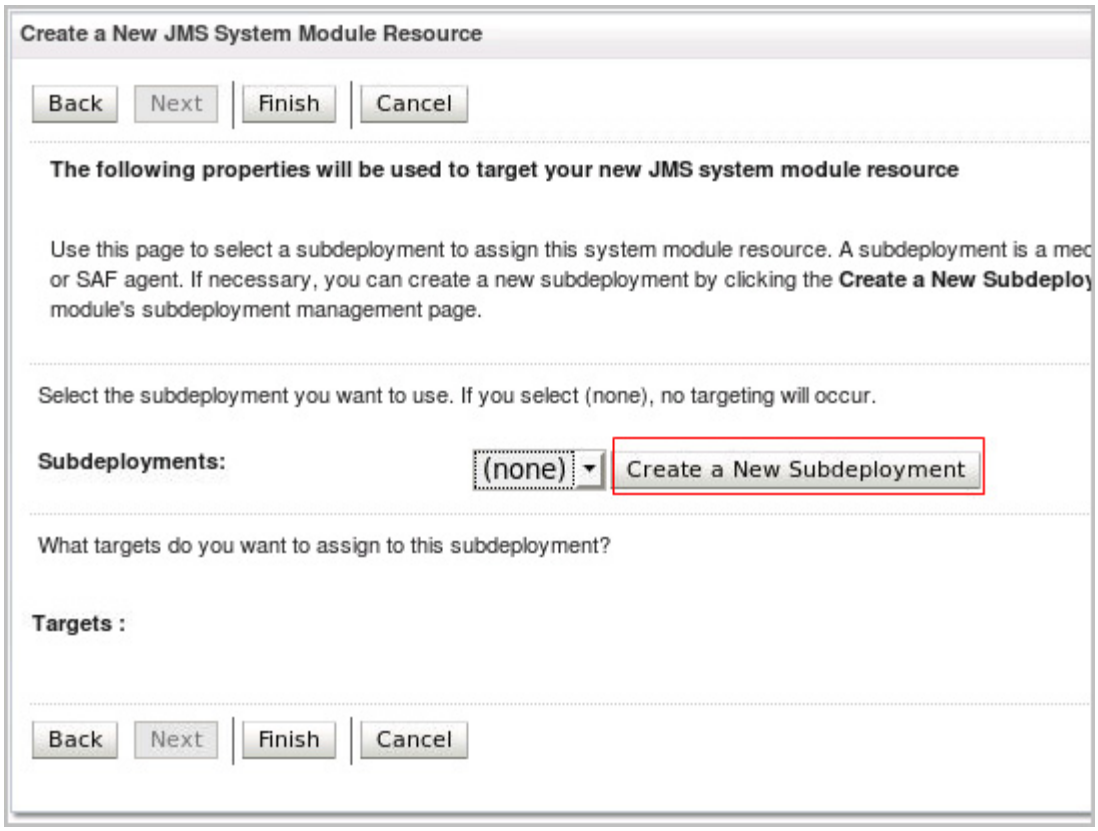


c. Enter **`orderQueue`** as the Name for your queue and **`jms/orderQueue`** as the JNDI Name for the lookup. Click **Next**.

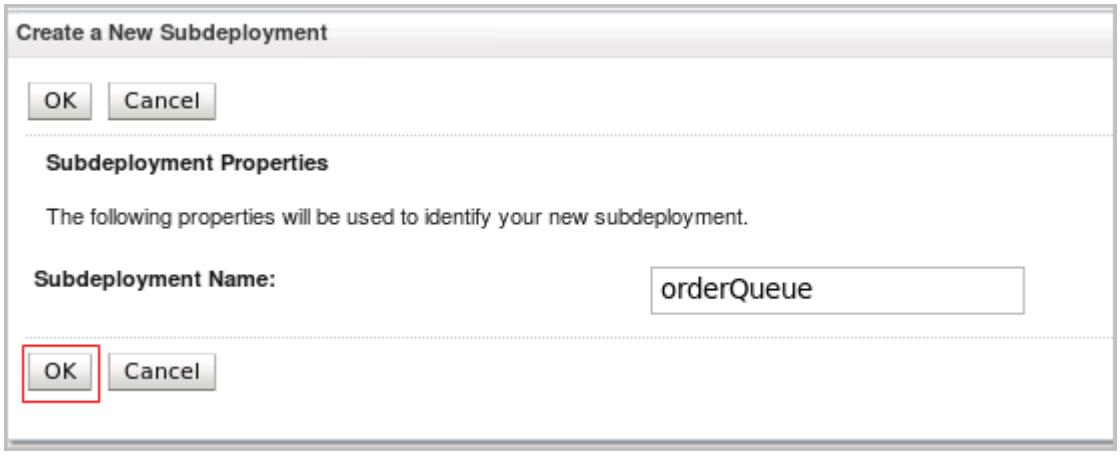


d. On the next screen, click **Create a New Subdeployment** to create a subdeployment for the system module.

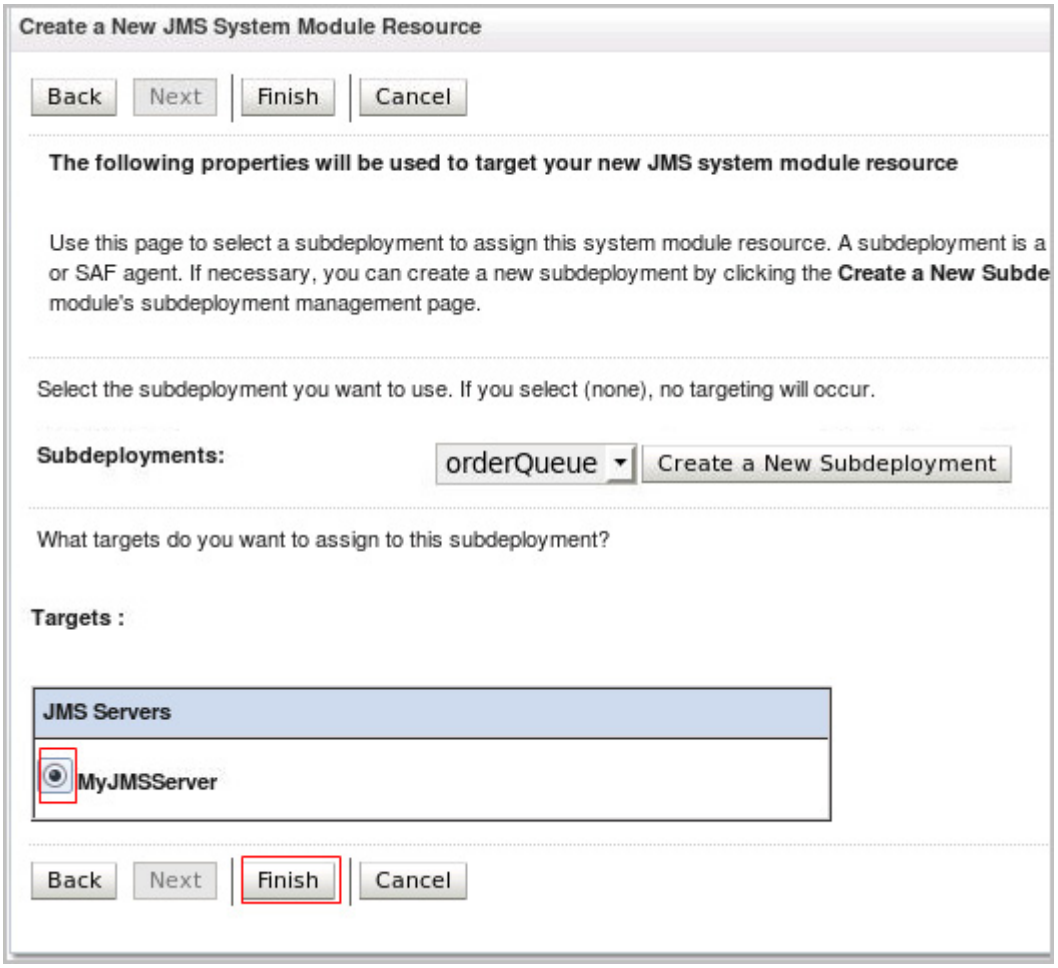
A subdeployment is the mechanism by which targetable JMS module resources (such as queues, topics, and connection factories) are grouped and targeted to a server resource (such as JMS servers, server instances, or a cluster).



e. On the next screen, click **OK** to accept the default Subdeployment Name.

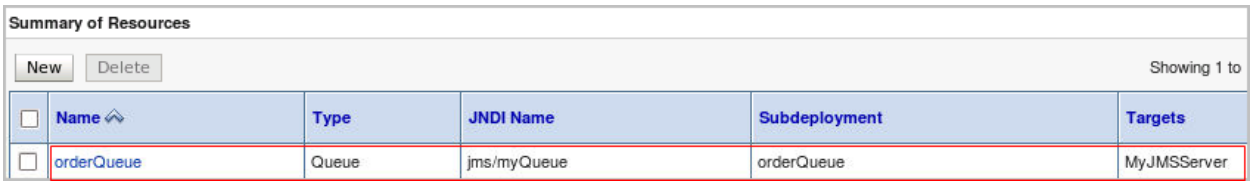


f. With the new subdeployment now selected, select the **MyJMSServer** module as the target, and click **Finish**.



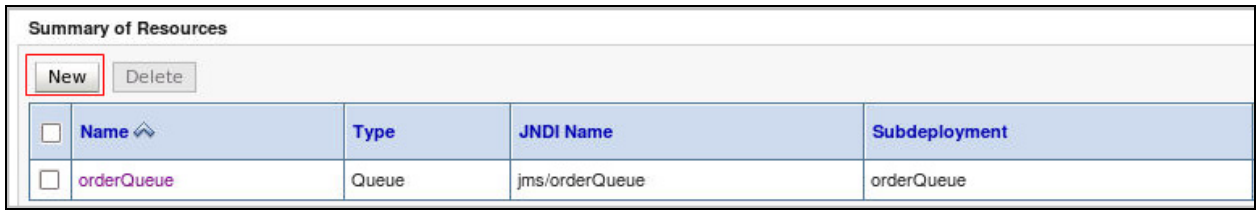
g. A message will appear at the top of the page, indicating that you successfully created the queue destination.

h. Near the bottom of the page, the **Summary of Resources** section will show your new queue resource.

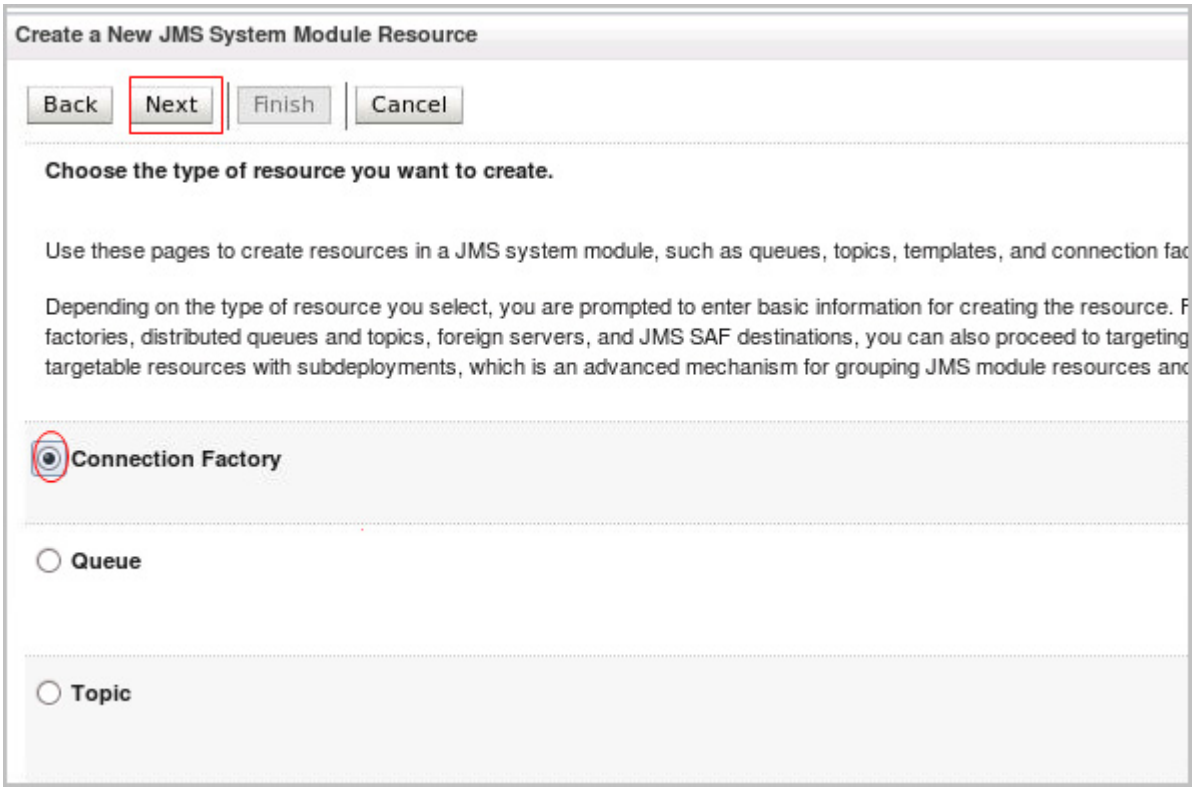


4. Create the Connection Factory.

   a. Click **New** to add another resource to the JMS Server Module.

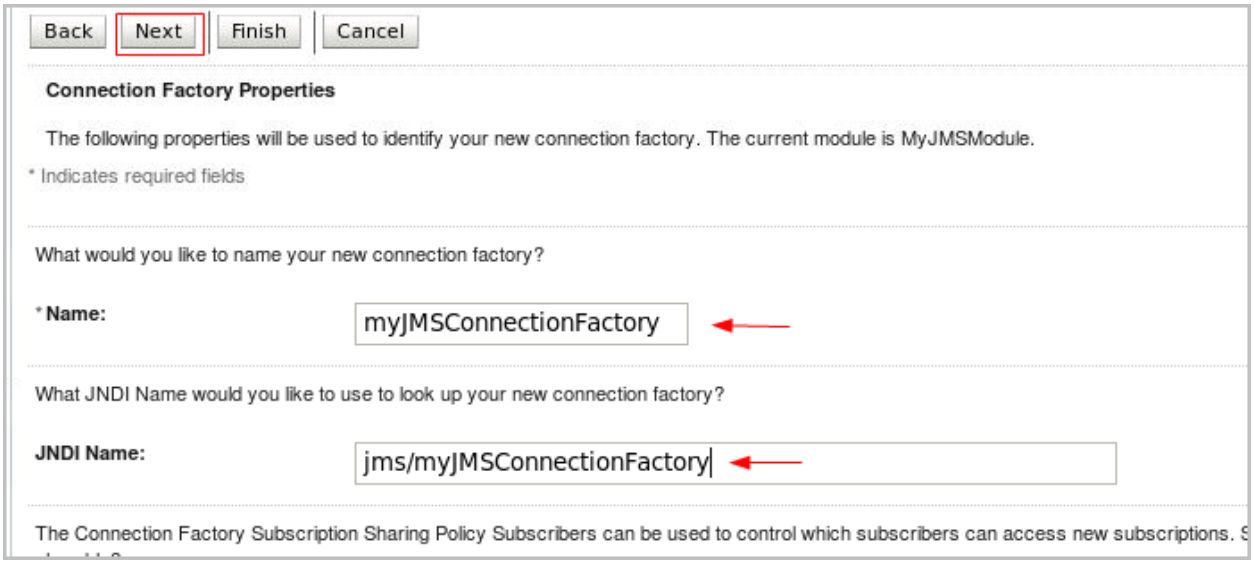


   b. Create a Connection Factory by selecting **Connection Factory** from the list of resources and click **Next**.

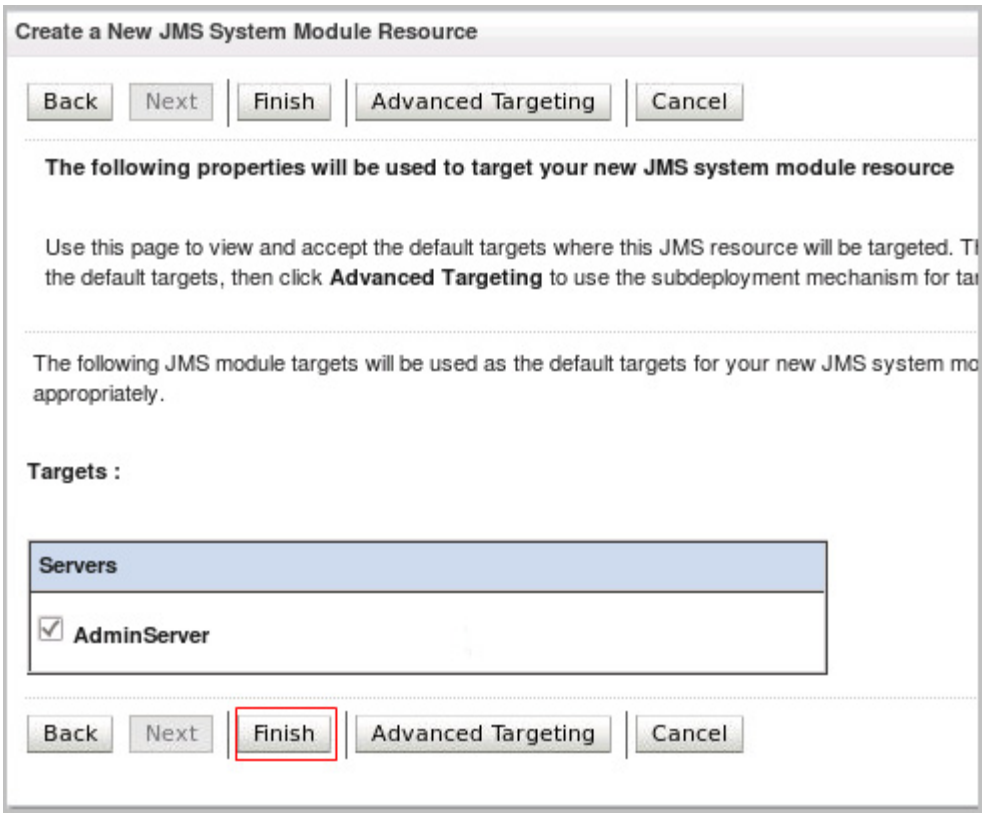


   c. Enter `myJMSConnectionFactory` as the name of your Connection Factory.

   d. Enter `jms/myJMSConnectionFactory` as the JNDI name for the factory, accept the other options, and click **Next**.



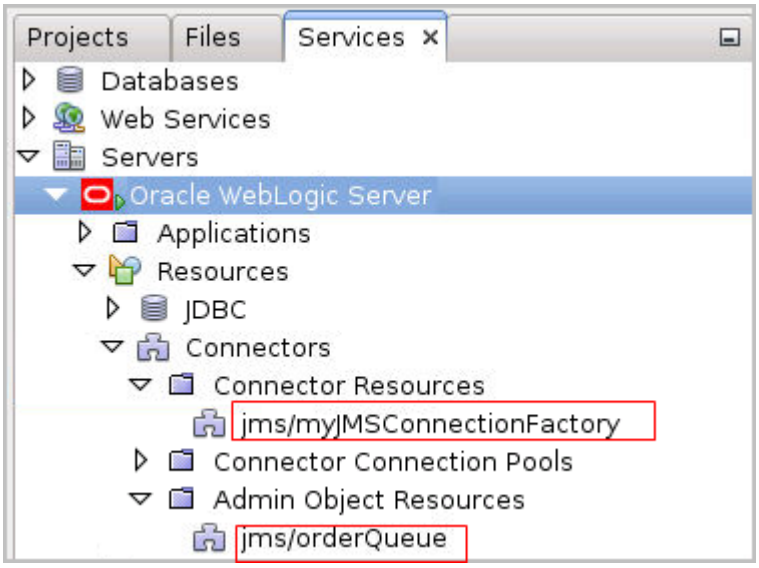
e. Your JMS Module is automatically associated with the WLS instance AdminServer. Click **Finish**.



f. At the top of the page, a message will indicate that you have successfully created the connection factory.

g. The Connection Factory and Queue will appear as resources in the **Summary of Resources** section for the module.

5. Look at the JMS resources through NetBeans.
   a. Open NetBeans and click the **Services** tab.
   b. Expand **Servers** > **Oracle WebLogic Server**.
   c. Expand **Resources > Connectors**.
   d. Expand the **Connector Resources** and the **Admin Object Resources** folders.



Note: If you cannot see any resources for your Oracle WebLogic Server server instance and you are using NetBeans 8.1, first make sure that Oracle WebLogic Server is running. Second, check the proxy setting for NetBeans. Open the Options menu (**Tools > Options**) and change the Proxy Settings from Use System Proxy Settings to No Proxy. Then stop Oracle WebLogic Server, quit NetBeans, and restart NetBeans and Oracle WebLogic Server.

Practice 9-2: Creating a Web-Based JMS Queue Producer and Consumer

Overview

In this practice, you create a Java EE web application. The application contains two request-scoped CDI beans. One CDI bean is a JMS producer and the second CDI bean is a JMS consumer.

Assumptions

Practice 9-1 is completed.

Tasks

1. In NetBeans, open the `OrderMgmtApp` project from `/home/oracle/labs/projects/09_JMS`.
2. Expand the project directories and observe the following source files:
   - `index.html`: This file defines the UI components for this application. Observe that the file contains a `submit` button, which invokes the `OrderServlet` when the form is submitted.
   - `OrderProducerServlet.java` in the `com.example` package: This servlet parses the web form, creates a message, and sends that message to the `orderQueue` through the default connection factory.
3. Develop a JMS producer, `OrderProducerBean.java`, which is a request-scoped CDI bean. It contains a single method, `sendMessage`.
   - Right-click the `com.example` package in the `OrderMgmtApp` project and select New > Other.
   - Select Java from Categories and Java Class from File Types, and click Next.
   - Enter `OrderProducerBean` as Class Name and click Finish.
   - Annotate the class by using the `@RequestScoped` annotation.

```
@RequestScoped
public class OrderProducerBean {
```

   - Add a no-arg constructor.

```
public OrderProducerBean() {
}
```

Tip: In NetBeans, you can quickly add constructors by right-clicking in the editor window and selecting Insert Code (or by pressing Alt + Insert), and selecting Constructor from the Generate menu.

   - Above the constructor, inject a `JMSContext` instance and resource by using the `orderQueue` that you created.

```
@Inject
private JMSContext context;
@Resource(lookup = "jms/orderQueue")
private Queue queue;
```

- Add a method, `sendMessage`, which takes a String argument, `message`, and returns void. This method uses the injected context to create the JMS producer and send the message parameter to the queue.

```
public void sendMessage(String message) {
  context.createProducer().send(queue, message);
}
```

- Fix any missing import statements and save the class.

Tip: In NetBeans, you can quickly add missing import statements by right-clicking in the editor window and selecting Fix Imports or by pressing the Ctrl + Shift + I keys.

4. Develop a JMS consumer, `OrderReceiverBean.java`, which is a request-scoped CDI bean. It has a single method, `receiveMessage`.
   - Right-click the `com.example` package in the `OrderMgmtApp` project and select New > Other.
   - Select Java from Categories and Java Class from File Types, and click Next.
   - Enter `OrderReceiverBean` as Class Name and click Finish.
   - Annotate the class by using the `@RequestScoped` annotation.

```
@RequestScoped
public class OrderReceiverBean {
```

   - Add a logger to the class.

```
private static final Logger logger =
Logger.getLogger("com.example.OrderReceiverBean");
```

   - Add a no-arg constructor.

```
public OrderReceiverBean() {
}
```

   - Above the constructor, inject a `JMSContext` instance and resource by using the `orderQueue` that you created.

```
@Inject
private JMSContext context;
@Resource(lookup = "jms/orderQueue")
private Queue queue;
```

   - Add a `receiveMessage` method that takes no arguments and returns void.

```
public void receiveMessage() {

}
```

   - In the method body, use a try-with-resources statement to create a `JMSConsumer` and read a message from the consumer with a 1000 millisecond (1 second) timeout. If the message is not null, log it, and if the message is null, log that the message likely timed out. Finally, log any `JMSException` that is thrown.

```
  try (JMSConsumer consumer = context.createConsumer(queue)) {
    Message m = consumer.receive(1000);
    if (m != null) {
      logger.log(Level.INFO, "Received message" +
```

```
                                m.getBody(String.class));
    } else {
      logger.log(Level.INFO, "Received no message - likely timed
out.");
    }
  } catch (JMSException ex) {
    logger.log(Level.INFO,"exception" + ex);
  }
```

- Fix any missing import statements (be sure to select `java.util.logging` for the Logger class) and save the file.

5. Modify `com.example.OrderProducerServlet` to use the two CDI beans.
   - Below the `Logger` declaration, add two injections: one for `OrderProducerBean` as `sender` and one for `OrderReceiverBean` as `receiver`.

```
private static final Logger logger =
Logger.getLogger("com.example.OrderProducerServlet");

@Inject
private OrderProducerBean sender;
@Inject
private OrderReceiverBean receiver;
```

   - In the `processRequest` method, find the first `TODO` comment and beneath it, add a call to the `sendMessage` method of the sender bean that you injected, passing in the order object.

```
//TODO: Create a sender CDI Bean to send the message
sender.sendMessage(order);
```

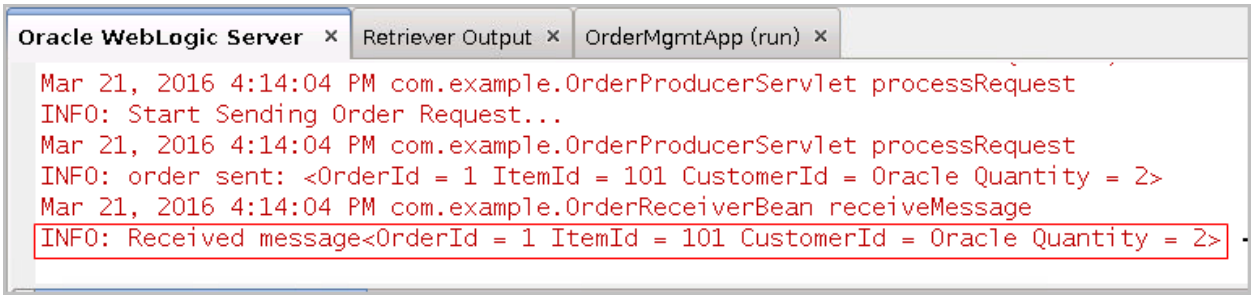
   - Find the second `TODO` comment and below it, add the call to the `receiveMessage` method of the receiver bean that you injected.

```
//TODO: Create a receiver CDI Bean and get the message
receiver.receiveMessage();
```

   - Fix any missing import statements and save the file.
6. Test your application.
   - Right-click the `OrderMgmtApp` project and select Run.
   - In the browser window that opens, enter an order record as shown, and click Order.

- The application should respond with a page that indicates that the message was sent.

- Look in the Output window in the WebLogic console to see that the message that you sent was received by `OrderReceiverBean`.



Practice 9-3: Creating an Asynchronous Java SE Client for a Queue

Overview

In this practice, you write a Java SE client to mimic a JMS application that asynchronously listens for messages from the `OrderMgmtApp` web application.

Assumptions

Practice 9-2 is completed.

Tasks

1. In NetBeans, open the `AsynchOrderConsumer` client project from `/home/oracle/labs/projects/09_JMS`.
   - Open the `com.example.AsynchOrderConsumer.java` class in the editor.
2. Add the code to the main method in the `AsynchOrderConsumer` class to use JNDI to get the JMS queue.
   - Find the first `TODO` comment in the code.
   - Add a try block. Inside the try, get the initial context and use that context to get the default connection factory. Then use the connection factory to create a `JMSContext` and use that context to look up "`jms/orderQueue`".

```
//TODO: In a try block, use JNDI to lookup the default
ConnectionFactory, create the JMSContext and lookup the
orderQueue
try {
  Context jndiContext = getInitialContext();
  ConnectionFactory connectionFactory = (ConnectionFactory)
jndiContext.lookup("jms/myJMSConnectionFactory");
  JMSContext context = connectionFactory.createContext();
  Queue queue = (Queue) jndiContext.lookup("jms/orderQueue");
```

   - Below the second `TODO` comment, create an instance of a `JMSConsumer`, create an instance of `TextListener`, and use the `consumer` instance.

```
  //TODO: Create a JMSConsumer from the context, create an
instance of TextListener and set the listener into the consumer
  JMSConsumer consumer = context.createConsumer(queue);
  MessageListener listener = new TextListener();
  consumer.setMessageListener(listener);
```

   - After the final `TODO` comment, add the catch block for the try statement.

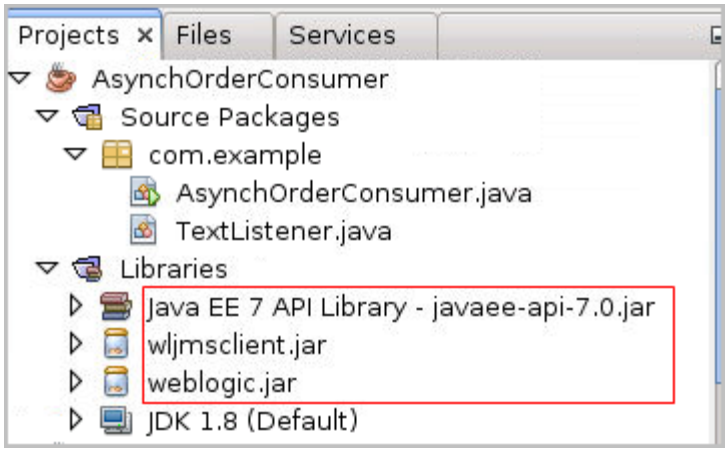
```
//TODO: add the catch block for the try block
} catch (NamingException | JMSRuntimeException e) {
  System.err.println("Exception occurred: " + e.toString());
  System.exit(1);
}
```

Make sure that the `while` loop that waits for a key entry is inside the try/catch block.

- Fix any missing import statements and save the file.
- Add the `getInitialContext` method to obtain a reference to JNDI, outside the main method.

```
private static InitialContext getInitialContext() throws
NamingException {
        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
        env.put(Context.PROVIDER_URL, "t3://localhost:7001");
        return new InitialContext(env);
    }
```

- Verify the Java EE 7 and WebLogic runtime libraries, `wljmsclient.jar` and `weblogic.jar`, added to the project. Perform the following steps:
  a. Expand the `AsynchOrderConsumer` project and then expand the `Libraries` folder.
  b. Observe the Libraries added to the project:

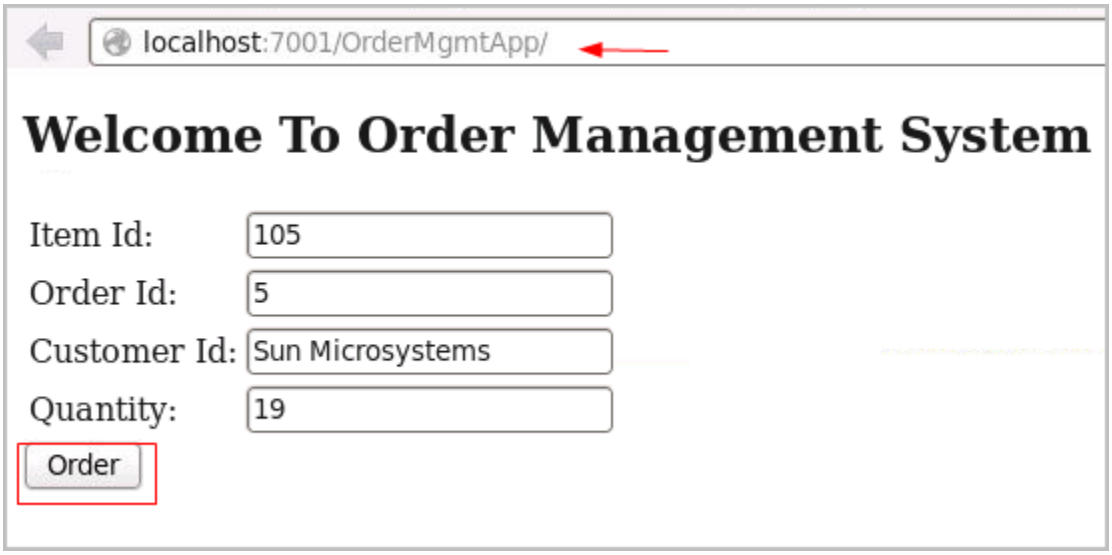


3. Open the `com.example.TextListener` class in the editor.
   - Note that this class implements `MessageListener` and implements the one required method, `onMessage`.
4. Modify `OrderMgmtApp`.
   - Open the `com.example.OrderProducerServlet` class.
   - Find the call to `receiver.receiveMessage()` and comment the line out.

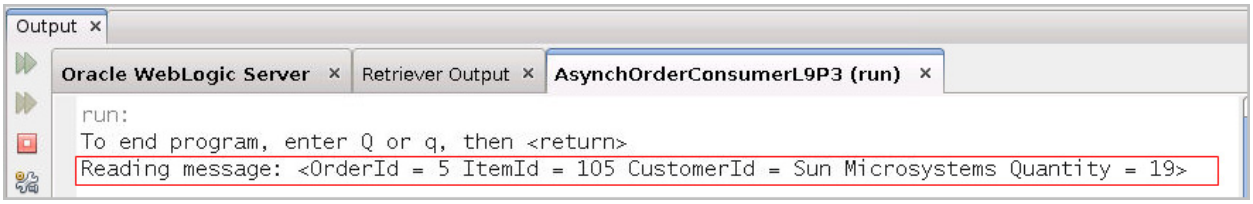
```
out.format("Message sent to orderQueue");
//receiver.receiveMessage();
```

   Recall that messages are delivered to only one consumer that is listening to a queue.
   - Save the file.
   - Make sure that you see that `OrderMgmtApp` is successfully deployed on the WebLogic Server tab.

5. Test the application.
   - Start the `AsynchOrderConsumer` client by right-clicking the project and selecting Run.
   - In the browser, enter the URL to the main HTML page: `http://localhost:7001/OrderMgmtApp`.
   - Enter a new record. For example:



   - Click Order.
   - In the `AsynchOrderConsumer` output window, you should see that the message was asynchronously received.



   - Type `q` in the window and press the Enter key to quit the `AsynchOrderConsumer` client.

Practices for Lesson 10: Developing Message-Driven Beans

Chapter 10

Practices for Lesson 10: Overview

Practices Overview

In these practices, you implement a publish/subscribe messaging model by using JMS message-driven beans.

Practice 10-1: Creating a JMS Topic by Using WebLogic Server

Overview

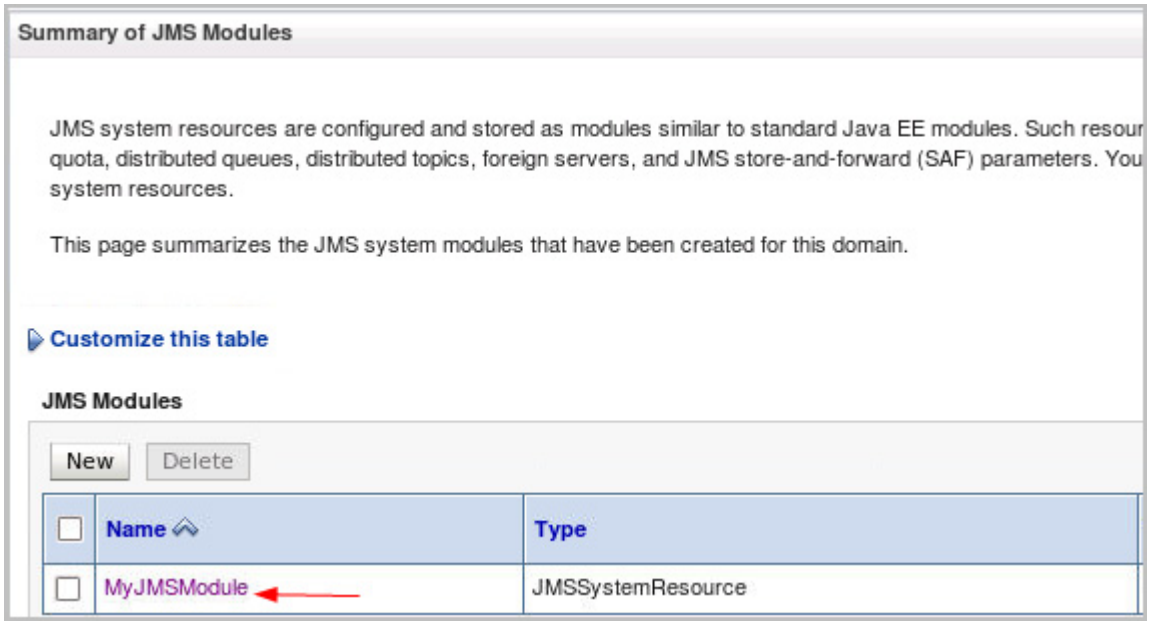
In this practice, you modify `OrderMgmtApp` to send messages to a topic instead of a queue. You use the NetBeans IDE to create the necessary server resources. Specifically, you create a JMS topic destination, `orderTopic`, in the WebLogic Server by using NetBeans.

Assumptions

You completed practice 9-3.

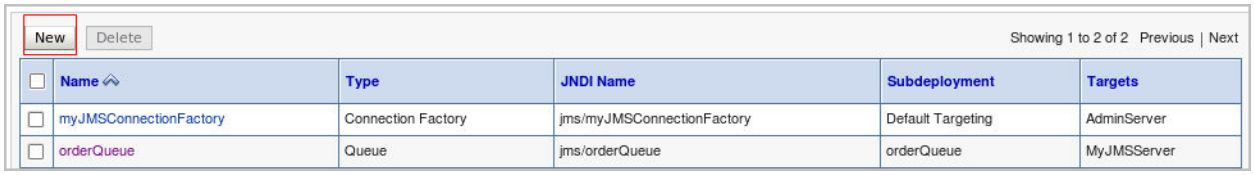
Tasks

1. Create the JMS topic destination `orderTopic` using the WebLogic Server Admin Console.
2. Perform the following steps to create `orderTopic`:
   a. Launch the Admin Console by right-clicking **Oracle WebLogic Server** in Services and selecting **View Admin Console**.
   b. Enter the username and password and click **Login**.
   c. In the **Domain Structure** panel on the left, expand **Services**, expand **Messaging**, and then select **JMS Modules.**
   d. In the Summary of JMS Modules page, click **MyJMSModule**.

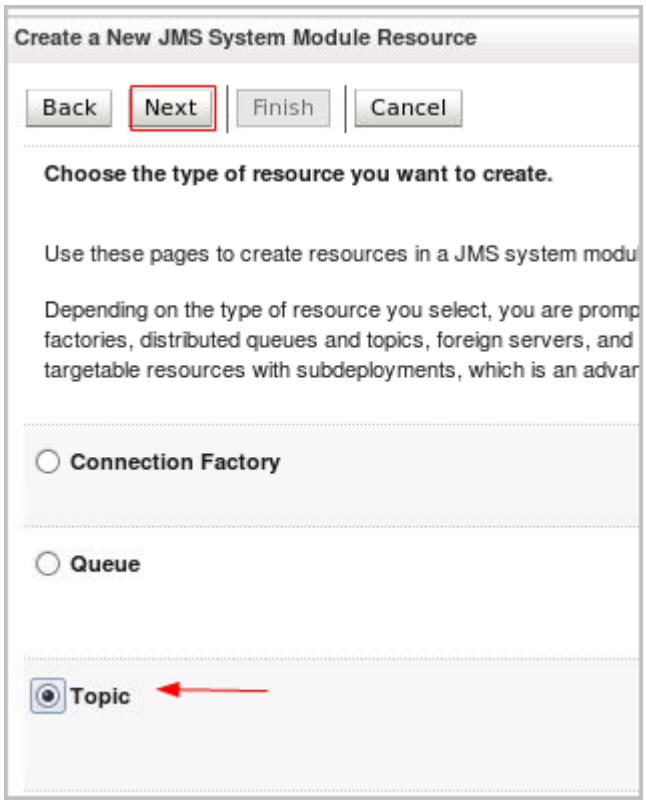


   e. In the Summary of Resources page, the `orderQueue` and the `myJMSConnectionFactory` created in practice 9 are displayed.

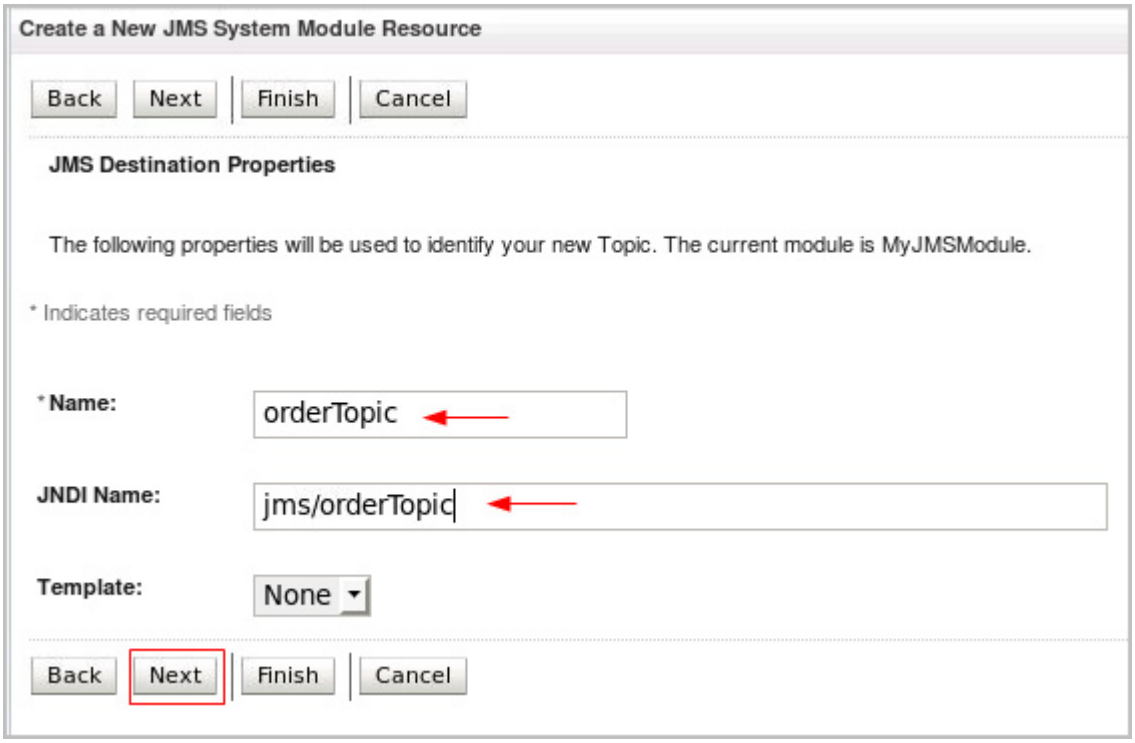
f. Click **New** to add a new resource to the system module.



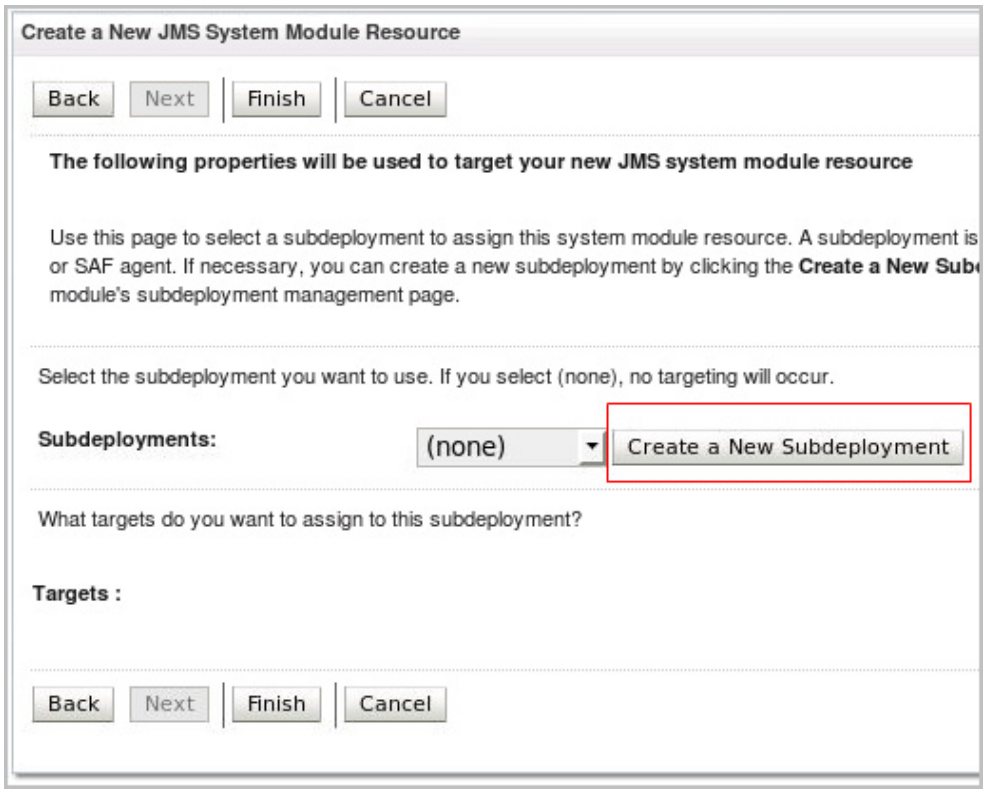
g. Create a new Topic destination by selecting **Topic** from the options. Click **Next**.



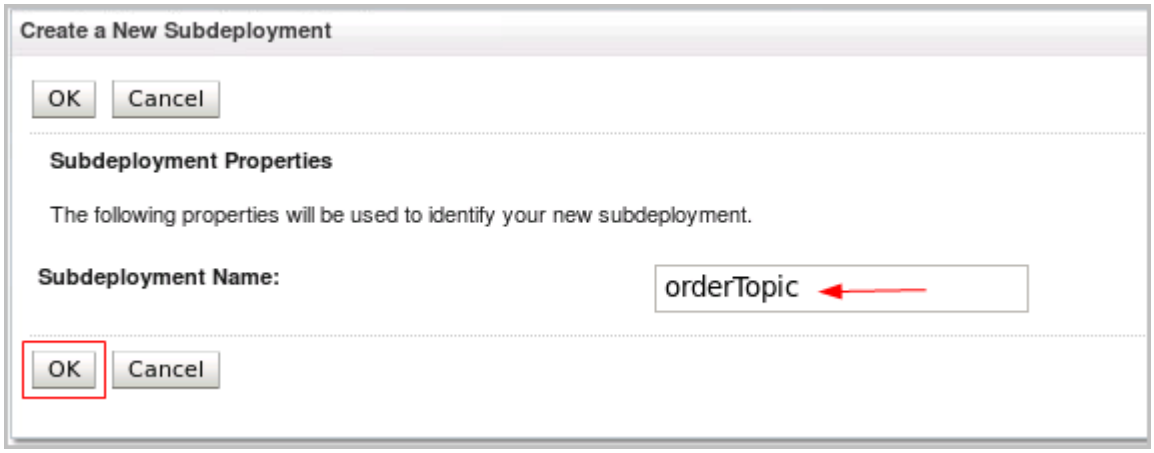
h. Enter `orderTopic` as the Name and `jms/orderTopic` as the JNDI Name for the lookup. Click Next.



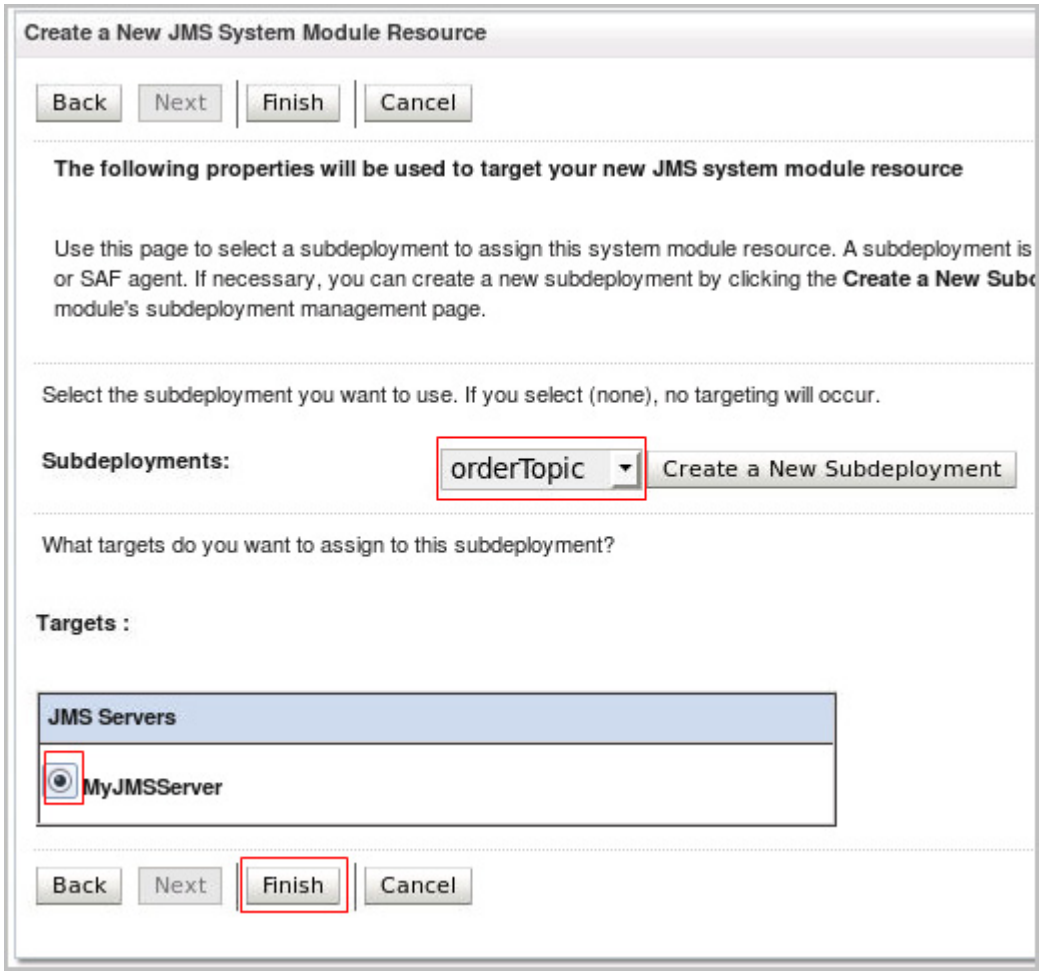
i. On the next screen, click **Create a New SubDeployment** to create a subdeployment for the system module.



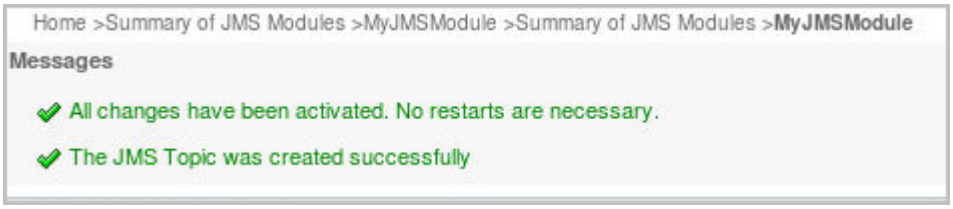
j. On the next screen, click **OK** to accept the default Subdeployment Name.



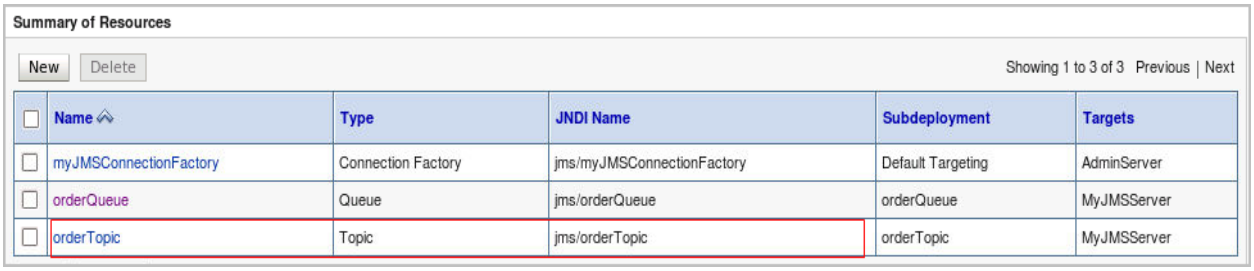
k. With the new subdeployment now selected, select **MyJMSServer** module as the target and click **Finish**.



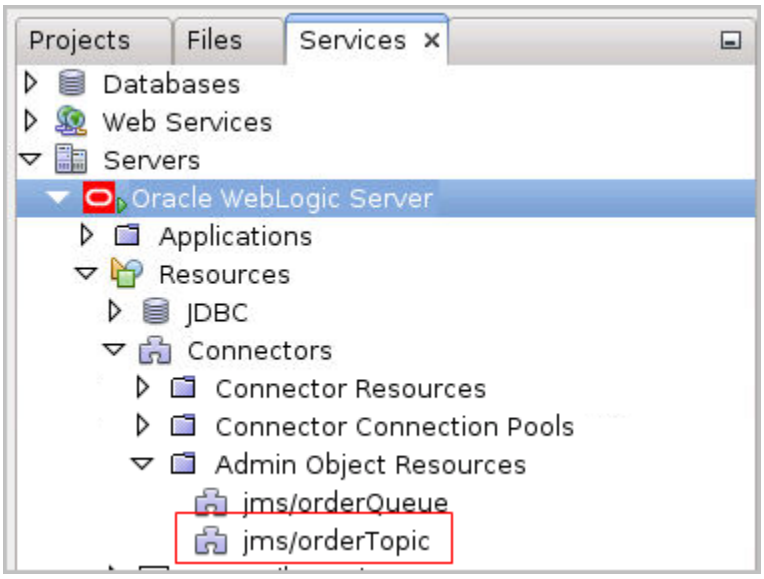
l. Messages will appear at the top of the page to indicate that you successfully created the Topic destination.



m. Near the bottom of the page, the **Summary of Resources** section will show your new Topic resource.



3. Look at the Topic created through NetBeans.
   a. In NetBeans, click the **Services** tab.
   b. Expand **Servers** > **Oracle WebLogic Server**.
   c. Expand **Resources > Connectors**.
   d. Expand the **Admin Object Resources** folder.



4. If you have closed the `OrderMgmtApp` project, reopen it in NetBeans. Modify `OrderProducerBean` to use a Topic instead of a Queue.
   - Open `com.example.OrderProducerBean`.
   - Modify the Resource injection to create an instance of a Topic object instead of a Queue object.

```
@Resource(lookup = "jms/orderTopic")
private Topic topic;
```

   - Modify the `sendMessage` method to use the topic instead of the queue.

```
  public void sendMessage(String message) {
    context.createProducer().send(topic, message);
  }
```

   - Fix any missing import statements and save the file.
5. Redeploy the application.
   - Right-click the project and select **Clean and Build**.
   - When you see that the build was successful, right-click the project again and select **Deploy**.

Practice 10-2: Creating a JMS Message-Driven Bean

Overview

In this practice, you create a message-driven bean, `OrderRecvBean`. The MDB handles messages associated with the `orderTopic` setup in the previous practice.

Assumptions

You have completed practice 10-1.

Tasks

1. Create a new project named `OrderReceiver`.
   a. Click File > New Project.
   b. Select Java Web from Categories and Web Application from File Types, and click Next.
   c. Enter `OrderReceiver` as Project Name and `/home/oracle/labs/projects` as the project location, and click Next.
   d. Retain the defaults in the next window and click Finish.
2. Create a new message-driven bean by using the NetBeans tools.
   a. Right-click the project and select New > Other.
   b. Select Enterprise JavaBeans from Categories and Message-Driven Bean from File Types, and click Next.
   c. Enter the following information and click Next:
      - EJB Name: `OrderRecvBean`
      - Package: `com.example`
      - Server Destinations: `jms/orderTopic` (select from the drop-down menu)
   d. Retain the defaults in the next window and click Finish.
3. Modify the template created by NetBeans.
   a. Remove or comment out the `clientId` property.

```
//@ActivationConfigProperty(propertyName = "clientId",
propertyValue = "jms/orderTopic")
```

4. Implement the `onMessage` method.
   a. Add a `Logger` to the class.

```
private static final Logger logger =
Logger.getLogger("com.example.OrderRecvBean");
```

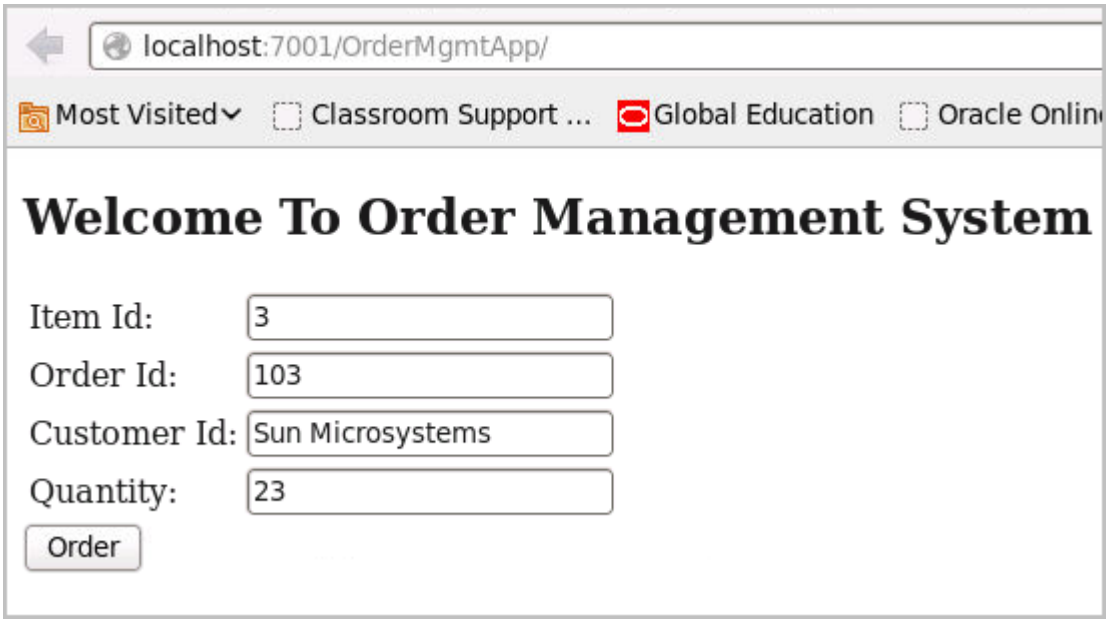
b. In the `onMessage` method body, in a `try/catch` block, use the logger to log the message body. Use the logger to log any `JMSException` that is thrown.

```
try {
  logger.log(Level.INFO, "OrderRecvBean: " +
             message.getBody(String.class));
} catch (JMSException ex) {
  logger.log(Level.INFO, ex.getMessage());
}
```

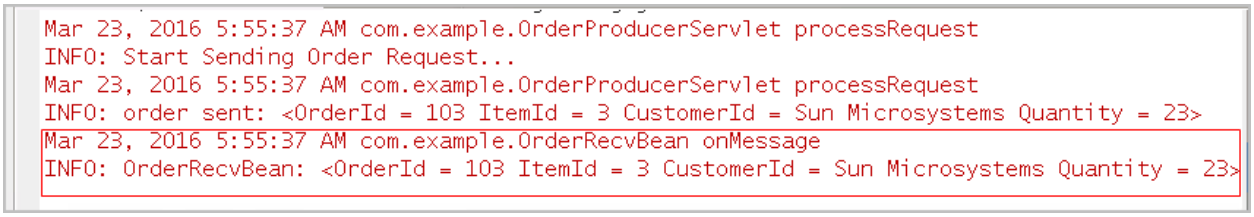
c. Fix any missing import statements (be sure to import `java.util.logging`) and save the file.

5. Deploy `OrderReceiver`.
   - Right-click the project and select **Deploy**, and then **Run**.
6. Test the project.
   a. Open the `OrderMgmtApp` web form by entering `http://localhost:7001/OrderMgmtApp` into a browser.

   b. Enter an order and click Order. For example:



   c. In the Output window of the WebLogic Server, you see the message received by your `OrderRecvBean`:



Practice 10-3: Using a Publish/Subscribe Model with Multiple MDBs

Overview

In this practice, you create a second message-driven bean that is listening to the same topic as the MDB in the previous practice.

Assumptions

You have completed the previous practice.

Tasks

1. Copy the `OrderReceiver` project and create a new project from the copy.
   - Right-click the `OrderReceiver` project and select Copy.
   - In the Copy Project window, change Project Name to `OrderReceiver_2` and click Copy.
2. Modify `com.example.OrderRecvBean` to `com.example.OrderRecvBean2` so as to have a unique client id for the MDB.
   a. Right-click `OrderRecvBean.java`, select Refactor > Rename, modify the name as `OrderRecvBean2`, and click Refactor.
3. In the editor, make the following changes:
   a. Rename the class name and the constructor name to `OrderRecvBean2`.
   b. Modify the Logger statement as shown below:

```
public class OrderRecvBean2 implements MessageListener {
private static final Logger logger =
Logger.getLogger("com.example.OrderRecvBean2");
public OrderRecvBean2() {
}
```

   c. Modify the log message to show that it came from a different MDB, `OrderRecvBean2`:

```
logger.log(Level.INFO, "OrderRecvBean2: " +
message.getBody(String.class));
```

4. Test the application.
   - Follow the steps that you performed in practice 10-2, step 6.
   - The output in the WebLogic Output window should show messages logged from both MDBs:

```
Apr 26, 2016 11:50:58 AM com.example.OrderProducerServlet processRequest
INFO: Start Sending Order Request...
Apr 26, 2016 11:50:58 AM com.example.OrderProducerServlet processRequest
INFO: order sent: <OrderId = 103 ItemId = 3 CustomerId = Sun Microsystems Quantity = 23>
Apr 26, 2016 11:50:58 AM com.example.OrderRecvBean2 onMessage
INFO: OrderRecvBean2: <OrderId = 103 ItemId = 3 CustomerId = Sun Microsystems Quantity = 23>
Apr 26, 2016 11:50:58 AM com.example.OrderRecvBean onMessage
INFO: OrderRecvBean: <OrderId = 103 ItemId = 3 CustomerId = Sun Microsystems Quantity = 23>
```

Practices for Lesson 11: Java EE Concurrency

Chapter 11

Practices for Lesson 11: Overview

Practice Overview

In these practices, you optimize the performance of several scenarios by using the asynchronous utilities in Java EE.

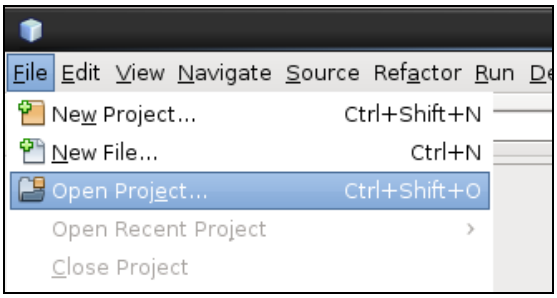
Practice 11-1: Asynchronous EJB

Overview

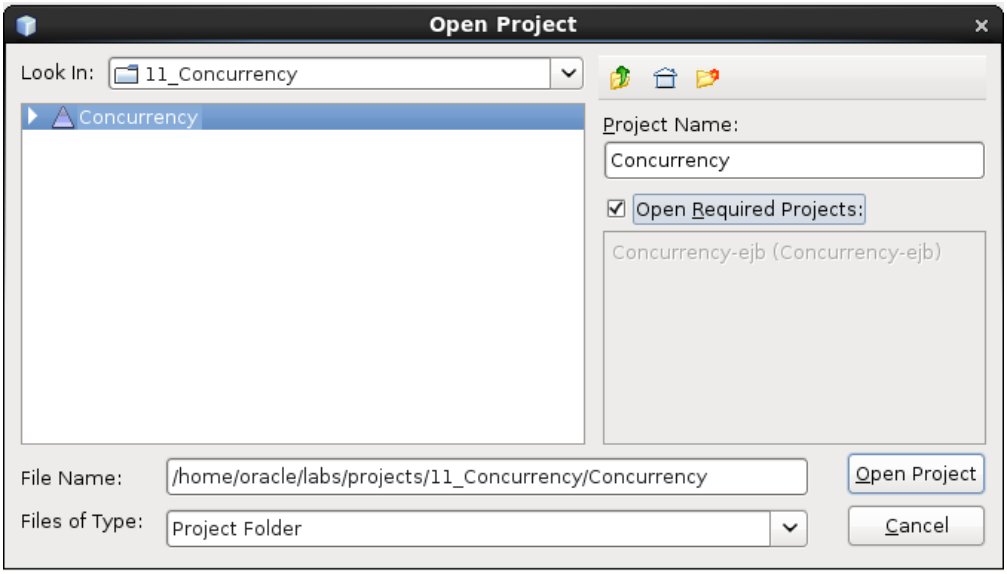
In this practice, you use asynchronous EJB for methods that return no value, to optimize a Web Service that takes a long time to complete.

Tasks

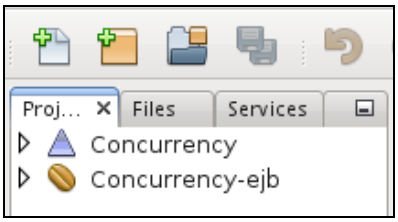
1. Double-click the NetBeans icon on your desktop to open it.
2. Select File > Open Project.



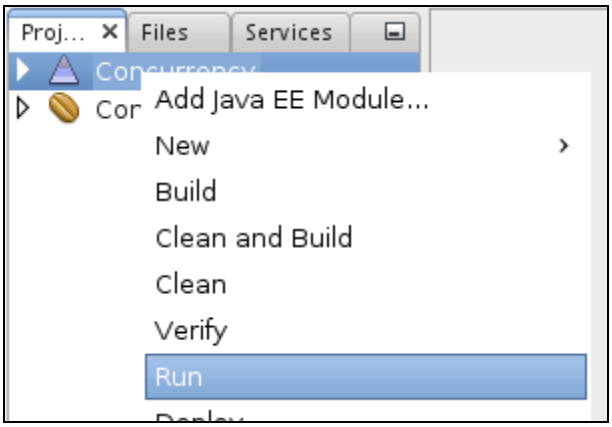
3. Browse to the `/home/oracle/labs/projects/11_Concurrency` folder, click the `Concurrency` project, select the Open Required Projects check box, and open the project.



After a short delay, you should see two projects open.

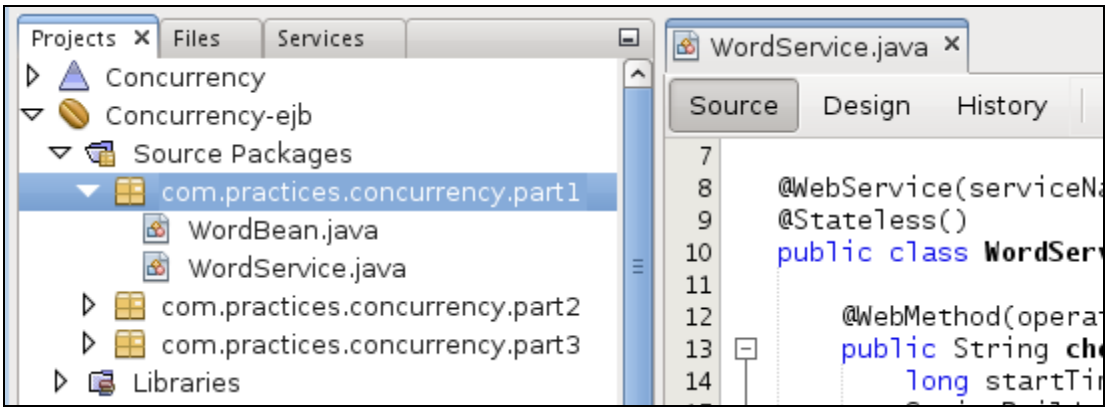


4. Right-click the `Concurrency` project and select Run from the context menu.



The project is built and deployed on the WebLogic server.

5. Expand the `Concurrency-ejb` project, Source Packages, and `com.practices.concurrency.part1` nodes. Open the `WordService` class.



6. Inject the `WordBean` EJB. After the class declaration, add the following code:

```
@Inject
private WordBean wordBean;
```

7. Press Ctrl + Shift + I to resolve any Java class import issues. Alternatively, add the import:

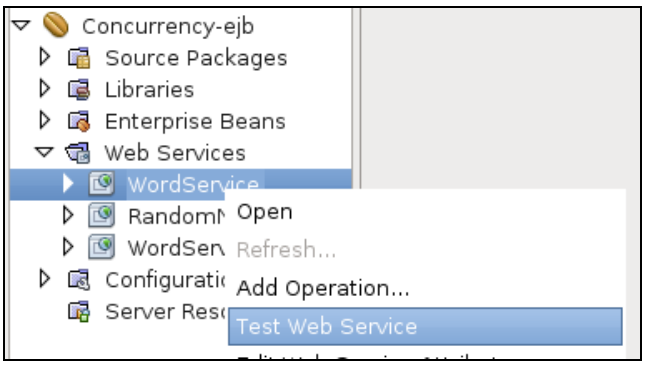
```
import javax.inject.Inject;
```

8. Locate the "`//- Register Words Here -`" comment. Add the following code to register the words by using the `WordBean` EJB:

```
for (String word : words.split("\\ ")) {
  wordBean.register(word);
}
```

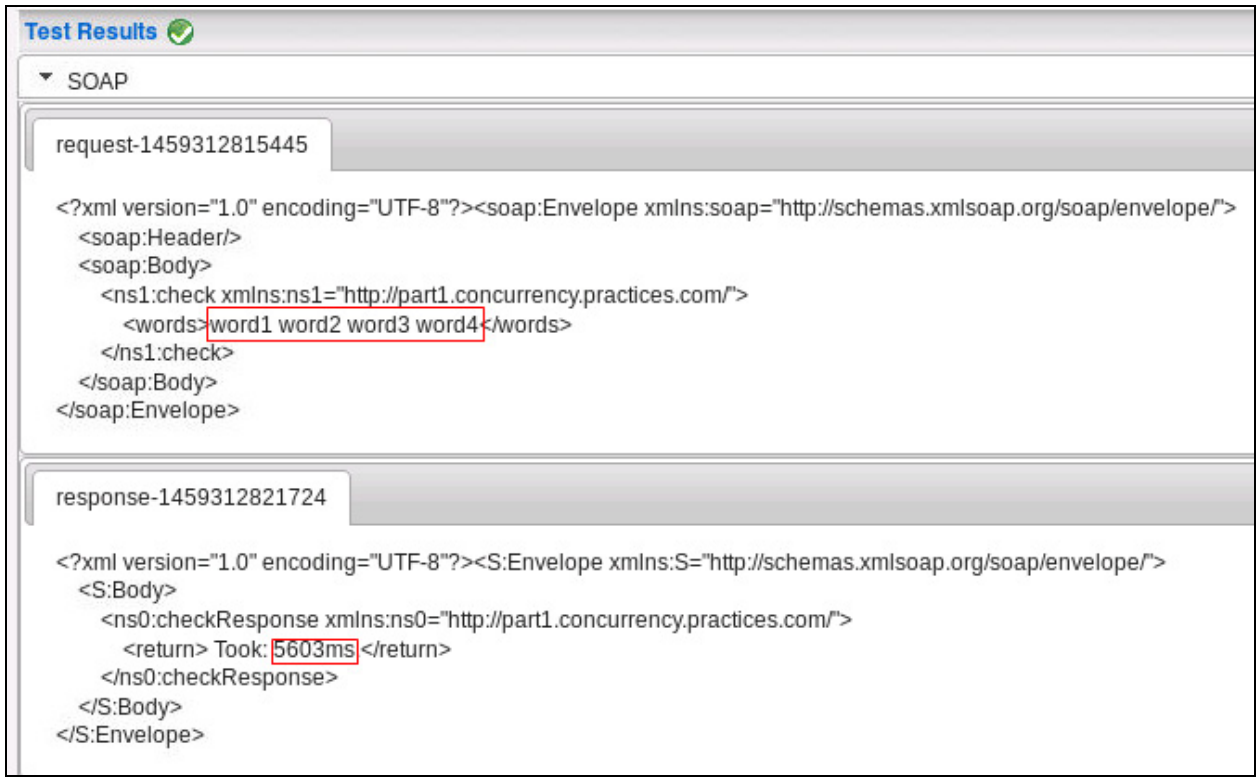
9. Save the file.

10. Expand the Web Services node, right-click the WordService web service, and select Test Web Service.



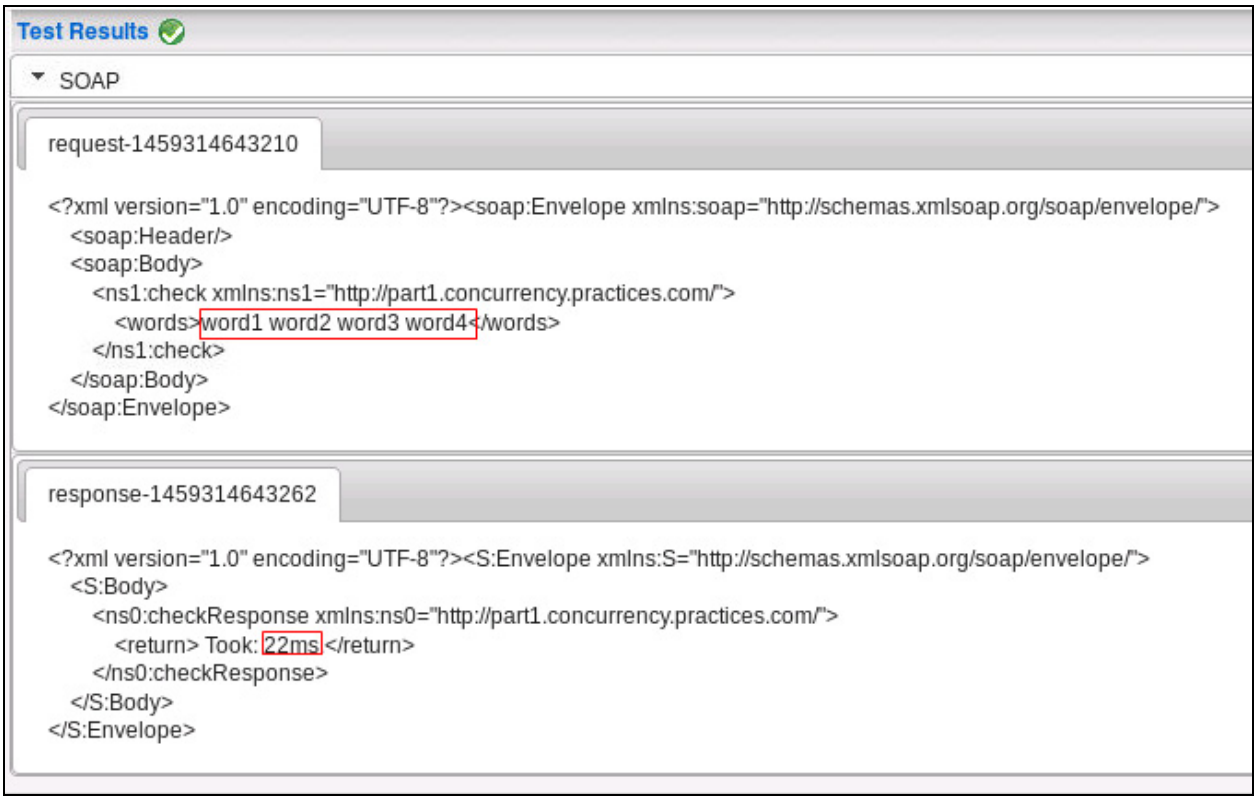
11. A browser opens with the Web Service tester. Click **Test**.

12. Enter `word1 word2 word3 word4` in the field and click **Invoke** at the bottom of the page.
    Note the time that it takes to complete. It takes too much time because the `WordBean` is taking a while to process each of the words. Also the service does not require the `WordBean` to return any value.



This can be optimized by using an asynchronous EJB.

13. Open the `WordBean` class in the `com.practices.concurrency.part1` package:
    a. Add the `@javax.ejb.Asynchronous` annotation to the class. Alternatively, you can add the annotation to the `register` method.
    b. Save the file.
14. Open the web browser with the Web Service tester. Click **Test**.
15. Enter `word1 word2 word3 word4` in the field and click **Invoke**.



The time taken reduces significantly because the invocations of the EJB take place in different threads and the services do not need to wait for the invocations to finish.

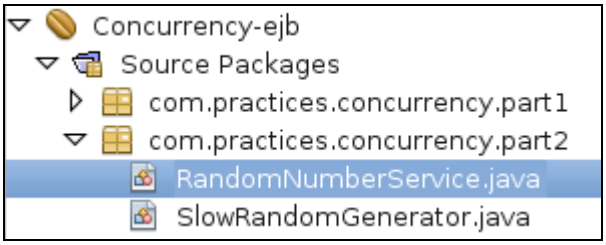
Practice 11-2: Asynchronous Methods with Return Values

Overview

In this practice, you modify a web service that generates random numbers by using an EJB. You add asynchronous support to the number generation.

Tasks

1. Open the `RandomNumberService` class in the `com.practices.concurrency.part2` package and perform the following steps:



   a. Inject the `SlowRandomGenerator` EJB in the service class. Add the following code after the class declaration:

```
@Inject
private SlowRandomGenerator slowBean;
```

   b. Press Ctrl + Shift + I to resolve any Java class import issues. Alternatively, add the following import:

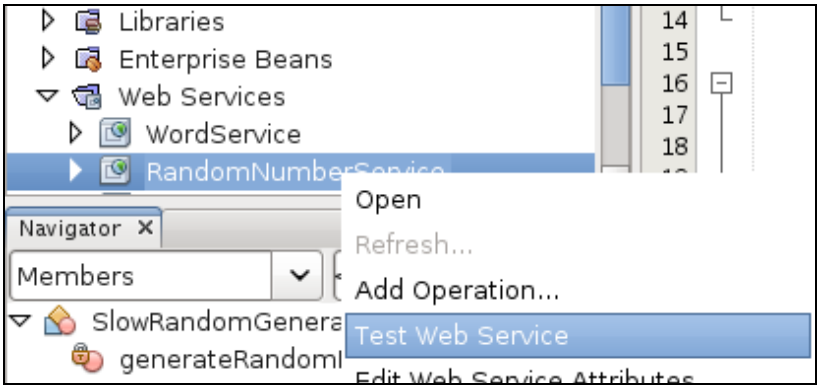
```
import javax.inject.Inject;
```

   c. Locate the "`//- Generate Random Numbers-`" comment. Add the following code to generate random numbers by using the `SlowRandomGenerator` EJB:

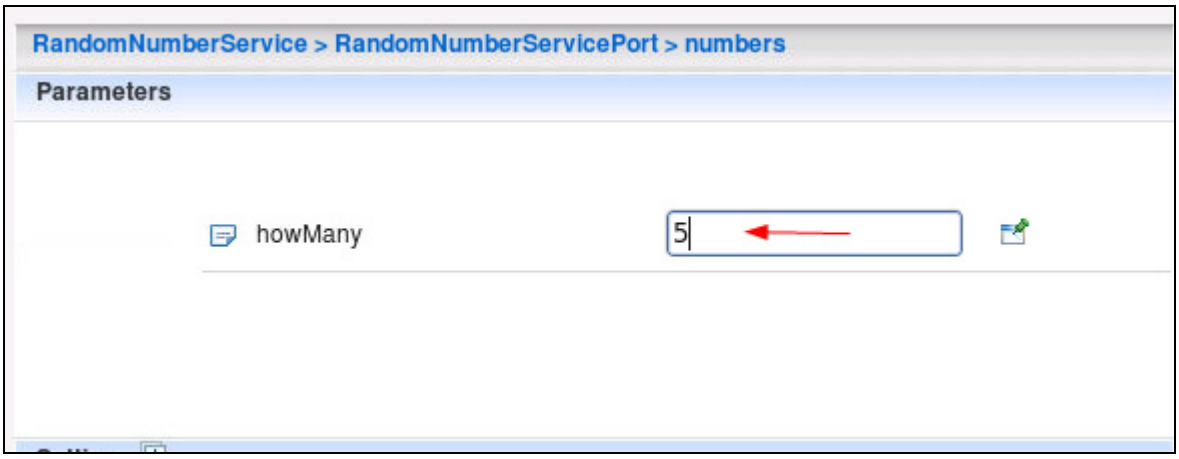
```
for (int i = 0; i < howMany; i++) {
  str.append(slowBean.getRandomNumber());
  str.append(", ");
}
```

   d. Save the file.

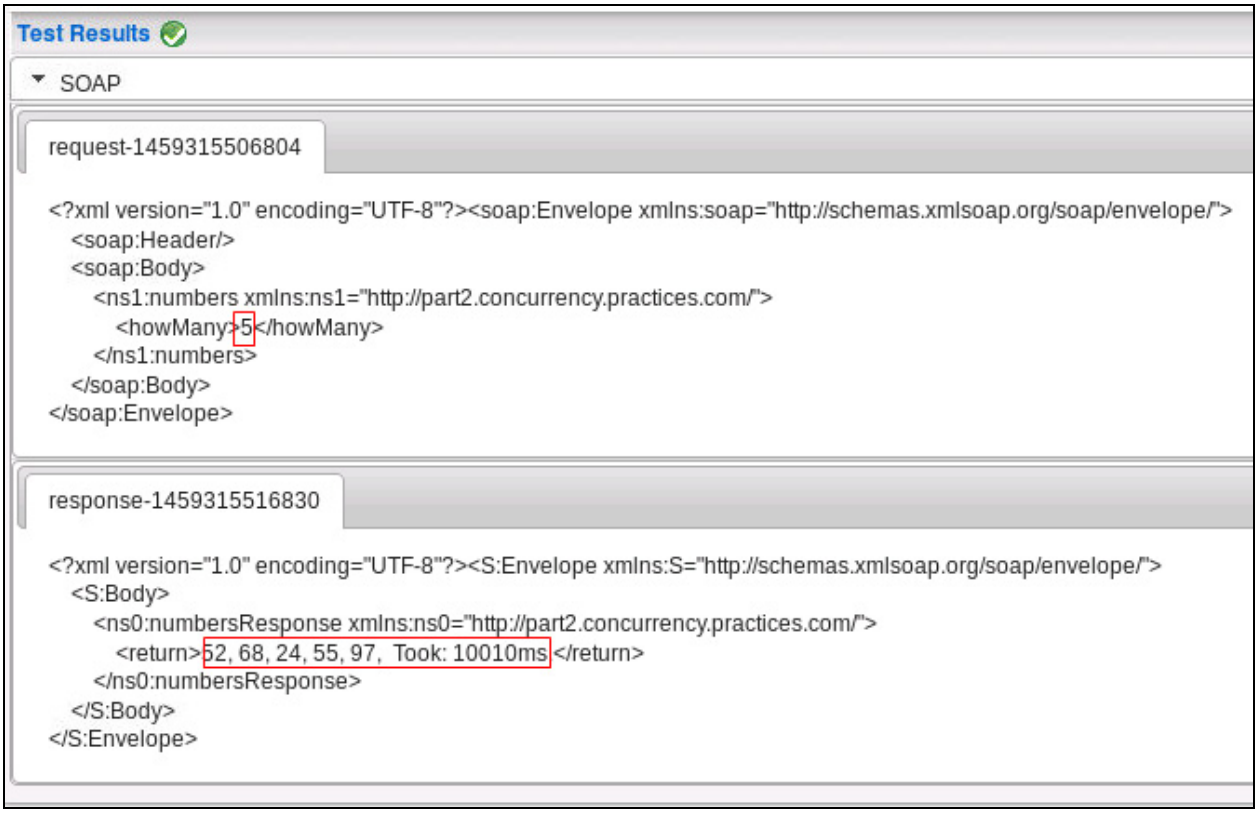
2. Expand the Web Services node, right-click the `RandomNumberService` web service, and select Test Web Service.



3. The browser opens with the Web Service tester. Click **Test**.
4. Enter `5` in the `howMany` field and click **Invoke** at the bottom of the page.



The result of the invocation is displayed at the bottom of the page.



Note that the invocation takes a long time to complete. You modify the bean and the service to use asynchronous invocations with return values.

5. Open the `SlowRandomGenerator` class in the `com.practices.concurrency.part2` package and perform the following steps:
   a. Add the `@Asynchronous` annotation to the `getRandomNumber` method.
   b. Change the return type from `int` to `Future<Integer>`.
   c. Wrap the result of `generateRandomNumber` inside an `AsyncResult<Integer>` object.
      Use the following code for reference:

```
@Asynchronous
public Future<Integer> getRandomNumber() {
  final int number = generateRandomNumber();
  return new AsyncResult<>(number);
}
```

   d. Press Ctrl + Shift + I to resolve Java class import issues. Alternatively, add the following imports:

```
import java.util.concurrent.Future;
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
```

   e. Save the file.
6. Open the `RandomNumberService` class. You need to adapt your code for multiple return values asynchronously. Perform the following steps:
   a. Add the `throws` keyword to the `getNumbers` method for the following exception types: `java.util.concurrent.ExecutionException` and `java.lang.InterruptedException`.

```
public String getNumbers(@WebParam int howMany) throws
InterruptedException, ExecutionException {
```

   b. After the "`//- Generate Random Numbers-`" comment and before the loop starts, declare a `List<Future<Integer>> numbers ArrayList` variable.

```
List<Future<Integer>> numbers = new ArrayList<>();
```

   c. Remove the contents of the "for" loop and add the following code instead:

```
numbers.add(slowBean.getRandomNumber());
```

      This invokes the random number generation asynchronously and stores the results in a list where you can request for numbers at the same time.
   d. Add another loop to iterate all the elements in the numbers list and add the numbers by using the `future.get` method.

```
        for (Future<Integer> number : numbers) {
            str.append(number.get());
            str.append(", ");
        }
```

e. Press Ctrl + Shift + I to resolve any Java class import issues. Alternatively, add the following imports:

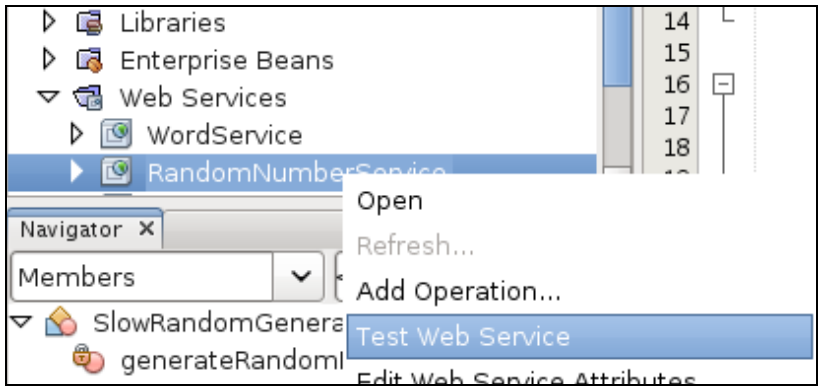
```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Future;
import java.util.concurrent.ExecutionException;
```

f. Save the file.

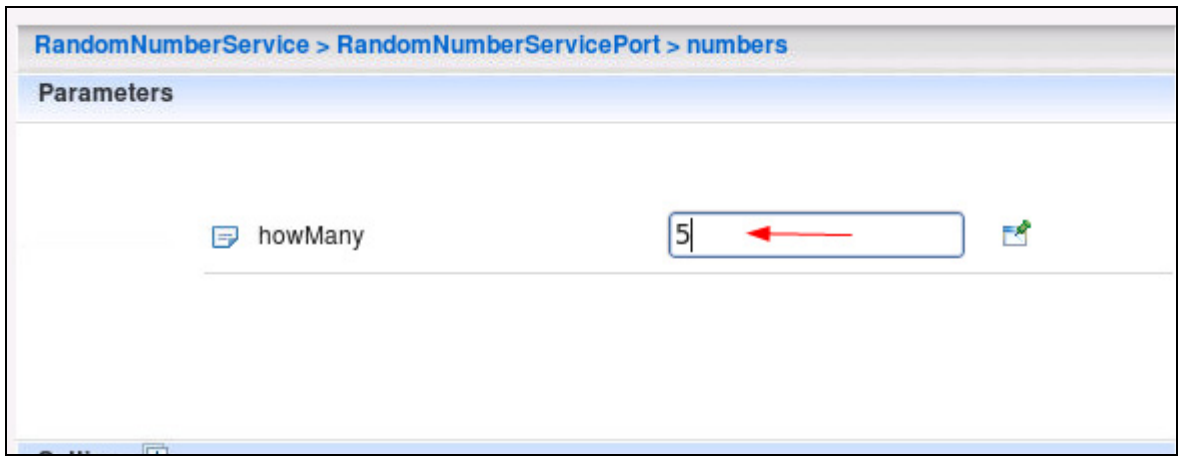
Use the following code for reference:

```
// - Generate Random Numbers -
List<Future<Integer>> numbers = new ArrayList<>();
for (int i = 0; i < howMany; i++) {
  numbers.add(slowBean.getRandomNumber());
}
for (Future<Integer> number : numbers) {
  str.append(number.get());
  str.append(", ");
}
// - End Random Number Generation -
```

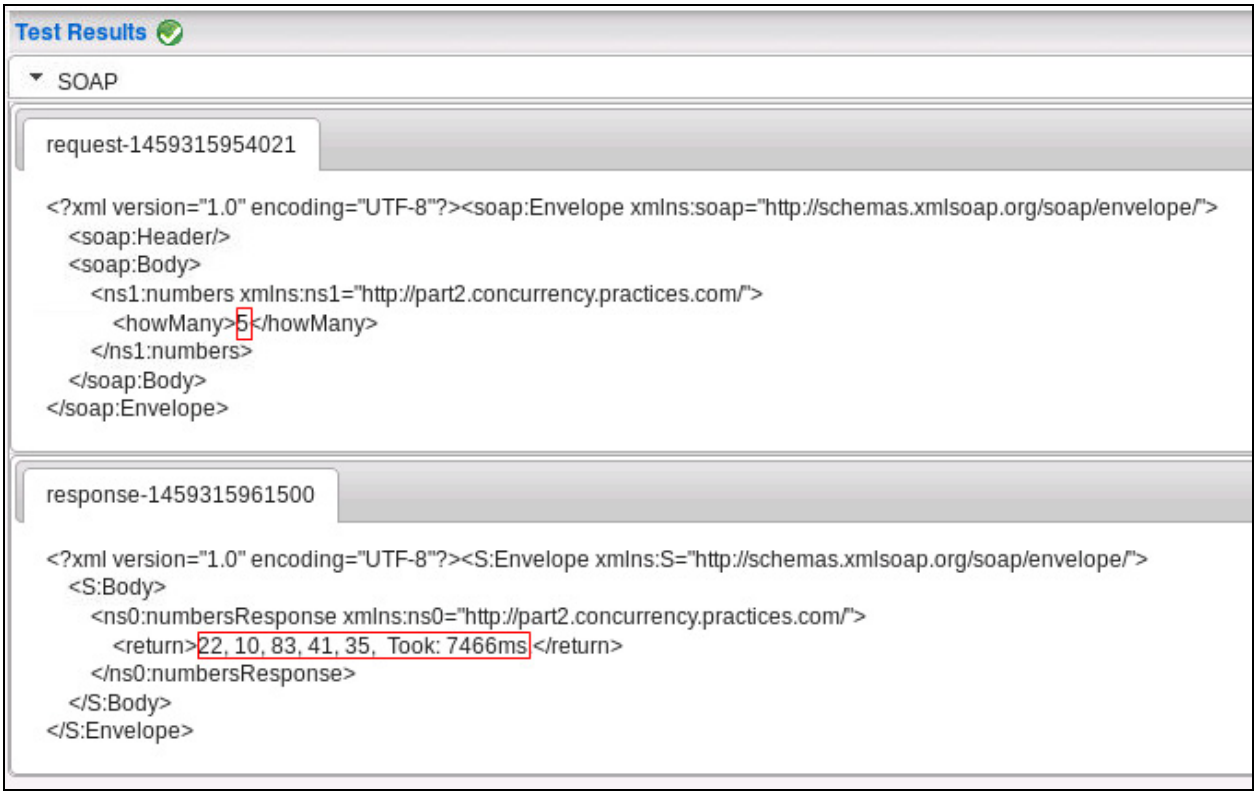
7. Expand the Web Services node, right-click the `RandomNumberService` web service, and select Test Web Service.



8. The browser opens with the Web Service tester. Click **Test**.
9. Enter `5` in the `howMany` field and click the **Invoke** button at the bottom of the page.



The result of the invocation is displayed at the bottom of the page.



Note: Try increasing the number from 5 to 10 and notice if the time increments. You note that the performance is better by using this technique.

Practice 11-3: Concurrency Utilities for Java EE

Overview

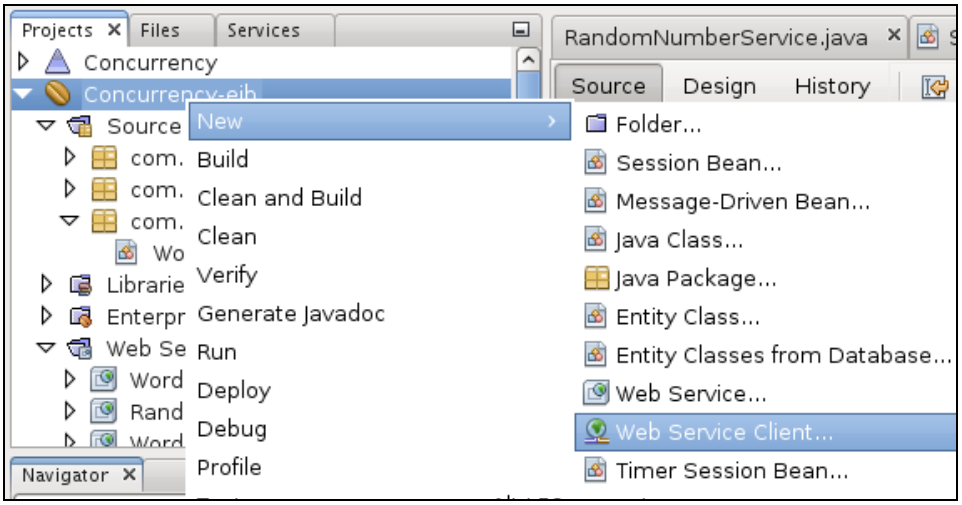
In this practice, you invoke remote Web Services by using the Java EE Concurrency Utilities.

With external services, performance does not depend entirely on your EJBs. Network latency can influence the performance of your services. You can use asynchronous methods to invoke remote services but you can also use Java EE Concurrency Utilities to invoke these services efficiently.

The web service used in this practice is very similar to the one that was used in practice 11-1, but in this case, the service is remote and has a return value for each of the words registered.

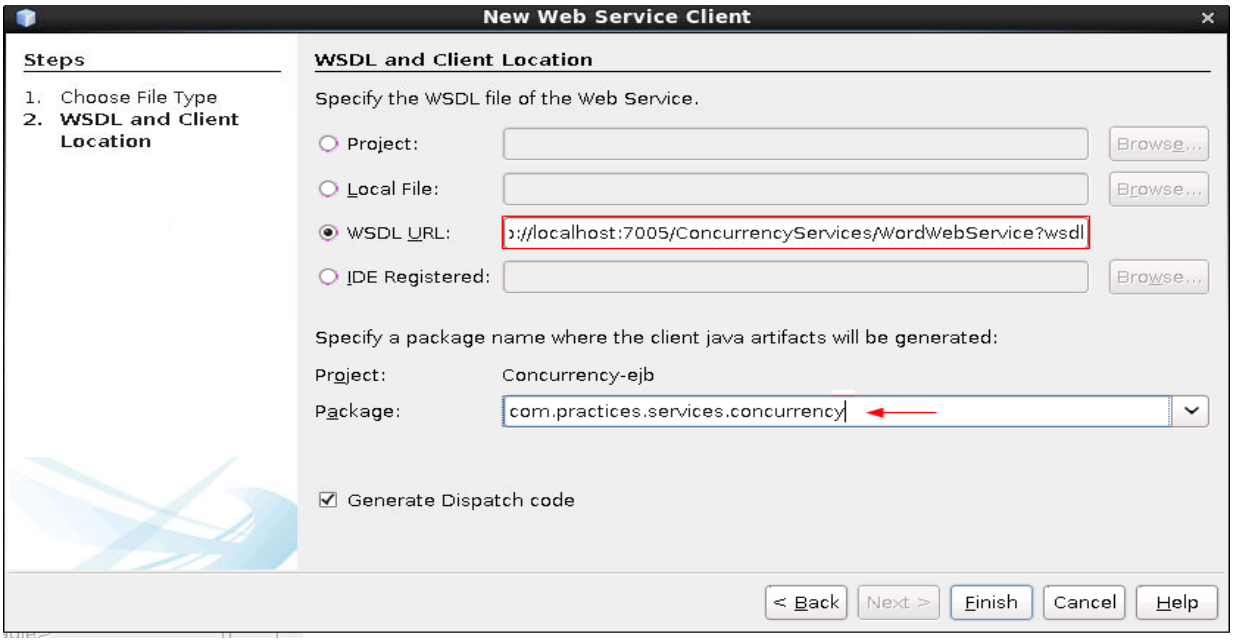
Tasks

1. Right-click the `Concurrency-ejb` project and select New > Web Service Client from the menu.



2. Select WSDL URL and set it to `http://localhost:7005/ConcurrencyServices/WordWebService?wsdl`.

3. Set the package to `com.practices.services.concurrency`.



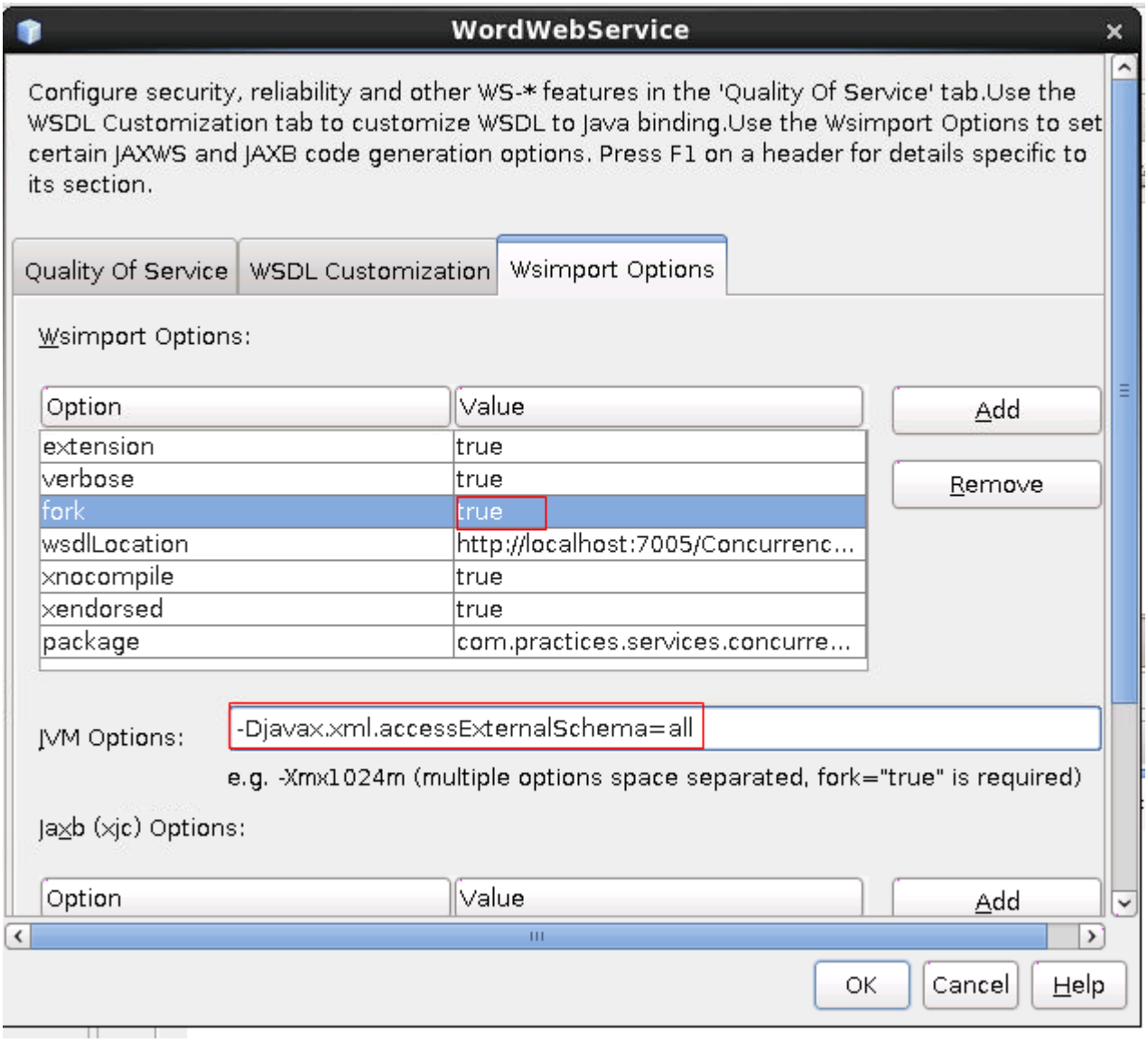
4. Click Finish.

   The wsimport operation might fail. To prevent this, you edit the wsimport options to allow the XML parser to use any protocol to get the WSDL file.

5. Open the Web Service References node, right-click the `WordWebService` element, and select Edit Web Service Attributes.

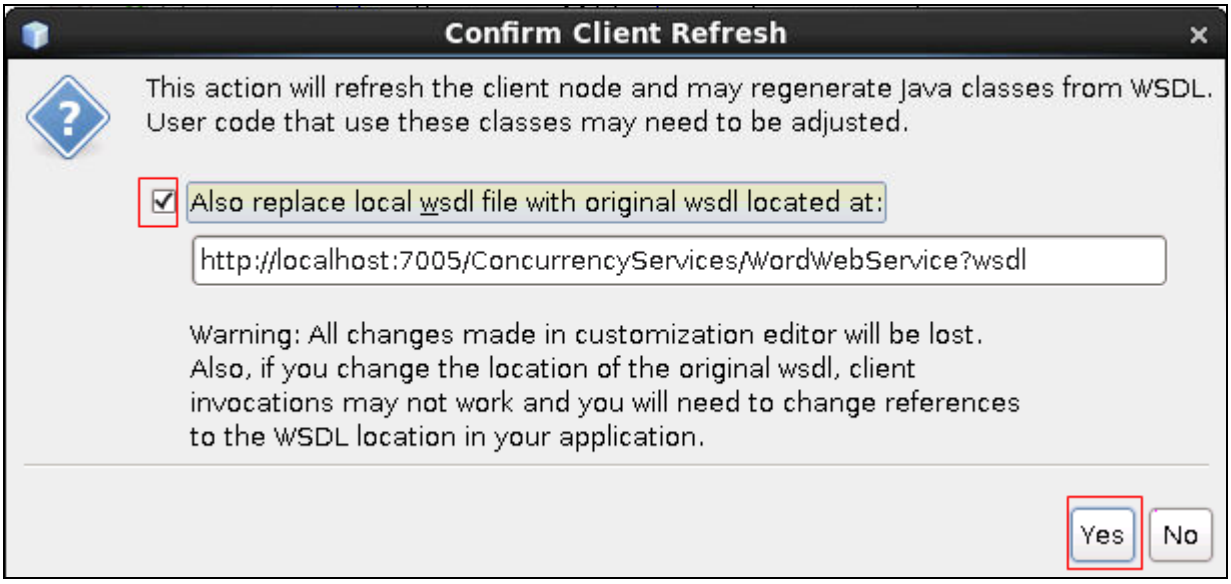
6. Go to the Wsimport Options tab.
7. Set fork to `true`.

8. Add the following JVM Option: `-Djavax.xml.accessExternalSchema=all`.



9. Click OK.
10. Right-click the `WordWebService` element and select Refresh.
11. Select the “Also replace local wsdl file with original wsdl located at:” check box.

12. Click Yes.



This generates the code that is required for the remote Web Service.

13. Open the `WordServiceDistributed` class in the `com.practices.concurrency.part3` package.
14. Add the `ManagedExecutorService` resource to the class. Add the following after the class declaration:

```
@Resource(name = "java:comp/env/concurrent/ServiceInvoker")
private ManagedExecutorService executorService;
```

15. Press Ctrl + Shift + I to resolve any Java class import issues. Alternatively, add the following imports:

```
import javax.annotation.Resource;
import javax.enterprise.concurrent.ManagedExecutorService;
```

16. Add a `throws` declaration to the process method of the `java.lang.Exception` exception type. The code to be added might throw exceptions.

```
public String process(@WebParam(name = "words") String words)
throws Exception {
```

17. Locate the "`// - invoke external registry service –`" comment and add a list to store the result of the Web Service invocations. Results are stored in `Future<String>` objects.

```
List<Future<String>> results = new ArrayList<>();
```

18. Press Ctrl + Shift + I to resolve any Java class import issues. Alternatively, add the following imports:

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Future;
```

19. Create an instance of the WebService to invoke. You reuse this instance for each web service call.

```
WordWebService service = new
WordWebService_Service().getPort(WordWebService.class);
```

20. Press Ctrl + Shift + I to resolve any Java class import issues. Alternatively, add the following imports:

```
import com.practices.services.concurrency.WordWebService;
import com.practices.services.concurrency.WordWebService_Service;
```

21. Add the following code to add results to the results list:

```
for (String word : words.split("\\ ")) {
  results.add(invokeRemote(word, service));
}
```

Note that the code will not compile because the `invokeRemote` method does not exist yet. You create the method in the following steps.

22. Gather the results and add them to the response message of the Web Service.

```
for (Future<String> result : results) {
  str.append(result.get());
  str.append("\n");
}
```

23. Create the `invokeRemote` method.

```
private Future<String> invokeRemote(String word, WordWebService
service)
```

24. Create a new Callable instance that invokes the `WordWebService` service and returns the result of the invocation.

```
Callable<String> callable = new Callable<String>() {
  @Override
  public String call() throws Exception {
    return service.register(word);
  }
};
```

25. Finally return the result of `executorService.submit(callable);`.

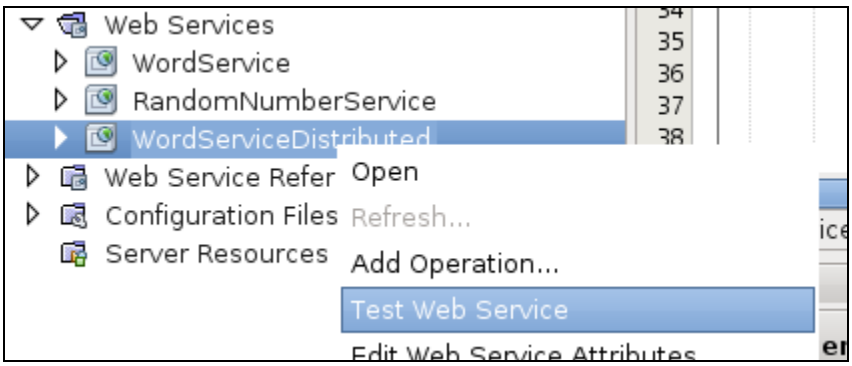
Use the following code for reference:

```
private Future<String> invokeRemote(String word, WordWebService
service) {
  Callable<String> callable = new Callable<String>() {
    @Override
    public String call() throws Exception {
      return service.register(word);
    }
  };
  return executorService.submit(callable);
}
```

26. Press Ctrl + Shift + I to resolve any Java class import issues. Alternatively, add the following import:

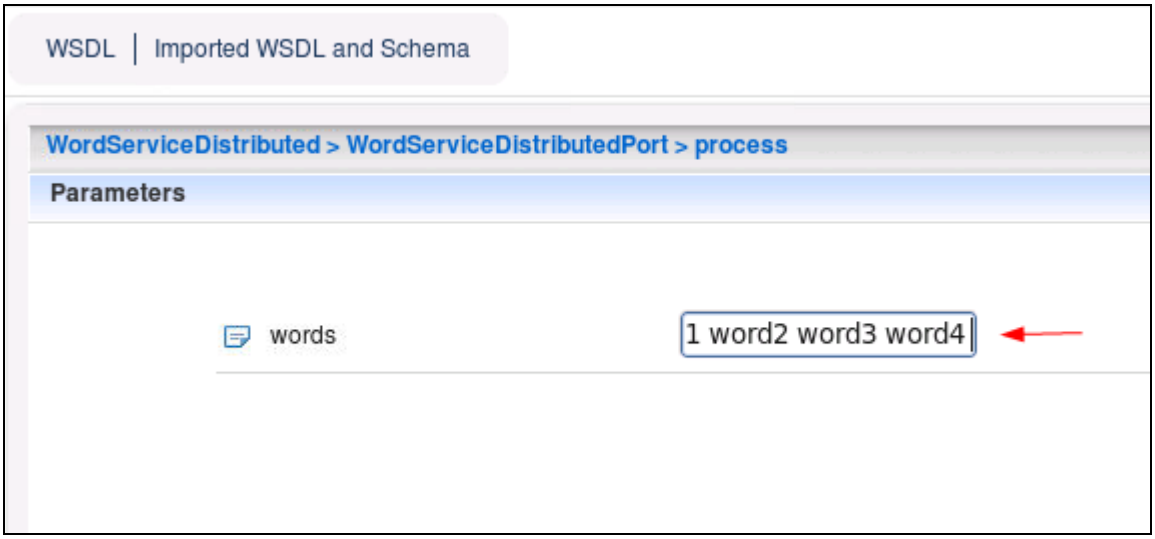
```
import java.util.concurrent.Callable;
```

27. Save the file.
28. Expand the Web Services node, right-click the `WordServiceDistributed` web service, and select Test Web Service.

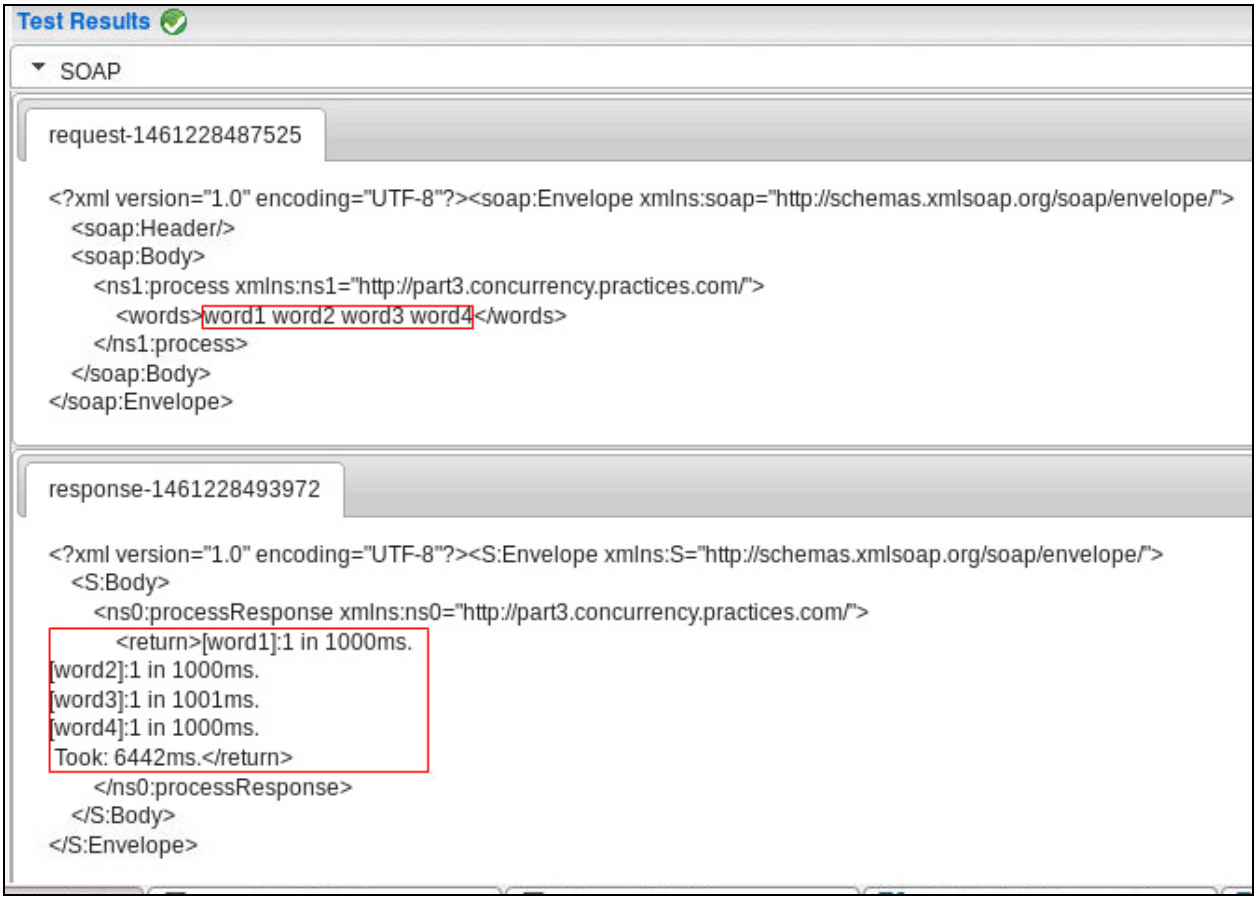


29. A browser opens with the Web Service tester. Click **Test**.

30. Enter `word1 word2 word3 word4` in the `words` field and click **Invoke**.



After a delay, the result of the invocation is displayed.



Note that each word takes 1 second to process on the remote service, but the words are processed asynchronously and simultaneously. If you do not use asynchronous processing, the total time would be the sum of the processing time of each word.

Practices for Lesson 12: Using JDBC in Java EE Environments

Chapter 12

Practices for Lesson 12: Overview

Practices Overview

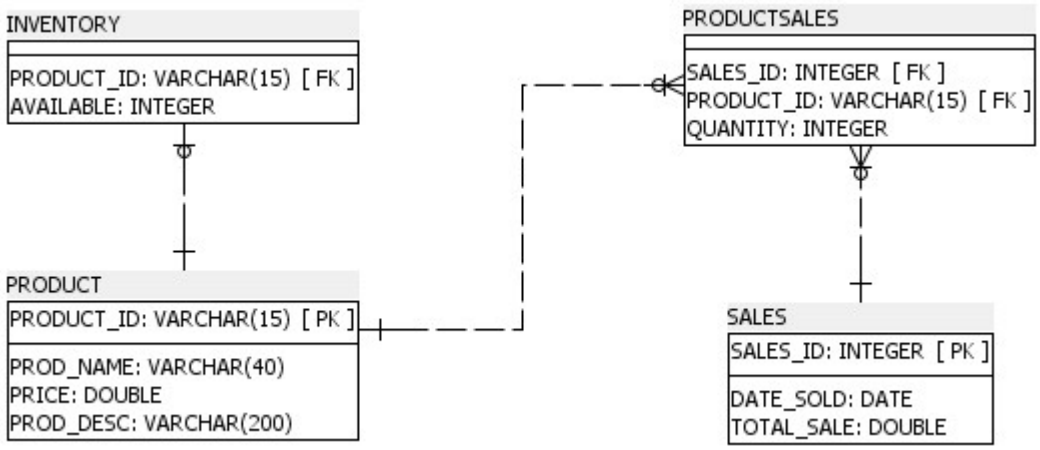
In these practices, you create an application that interacts with a database by using Java Database Connectivity (JDBC).

Practice 12-1: Creating the `JavaMart` Database

Overview

In this practice, you create a database for the `JavaMart` application and populate the database with sample data.

The `JavaMart` application is a simple store-front application that sells Java and Duke items. The data model is as follows:



- The Product table contains information about each product item, including name, description, and price.
- The Inventory table tracks how many products of each product ID are available to ship.
- When a shopping cart is purchased, the items and their quantities are written to the ProductSales table.
- The total sale is tracked in the Sales table.

Assumptions

If you completed practice 6-1 titled “Setting Up a Java DB Database,” create a new database in this practice with the following parameters:

```
Database name: JavaMartDB
User name: oracle
Password: as specified in the security credentials page
Confirm password: as specified in the security credentials page
```

Create the database tables and initial data by using the script that is located in: `/home/oracle/labs/resource/JavaMartDB.sql`.

Detailed instructions are as follows:

Tasks

1. Start the Java DB database (if it is not already running).
   - Open the Services tab by clicking Windows > Services.
   - Expand Databases, right-click Java DB, and select Start Server.

   If the Start Server option is not available, the database is already running.
2. Create a new database named `JavaMartDB`.
   - Right-click Java DB and select Create Database.

- Enter `JavaMartDB` as the database name.
- Enter `oracle` as the username.
- Enter the password as specified in the security credentials page.
- Click OK.

3. Enable the NetBeans Favorites window.
   - In NetBeans, click Window in the menu bar and select Favorites. A new tab should appear in the same group as the Projects tab. The Favorites window lets you view, copy, and edit the files that are located in any directory on your system.
   - Right-click in the Favorites window and add the following directories:
     - `/home/oracle/labs/resources`
     - `/home/oracle/labs/solutions`
4. Execute the script to create and populate the database.
   - Click the Favorites tab and expand `/home/oracle/labs/resources`.
   - Double-click the file `JavaMartDB.sql` to open it in the editor pane.
   - In the editor pane that opens, select `jdbc:derby://localhost:1527/JavaMartDB[oracle on ORACLE]` from the Connection drop-down list.
   - Click the Run SQL icon to the right of the connection.

   Note: You may see some errors that the tables cannot be dropped, which is OK. If you run the script a second time, there should be no errors.

Practice 12-2: Writing Data Access Objects with JDBC

Overview

In this practice, you complete the starter project `JavaMartWeb` by writing the required JDBC DAO methods.

Assumptions

You have completed the previous practice and the database is running.

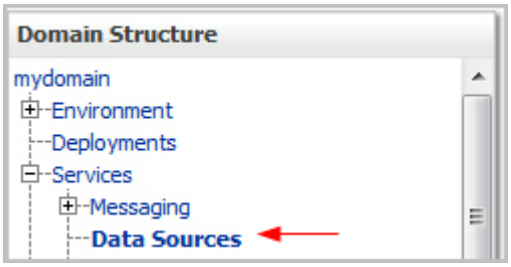
Tasks

1. Review the starter project, `JavaMartWeb`.
   - Open the `JavaMartWeb` project from the `/home/oracle/labs/projects/12_JDBC` directory.
   - Expand Source Packages and then expand each of the package directories individually.
   - In the servlet package, note that two servlets have already been developed for you—these provide a basic interface to test the functionality of the application.
   - In the model package, there is one POJO for each of the four data tables (entities) in the database.
   - In the beans package, there is a `ShoppingCart` CDI bean that is `SessionScoped`. This bean holds the items that users place in their carts.
   - In the `dao` package, note that the CDI beans are annotated with `@Model`, making them `RequestScoped` and `Named`. These are the CDI beans that you complete in this practice.
2. Add a JDBC Connection and Connection Pool to WebLogic server.

   To add a connection pool for the new database, you need to configure a JDBC data source using the WebLogic Server administration console. Perform the following steps:

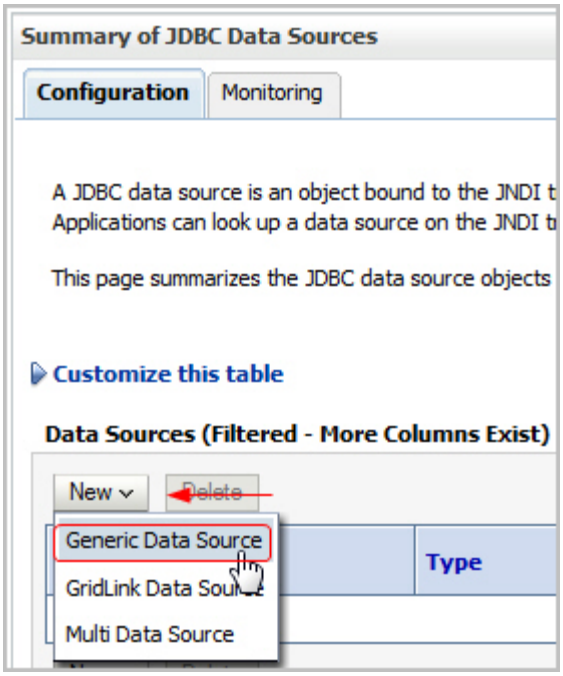
   a. Access the WebLogic Server administration console and log in to the console.

   b. Under Domain Structure, expand **Services**, and then click **Data Sources**.

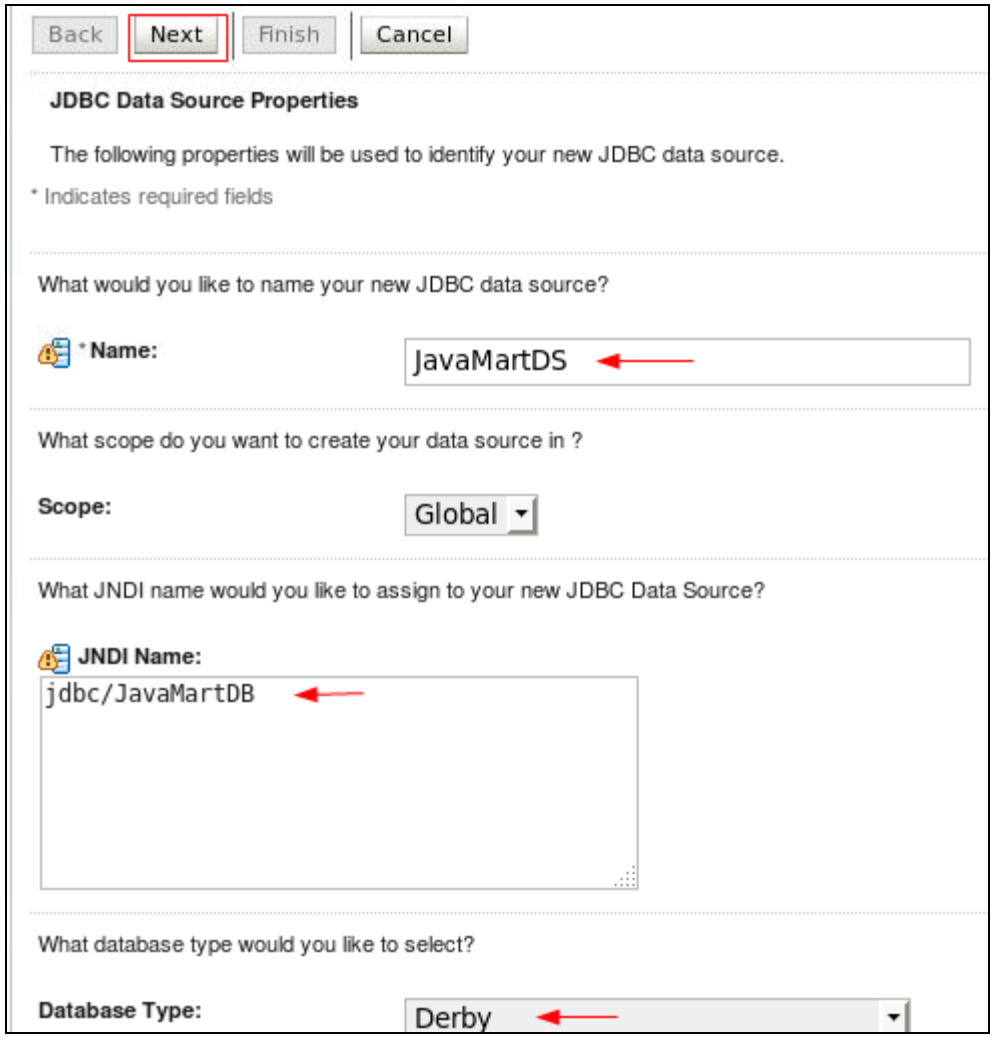
   On the right, notice that the Summary of JDBC Data Sources section appears.

c. Under the Data Sources table heading, click the New drop-down list and select Generic Data Source.

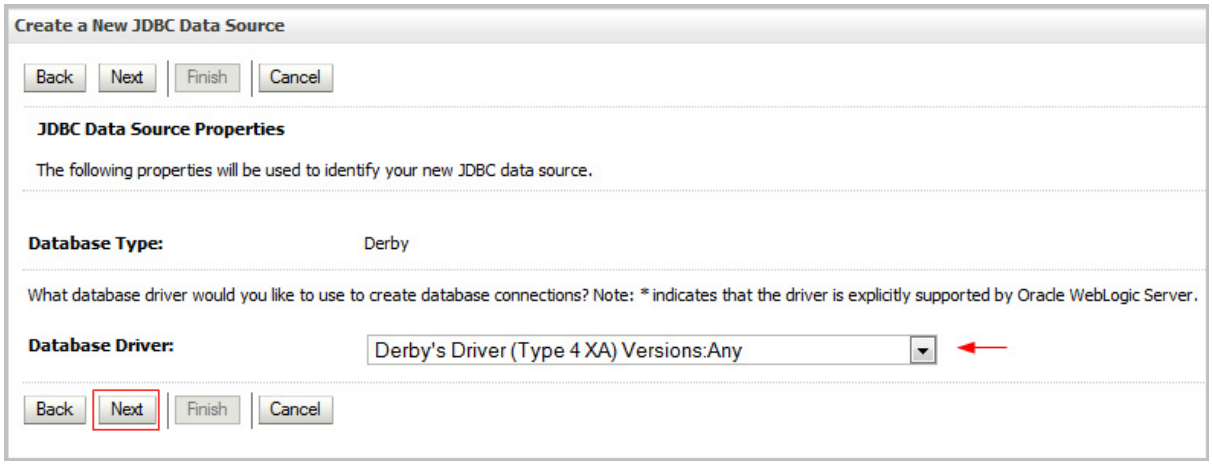


d. On the first page of the Create a New JDBC Data Source Wizard:
   - Enter `JavaMartDS` as the data source name.
   - Enter the JNDI name as `jdbc/JavaMartDB`.

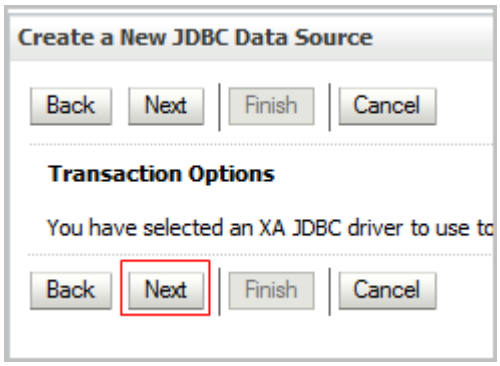
- Select `Derby` from the Database Type drop-down list, and click **Next**.



e. Select `Derby's Driver(Type 4XA) Versions:Any` from the Database Driver drop-down list, and click **Next**.

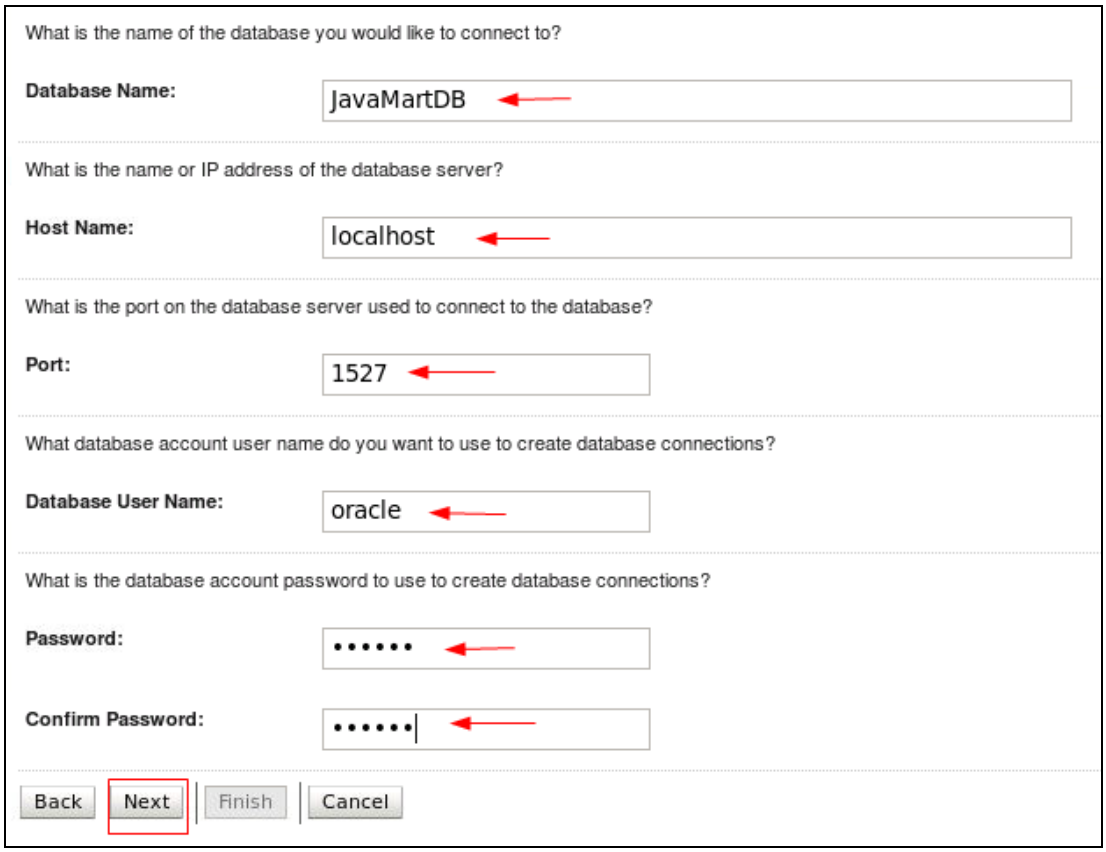


f. On the next page of the wizard, click **Next**.

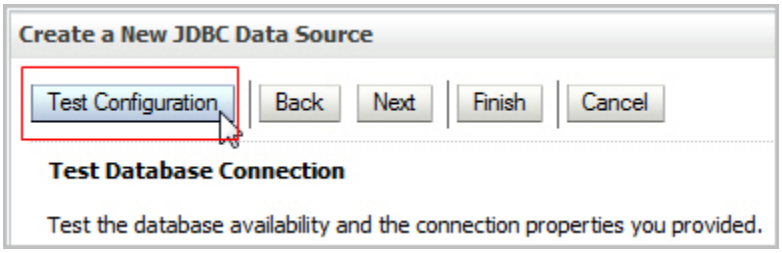


g. On the next page of the wizard, enter the following and click Next:

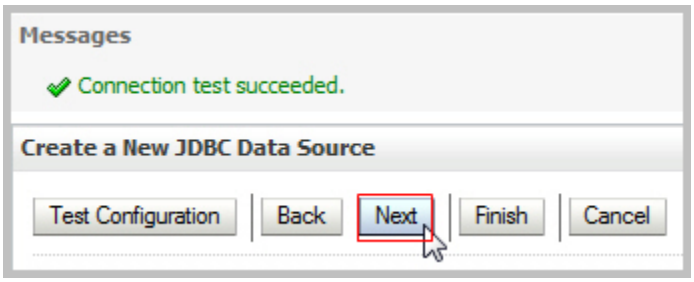
- Database Name: JavaMartDB
- Host name: localhost
- Port: 1527
- Database User Name: oracle
- Password: Enter the password as specified in the security credentials page.
- Confirm Password: Enter the password as specified in the security credentials page.



h. On the next page of the wizard, click **Test Configuration** to check if you can make a connection to the database based on the information that you entered.

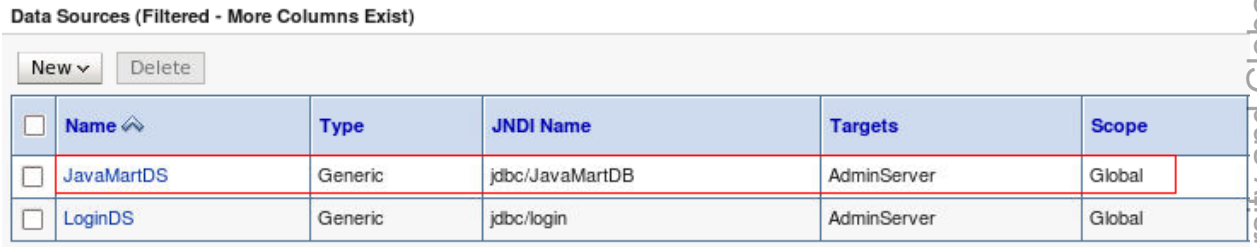


i. The message "Connection test succeeded." is displayed. Click **Next**.



j. On the last page of the wizard, the data source is targeted. Targeting a data source to a server means that the server manages its own instance of the data source. The data source is available as one of the resources of that server. Select **AdminServer** and click **Finish**.

k. In the Data Sources table, click **`JavaMartDS`** to modify its configuration.



l. Under Settings for `JavaMartDS`, click the **Configuration** tab and the **Connection Pool** subtab.

m. Scroll down to the capacity fields, enter the following values, and then click **Save**.
- Initial Capacity: 2
- Maximum Capacity: 15
- Minimum Capacity: 2

3. Complete the required code in the `getAllProducts` method in the `com.example.dao.ProductDAO` class.
- Make sure that the `JavaMartWeb` project is selected.
- Select Window > Action Items to open the Action Items list.

- Double-click the following Action Item to open the `ProductDAO` class at the appropriate place in the file to edit (the `getAllProducts` method).

```
TODO: Use the appropriate SQL statement to get all of the
products from the database
```

Note that the code created an instance of a List object named `prodList` to hold the `Product` objects that will be returned by the method. In a try-with-resources statement, a `Connection` object is created, and then a `Statement` object is created from the `Connection`.

- Below the comment, use an `executeQuery` method to execute the appropriate SQL statement to return all the `Product` items to a `ResultSet` object.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM PRODUCT");
```

- Iterate through the `ResultSet` object and build instances of the `Product` objects by using the table fields that are returned in the `ResultSet` object instance. Then add the `Product` object to `prodList`.

```
while (rs.next()) {
    String id = rs.getString(idCol);
    String name = rs.getString(nameCol);
    double price = rs.getDouble(priceCol);
    String desc = rs.getString(descCol);
    prodList.add(new Product(id, name, price, desc));
}
```

4. Complete the required code for the `getProductById` method in the `com.example.dao.ProductDAO` class.
   - This method is similar to the `getAllProducts` method, except that it should return only a single `Product` object.
   - Below the `TODO` comment in this method, use an `executeQuery` method to execute a SQL statement that returns the fields of a Product row that matches the `product_id` that is passed to the method as an argument.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM PRODUCT WHERE
Product_ID = '" + product_id + "'");
```

Note the single quotation marks around the String `product_id`.

   - Advance to the first record returned and use the same methods that you wrote in the `getAllProducts` method to create an instance of a `Product` object and return it.

```
rs.next();
String id = rs.getString(idCol);
String name = rs.getString(nameCol);
double price = rs.getDouble(priceCol);
String desc = rs.getString(descCol);
prod = new Product(id, name, price, desc);
```

   - Resolve any missing import statements and save the file.
5. Complete the required code for the `updateInventory` method in the `com.example.dao.InventoryDAO` class.

- Double-click the Action Item below to open the `InventoryDAO` class to the method:

```
TODO: Update the inventory with the quantity value passed
```

- Write the appropriate SQL string by using a `Statement` `executeUpdate` method to update the quantity of the records in the database identified by `product_id`.

```
stmt.executeUpdate("UPDATE INVENTORY SET Available = " +
quantity + " WHERE Product_ID = '" + product_id + "'");
```

- Resolve any missing import statements and save the file.

6. Complete the required code for the `updateSalesRecord` method in the `com.example.dao.SalesDAO` class.
   - Double-click the Action Item below to open the `SalesDAO` class to the method:

```
TODO: complete the method to update the sales record with
productSale.getSales_id()
```

   - This SQL statement should be performed with a `PreparedStatement`; so use the `Connection` object to create a `PreparedStatement` object that updates a `Sales` record with the sales date and total sale.

```
PreparedStatement pStmt = conn.prepareStatement("UPDATE Sales
SET Date_Sold = ?, Total_Sale = ? WHERE Sales_id = ?");
```

   - Using the data contained in the `Sales` object that is passed to the method, set the appropriate fields in the prepared statement.

```
pStmt.setInt(3, productSale.getSales_id());
pStmt.setDate(1, new
java.sql.Date(productSale.getSales_date().getTime()));
pStmt.setDouble(2, productSale.getTotal_sale());
```

   - Execute the prepared statement and if the number of rows updated is not equal to 1, throw an exception.

```
if (pStmt.executeUpdate() != 1) {
  throw new SQLException("Failed to update the Sales record");
}
```

   - Resolve any missing import statements and save the file.

7. Complete the required code for the `addProductSalesRecord` method in the `com.example.dao.ProductSalesDAO` class.
   - Double-click the Action Item below to open the `ProductSalesDAO` class to the method:

```
TODO: Insert a new ProductSales record into the database
```

   - Create a `PreparedStatement` that inserts a new record into the `ProductSales` table by using the `ProductSales` object that is passed to the method.

```
PreparedStatement pStmt = conn.prepareCall("INSERT INTO
ProductSales VALUES(?, ?, ?)");
```

   - Use the data passed in the `ProductSales` object to set the appropriate fields in the prepared statement.

```
pStmt.setInt(1, salesRecord.getSales_id());
pStmt.setString(2, salesRecord.getProduct_id());
pStmt.setInt(3, salesRecord.getQuantity_sold());
```

- Execute the SQL statement by using the `executeUpdate` method.

```
pStmt.executeUpdate();
```

- Resolve any missing import statements and save the file.

8. Test the code in your application.
   - Right-click the application and select Run.
   - In the browser that opens, the `JavaMart` servlet runs automatically.
   - Click the Buy button to select several items to add to your cart.

   Note that the quantities do not decrease. This is intentional.
   - Click the Buy Cart button. Note that all the items that you selected were added to the cart.
   - Click Purchase and you see a note that your order will soon ship.
   - Click the Return to Shopping button to return to the main page.

   Note that now the quantities of the items are decremented by the number that you "ordered."
   - Attempt to buy more items than are available by clicking the Buy button for an item more times than the number of available items. (For example, attempt to buy six Small (S) JavaOne T-shirts.) You get an exception, similar to the following:

```
Unable to purchase: JAVAONE2015-S : Desired quantity: 6 :
Available quantity: 5
```

   Note that you can reset the database at any time by re-running the `JavaMartDB.sql` script.

Practices for Lesson 13: Using Transactions in Java EE Environments

Chapter 13

Practices for Lesson 13: Overview

Practices Overview

In these practices, you apply two types of transaction demarcation to CDI beans.

Practice 13-1: Using Bean-Managed Transactions

Overview

In this practice, you use a bean-managed transaction context to wrap the `purchaseCart` method in a transaction.

Assumptions

You have completed the previous practice, the database is running, and the `JavaMart` application is open in NetBeans.

Tasks

1. Simulate two users that are attempting to purchase the same number of an item.
   - Reload the `JavaMart` database by running the `JavaMartDB.sql` script again.
   - Open both Firefox and Opera browsers and enter the URL to the application in both browsers.

     ```
     http://localhost:7001/JavaMartWeb
     ```
   - In Firefox, add two Duke Coffee Mugs and a Java Hat.
   - In Opera, add one Duke Coffee Mug and two Key Rings.
   - Click Buy Cart in both browsers, but do not click the Purchase button.
   - Click the Purchase button in the Firefox browser, and then click the Purchase button in the Opera browser.
   - You should see the following error in the Opera browser window:

     ```
     Unable to purchase: DUKEMUG : Desired quantity: 1 : Available
     quantity: 0
     ```

2. Look at the results in the database.
   - Click the Services tab.
   - Right-click the connection to the JavaMartDB database (`jdbc:derby://localhost:1527/JavaMartDB`) and select Connect.
   - Expand the connection, and then expand ORACLE and Tables.
   - Right-click PRODUCTSALES and select View Data. You should see the following:

#	SALES_ID	PRODUCT_ID	QUANTITY
1	1	JAVAHAT	1
2	1	DUKEMUG	2
3	2	KEYRING0111	2

- Right-click SALES and select View Data. You should see the following:

#	SALES_ID	DATE_SOLD	TOTAL_SALE
1	1	2016-02-04	52.85
2	2	<NULL>	0.0

This indicates a problem. The first purchase was completed successfully, and there is a valid Sales record in the database. However, because there were insufficient quantities of the DUKEMUG product, the second purchase started and then aborted, leaving behind a partial ProductSales record and an empty Sales record. What we really want is for the entire transaction to be atomic—either all the items are available for purchase and the sale is completed, or the entire purchase rolls back.

3. Add bean-managed transactions to the ShoppingCart bean class.
   - After the injected DAO instances, add a `@javax.annotation.Resource` injection of a `UserTransaction` object.

```
@Inject
private SalesDAO salesDAO;
@Resource
private UserTransaction ut;
```

   - Resolve any missing import statements and save the file.
   - In the `purchaseCart` method, just after the `try` statement, start (begin) the transaction context.

```
try {
    ut.begin();
```

   - Currently, when the check on the inventory quantity fails, the method throws a `PurchaseException`. Before the exception is thrown, you need to roll back the transaction.

```
    if (!inventory.subtract(quantity)) {
        ut.rollback();
```

   - Just above the `catch` statement, the transaction should be complete; so commit the transaction.

```
    ut.commit();
} catch (SQLException ex) {
```

   - Notice that NetBeans is indicating that exceptions are not being caught. So you need to alter the `catch` clause to include the exceptions that are needed.

```
} catch (RollbackException | HeuristicMixedException |
         HeuristicRollbackException |
         NotSupportedException | SystemException |
         SQLException ex) {
```

   Tip: NetBeans has a shortcut for adding exceptions. Click in the parentheses for the `catch` statement and press Ctrl + Spacebar. Then select the exception to add from the list that appears. Be sure to separate each exception that you add to the `catch` statement with a pipe character (" | ").

- Resolve any missing import statements and save the file.

4. Repeat your transaction test from step 1.
   - Re-run the `JavaMartDB.sql` script to reset the database.
   - Repeat the steps that you performed in step 1.

   Note that you still get the error message (exception).
   - Repeat step 2 and look at the database tables.
   - Right-click PRODUCTSALES and select View Data. You should see the following:

#	SALES_ID	PRODUCT_ID	QUANTITY
1	1	JAVAHAT	1
2	1	DUKEMUG	2

   - Right-click SALES and select View Data. You should see the following:

#	SALES_ID	DATE_SOLD	TOTAL_SALE
1	1	2016-02-04	52.85

   Now only the single successful transaction was recorded and no additional table rows were created.

Practice 13-2: Using Container-Managed Transactions

Overview

In this practice, you replace the bean-managed transaction code and apply container-managed transactions to the `JavaMart` application.

Tasks

1. Remove the bean-managed transaction code from the `com.example.bean.ShoppingCart` class.
   - Comment out or remove the injected `UserTransaction` object instance.

```
//@Resource
//private UserTransaction ut;
```

   - In the `purchaseCart` method, comment out or remove the code that begins, commits, and rolls back the transaction.

```
try {
    //ut.begin();

        if (!inventory.subtract(quantity)) {
            //ut.rollback();

     //ut.commit();
} catch (...)
```

   - Edit the `catch` statement to catch only `SQLException`.

```
} catch (SQLException ex) {
```

2. Apply the CDI annotation to the `purchaseCart` method to enable container-managed transactions.
   - Annotate the method with `@Transactional` and set the value to `REQUIRED`. Set `rollbackOn` to the `PurchaseException` class.

```
@Transactional(value = Transactional.TxType.REQUIRED,
               rollbackOn = {PurchaseException.class})
public Sales purchaseCart() throws PurchaseException {
```

   - Resolve any missing import statements and save the file.
3. Modify the PurchaseCart servlet.

   When a `PurchaseException` is thrown from the method, the transaction interceptor automatically rolls back the transaction. However, if a transaction is rolled back due to any other exception, the interceptor throws its own TransactionException, which is a RuntimeException type. This rolls the transaction back regardless of where the transaction context is currently running.
   - Open the `com.example.servlet.PurchaseCart` class.

- In the `doPost` method, add a `catch` clause to catch exceptions that are thrown as a result of errors in the transaction other than Purchase exceptions.

```
} catch (PurchaseException ex) {
  logger.log(Level.INFO, ex.getMessage());
  out.println("<h2>" + ex.getMessage() + "</h2>");
} catch (Exception ex) {
  out.println("<h2>Exception completing your transaction</h2>");
  logger.log(Level.INFO, ex.getMessage());
}
```

4. Test that the application still rolls back the transaction if there is insufficient quantity of an item to purchase.
   - Re-run the `JavaMartDB.sql` script to reset the database.
   - Repeat the steps that you performed in practice 13-1, steps 1 and 2.

Practices for Lesson 14: Using the Java Persistence API

Chapter 14

Practices for Lesson 14: Overview

Practices Overview

In these practices, you refactor the JDBC-based DAO objects to JPA and refactor the Data Model POJO classes to be JPA entities.

Practice 14-1: Applying JPA to the JavaMart Application

Overview

In this practice, you refactor the JavaMart application to use Java Persistence API (JPA) entities and entity management.

Tasks

1. Because you will make several changes to the application, temporarily undeploy the current application from WebLogic server.
   - On the Services tab, expand WebLogic Server > Applications > Web Applications.
   - Right-click `JavaMartWeb` and select Undeploy.
2. Create a Persistence Unit and add it to the project.
   - Right-click the project and select New > Other.
   - Select Persistence from Categories and Persistence Unit from File Types, and click Next.
   - Leave `JavaMartWebPU` as Persistence Unit Name.
   - Select `jdbc/JavaMartDB` as Data Source; if not listed, then enter `jdbc/JavaMartDB` as Data Source.
   - Leave Use Java Transaction API selected.
   - Select None as Table Generation Strategy and click Finish.
   - When the `persistence.xml` file opens in the editor, select `None` as Shared Cache Mode.
   - Make sure that Include All Entity Classes in “JavaMartWeb” Module is selected and save the `persistence.xml` file.
3. Convert the POJOs that represent the data model into JPA entities by using the appropriate annotations, beginning with the Product model class.
   - Open the `com.example.model.Product` class in the editor.
   - Recall that the characteristics of an entity class are as follows:
     - Class is annotated with `@Entity`.
     - Class implements `Serializable`.
     - Class has a no-arg constructor.
     - Annotations exist on either class fields or accessor methods for the primary key.
   - Start by adding an `@Entity` annotation to the class.

```
@Entity
public class Product {
```

   - Modify the class signature to implement `Serializable`.

```
public class Product implements Serializable {
```

- Add a no-arg constructor to the class.

```
public Product() {
}
```

Tip: An easy way to do this is by right-clicking in the editor window and selecting Insert Code, or pressing the Alt + Insert keys, and then selecting Constructor from the list. Do not select any fields, and you will get a no-arg constructor.

- Identify the primary key by annotating the field with `@Id`. In this class, the primary key is `product_id`.

```
@Id
private String product_id;
```

- Resolve any missing import statements and save the file.

Note that all the JPA annotations should have import statements from the `javax.persistence` package.

4. Convert the `Inventory` class into an entity class.
   - Open the `com.example.model.Inventory` class.
   - Make the same changes as you made to the Product class. Note that `product_id` is the primary key for this class.
   - In the Product class, the class field names match the entity field names (column names in the table). However, in the Inventory class, the class field is named `quantity_on_hand`, whereas the corresponding entity field is named "`AVAILABLE`". Use the `@Column` annotation to create the match between the two.

```
@Column(name="AVAILABLE")
private int quantity_on_hand;
```

   - Resolve any missing import statements and save the file.
5. Convert the `ProductSales` class into an entity class.
   - Open the `com.example.model.ProductSales` class.
   - Make the same changes as you made to the Product class.
   - Like the `Inventory` class, there is a name mismatch in one of the fields: The `quantity_sold` field should be mapped to the column name "`QUANTITY`".

```
@Column(name = "QUANTITY")
private int quantity_sold;
```

   - Resolve any missing import statements and save the file.
6. Convert the `Sales` class into an entity class.
   - Open the `com.example.model.Sales` class.
   - Make the same changes as you made to the Product class. Note that `sales_id` is the primary key for this class.
   - This primary key is generated by the database—the key identity is created automatically every time a new row is created. Use the `@GeneratedValue` annotation to specify that this is the case.

```
@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
private int sales_id;
```

- This class also has a Date field. Recall that Java Date objects and SQL date representations are different; so you must apply an annotation to indicate that this is a `Temporal` type.

```
@Temporal(TemporalType.DATE)
private Date date_sold;
```

- Resolve any missing import statements and save the file.

7. Edit the Data Access Object (DAO) classes to use JPA instead of JDBC, starting with the `ProductDAO` class.
   - Open the `com.example.dao.ProductDAO` class in the editor.
   - Remove or comment out the resource injection, and then all column names.

```
//    @Resource(lookup = "jdbc/JavaMartDB")
//    private DataSource dataSource;
//    private final String idCol = "PRODUCT_ID";
//    private final String nameCol = "PROD_NAME";
//    private final String priceCol = "PRICE";
//    private final String descCol = "PROD_DESC";
```

   Tip: To comment out a large block of code in NetBeans, select the code with the mouse and click the Ctrl + Shift + C keys. You can use the same sequence of keys to uncomment code.

   - Use `PersistenceContext` to inject an `EntityManager`, `em`.

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

   - Comment out or remove all the code for the `getAllProducts` method, except for the `return` statement.

```
//List<Product> prodList = new ArrayList<>();
//try (Connection conn = dataSource.getConnection()) {
//  Statement stmt = conn.createStatement();
//  ResultSet rs = stmt.executeQuery("SELECT * FROM PRODUCT");
//  while (rs.next()) {
//     String id = rs.getString(idCol);
//     String name = rs.getString(nameCol);
//     double price = rs.getDouble(priceCol);
//     String desc = rs.getString(descCol);
//     prodList.add(new Product(id, name, price, desc));
//  }
//}
```

   - The method uses a `TypedQuery` to create a query that returns all the Product entities ("`SELECT p FROM Product p`") and then uses the query to get a List of Products and returns the list.

```
TypedQuery<Product> query
  = em.createQuery("SELECT p FROM Product p", Product.class);
List<Product> prodList = query.getResultList();
```

```
return prodList;
```

- Comment out or remove all the code for the `getProductById` method, except the `return` statement.

```
//Product prod = null;
//try (Connection conn = dataSource.getConnection()) {
//  Statement stmt = conn.createStatement();
//  ResultSet rs = stmt.executeQuery("SELECT * FROM PRODUCT
WHERE Product_ID = '" + product_id + "'");
//  rs.next();
//  String id = rs.getString(idCol);
//  String name = rs.getString(nameCol);
//  double price = rs.getDouble(priceCol);
//  String desc = rs.getString(descCol);
//  prod = new Product(id, name, price, desc);
//}
```

- The method should use the `find` method to manage the desired Product, and then return it.

```
Product prod = em.find(Product.class, product_id);
return prod;
```

- Resolve any missing import statements and save the file.

8. Edit the `com.example.dao.ProductSalesDAO` class.
   - Comment out or remove the `DataSource` and replace it with an injection of an `EntityManager`.

```
//    @Resource(lookup = "jdbc/JavaMartDB")
//    private DataSource dataSource;
@PersistenceContext
private EntityManager em;
```

   - Comment out or remove all the code in the two methods (except the `return` statement in the `getProductSalesBySalesID` method).
   - The `addProductSalesRecord` method should use a persist method to create a new managed `ProductSales` entity (the one that is passed to the method).

```
public void addProductSalesRecord(ProductSales salesRecord)
throws SQLException {
  em.persist(salesRecord);
}
```

   - The `getProductSalesBySalesID` method uses a `TypedQuery` to return all the `ProductSales` entities that have a specific `sales_ID` by using the query string "`SELECT p from ProductSales p WHERE p.sales_id = <id>`" and to return the list of `ProductSales` entities as a result.

```
public List<ProductSales> getProductSalesBySalesID(int sales_ID)
throws SQLException {
  String queryString = "SELECT p from ProductSales p WHERE
p.sales_id =" + sales_ID;
```

```
  TypedQuery<ProductSales> query = em.createQuery(queryString,
ProductSales.class);
  List<ProductSales> pSales = query.getResultList();
  return pSales;
}
```

- Resolve any missing import statements and save the file.

9. Edit the `com.example.dao.InventoryDAO` class.
   - Comment out or remove the `DataSource` and replace it with an injection of an `EntityManager`.
   - Comment out or remove all the code in the two methods (except the `return` statement in the `getInventory` method).
   - In the `getInventory` method, use the `find` method to return an Inventory entity by its primary key.

```
Inventory inventory = em.find(Inventory.class, product_id);
return inventory;
```

   - In the `updateInventory` method, create a new Inventory object and use the `merge` method to update the matching managed entity.

```
public void updateInventory(String product_id, int quantity)
throws SQLException {
  Inventory inventory = new Inventory(product_id, quantity);
  em.merge(inventory);
```

   - Resolve any missing import statements and save the file.
10. Edit the `com.example.dao.SalesDAO` class.
    - Comment out or remove the `DataSource` and replace it with an injection of an `EntityManager`.
    - Comment out or remove all the code in the three methods.
    - In the `createSalesRecord` method, create a new empty Sales record and use the `persist` method to add it to the managed pool. What the entity manager does is create a new entity by inserting it into the database, thus creating a new primary key. Use the `flush` method to update the managed entity (from the database) with the primary key that was created by the database and return the updated Sales entity to the caller.

```
Sales newSalesRecord = new Sales();
em.persist(newSalesRecord);
em.flush();
return newSalesRecord;
```

    - In the `updateSalesRecord` method, use the `merge` method to update the managed Sales entity.

```
public void updateSalesRecord(Sales productSale) throws
SQLException {
  em.merge(productSale);
```

    - In the `removeSalesRecord` method, use the `remove` method to remove the managed entity from the database.

```
public void removeSalesRecord(Sales productSales) throws
SQLException {
  em.remove(productSales);
```

- Resolve any missing import statements and save the file.

11. Rebuild, deploy, and test the application.
    - Right-click the project and select Clean and Build.
    - Right-click the project again and select Deploy.

      Test the functionality of the code by using the steps from practice 13-1 titled “Using Bean-Managed Transactions” (from step 1).

Practices for Lesson 15: Using Bean Validation

Chapter 15

Practices for Lesson 15: Overview

Practices Overview

In these practices, you use Bean Validation to check that the fields of the Product and Inventory entities meet specific criteria before they are persisted.

Practice 15-1: Using Bean Validation with JPA

Overview

In this practice, you use a new servlet to attempt to create new `Product` records in the database and use Bean Validation to prevent erroneous records.

Ordinarily, field validation is best done as close as possible to the user interface—in this case, at the browser level. Field validation can be done by using complex rules in JavaScript on the browser page, but it can also be done with much less complexity by using JavaServer Faces and Bean Validation.

However, because this is a course about Java back-end technologies, in this practice, you attempt to create new `Product` and `Inventory` records directly from a servlet form.

Tasks

1. Part of the solution for this practice has been provided for you. Make the following changes to your `JavaMartWeb` application:
   - Open the `com.example.dao.ProductDAO` class and add a new method for adding a new Product to the database.

```
public void addNewProduct(Product newProd) {
  em.persist(newProd);
  em.flush();
}
```

   - Resolve any missing import statements.
   - Copy `InventoryException.java` from `/home/oracle/labs/resources/beanValidation` to the `com.example.util` package.
   - Copy `ProductManager.java` from `/home/oracle/labs/resources/beanValidation` to the `com.example.beans` package.
   - Copy `UpdateInventory.java` from `/home/oracle/labs/resources/beanValidation` to the `com.example.servlet` package.

   To copy the files, you can use the system file explorer or drag the files from the Favorites tab to the Projects tab.

   These files allow you to attempt to create new Product records in the database through an HTML form.

   - Open the `com.example.beans.ProductManager` class. The `addNewProduct` method is invoked with a `Product` entity object and an integer `quant` field, which is the quantity of the item to add to the database. Notice that this method runs inside of a transaction context. If a `ConstraintViolationException` is thrown, the various `ConstraintViolations` that are received are added as suppressed exceptions to `InventoryException`, which is thrown from the method.
2. Annotate the `com.example.model.Product` entity class with Bean Validation annotations.

- The `product_id` field is the primary key for the entity and must not be null and should not be an empty ("") string, so annotate the field with the NotNull and Size annotations. Add a message that will be provided if the validation constraint is violated.

```
@Id
@NotNull(message = "The primary key may not be null")
@Size(min = 1, message = "The primary key, Product ID, cannot be
empty")
private String product_id;
```

- The product name field, `prod_name`, must also not be null and should be a minimum of 10 characters in length. Use the appropriate constraints.

```
@NotNull(message = "The product name field may not be null")
@Size(min = 10, message = "The product name must be at least 10
characters")
private String prod_name;
```

- The price for a product should never be less than 5 dollars, so set a constraint on the `price` field that prevents this from happening.

```
@Min(value = 5, message="The price field must be greater than
5")
private double price;
```

- Resolve any missing import statements and save the file.

3. Annotate the `com.example.model.Inventory` entity class with Bean Validation annotations.
   - The `product_id` field is the primary key for the entity and must not be null, so annotate this field appropriately.

```
@Id
@NotNull(message="The primary key may not be null")
private String product_id;
```

   - The `quantity_on_hand` field should never be less than 0, so use a constraint on this field.

```
@Column(name="AVAILABLE")
@Min(value=0, message="The Inventory quantity must be greater
than or equal to 0")
private int quantity_on_hand;
```

   - Resolve any missing import statements and save the file.
4. Test the application.
   - Run the application, and in the URL in the browser, enter the servlet that is used to update the inventory.

```
http://localhost:7001/JavaMartWeb/UpdateInventory
```

- Click the Submit button (all empty fields). You should see the following response:

```
Price and quantity must be numbers Product: Product id: Name:
Description: Price: $0.00 Quantity: 0


Exception: Bean Validation constraint(s) violated while
executing Automatic Bean Validation on callback
event:'prePersist'. Please refer to embedded
ConstraintViolations for details.
Value: Message: The primary key, Product ID, cannot be empty
Value: Message: The product name must be at least 10 characters
Value: 0.0 Message: The price field must be greater than 5
```

The first part of the response includes a view of the `Product` object created and the quantity of the product to be stored in inventory.

On the next line is the message from `ConstraintViolationException`, indicating that the `Product` entity was checked before the entity was persisted to the database, and it resulted in several `ConstraintViolation` instances.

The next lines contain the values that created the constraint violation and the messages received as a result of the violation.

- Click the Try Again button, enter the following data in the fields, and click Submit:

Field	Value
Product ID	1
Product Name	too short
Unit Price	4.99
Description	Blank
Quantity	1

Which entity fields do you think will throw constraint violations?

- Click the Try Again button, enter the following data in the fields, and click Submit:

Field	Value
Product ID	JAVAJACKET-L
Product Name	Leather Java Jacket (Ladies)
Unit Price	104.50
Description	Black leather Java Jacket embroidered with Java logo on the back and Duke on the front left. Order specific ladies size.
Quantity	-1

Which entity fields do you think will throw constraint violations?

- Click the Try Again button and fill in the fields again with the preceding data.
  Tip: Double-click in each field to see the data cached for the field. Replace `-1` with `10` and click Submit. You should see that you successfully added a record to the database.
- In another browser window, open the URL to the JavaMart page and you should see the new record entered there.

```
http://localhost:7001/JavaMartWeb/JavaMart
```

- Return to the browser window that has the `UpdateInventory` servlet.
- Fill in the fields again for the JAVAJACKET-L product. Enter any non-zero number as the quantity and click Submit. What do you expect will happen?

  This time you should see a transaction rolled back exception message:

```
Unable to process inventory update, transaction rolled back -
see log for more details
```

Practice 15-2: Using a Validator with Bean Validation

Overview

In this practice, you inject an instance of a `Validator` to test instances of `Product` before attempting any interaction with JPA.

In the previous practice, you attempted to add entities directly from a form. However, a more likely scenario is to create new product entities from a file that is loaded and run as a batch update of the inventory in the store.

Tasks

1. Open the `com.example.beans.ProductManager` class and add a method that uses a `Validator` to test instances of `Product` objects.
   - Inject an instance of a Validator as a class field.

```
@Inject
private Validator validator;
```

   - Add a method `validateProduct` that takes an instance of a `Product` and returns a `List` of strings. The list returned contains the string messages from any `ConstraintViolations`.

```
public List<String> validateProduct(Product p) {
}
```

   - In the method, declare a local variable, `messages`, of type `List` that is assigned a new `ArrayList`.

```
  List<String> messages = new ArrayList<>();
```

   - Call the `validate` method on the injected `Validator` instance, passing in the `Product` instance `p`. This method returns a set of `ConstraintViolations`.

```
  Set<ConstraintViolation<Product>> violations =
validator.validate(p);
```

   - Iterate through the set, creating a String `error` message that contains the value that caused the violation and the violation message. Add the error string to the list and log it.

```
  for (ConstraintViolation violation : violations) {
    String error = "Value: " + violation.getInvalidValue() +
                   " Message: " + violation.getMessage();
    messages.add(error);
    logger.log(Level.INFO, error);
  }
  return messages;
```

   - Resolve any missing import statements and save the file.
2. Copy the `BatchUpdateInventory` class from the `/home/oracle/labs/resources/beanValidation` folder to the `com.example.servlet` class.
   - Open this file.
   - Notice that this class defines several `Product` instances in an array:

```
private final Product[] newProducts = {
  // Failed record: empty id, too short product name
  new Product("", "Empty ID", 12.34, "This is a product with no
ID"),
  // Failed record: null product name
  new Product("DUKEHAT", null, 25.99, "A baseball cap with a
Duke logo!"),
  // Failed record: price too low
  new Product("JAVAKIDS", "Kids Java Logo T-Shirt", 4.99, "A
kid-sized T-shirt with a Java Logo"),
  // Passed record
  new Product("JAVAJACKET-L", "Leather Java Jacket (Ladies)",
104.90, "Black leather Java Jacket embroidered with Java logo on
back and Duke on the front left. Order specific ladies size.")
};
```

- The `processRequest` method iterates through this array, invokes the `validateProduct` method with each `Product` instance, and displays the results on the browser page. Note that this code is not attempting to persist these entities at this time.

3. Run the project by using the new `BatchUpdateInventory` servlet.
   - Enter `http://localhost:7001/JavaMartWeb/BatchUpdateInventory` in a browser to execute the servlet.
   - Note that the specific Bean Validation constraint violations are listed on the page.

   This method of validating objects before invoking a persistence context is a good way to evaluate objects that may be sent by other business objects through another back-end mechanism (such as JMS).

Practices for Lesson 16: Batch Processes

Chapter 16

Practices for Lesson 16: Overview

Practice Overview

In this practice, you create a batch process to load sales from a remote Representational State Transfer (REST) web service in JavaScript Object Notation (JSON) format, convert the JSON objects to Java objects, and store them in a database by using Java Database Connectivity (JDBC).

About REST

REST is an architecture style for networked applications that uses the HTTP protocol. It is an alternative for other architectures that are used to connect to machines over the network. Instead of using CORBA, RPC, or SOAP, REST connects two machines by using simple HTTP calls.

Java EE implements the REST architecture by using JAX-RS, which allows you to create RESTful services and HTTP clients to make REST service calls by using HTTP requests.

This practice uses JAX-RS to create a client to make a request to a service that provides the sales data.

About JSON

JSON is a format to represent JavaScript objects, arrays, and values as strings. It is one of the preferred ways of sending and receiving data from REST web services.

JSON allows you to represent objects as key-value pairs and arrays as sequential lists of items.

Values can also be represented as JSON, thus allowing you to create all sorts of objects by using strings, numbers, and Booleans.

JSON is often compared to XML because both can be used to represent structured data. The main difference between JSON and XML is that JSON contains only data without any schema information. Therefore, JSON has no direct validation mechanism.

Java EE provides an API to read and write JSON data. This practice uses the JSON Processing (JSON-P) API to parse the sales data from the REST service.

About the Service Architecture

You create the batch process that consumes the sales data from a remote server, processes it, and stores it in the database.

The Sales service is located at the following URL:

```
http://localhost:7005/BatchServices/rest/sales
```

Try opening a browser and opening the link. You see a big JSON file containing all the sales data that must be saved.

The data is composed of 10,000 registries, which should be stored in the database.

Note that data is randomly generated. This practice checks that you insert the right number of items in the database.

Sales data is provided in an array of JSON objects. Each object contains the sale information, as well as a smaller array of items that were sold for that particular sale.

```
REST service → JSON Data

JSON Data = [sale, sale, sale]

Sale = {"itemCount":2, "status":1, "timestamp":1423267984457,
"items":[Item, Item], "total":120}

Item =  {"productId":"JAVAONE2015-XS","productName":"JavaOne
2015 T-
Shirt","productPrice":39.95,"productCount":1,"productPrice":39.9
5}
```

The sale and item objects that are presented are examples. Actual data might differ.

You can check the quantity of sales data stored in the database by using your web browser and navigating to the following URL:

```
http://localhost:7005/BatchServices/
```

Practice 16-1: Batch Processing Sales Information into the Database

Overview

In this practice, you create the batch process that stores the sales information in the database.

Prerequisites

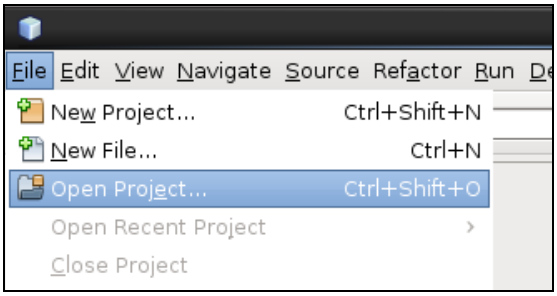
You must complete practice 12-2 titled “Writing Data Access Objects with JDBC.” The JavaMart data source must be configured with the JNDI name: `jdbc/JavaMartDB`.

Tasks

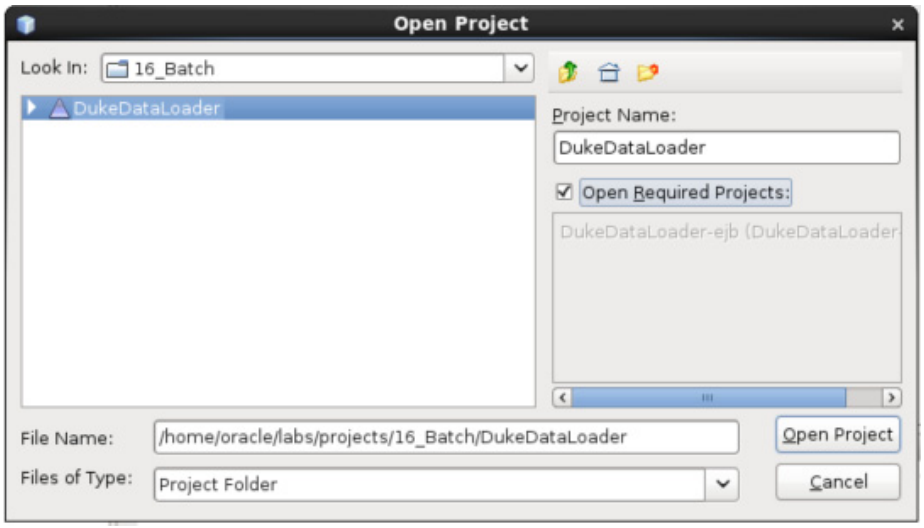
Batch Configuration

In this section, you configure the batch job.

1. Open the `DukeDataLoader` project for this lesson in NetBeans:
   a. Select File > Open Project.

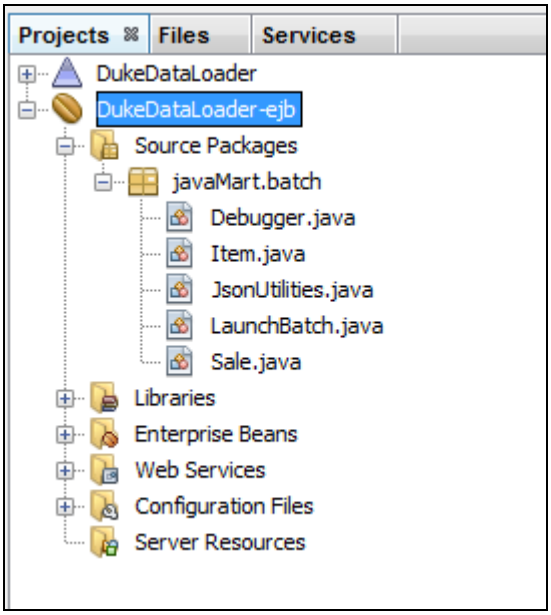


   b. Browse to the `/home/oracle/labs/projects/16_Batch` folder, click the `DukeDataLoader` project, and select the Open Required Projects check box. Click Open Project.

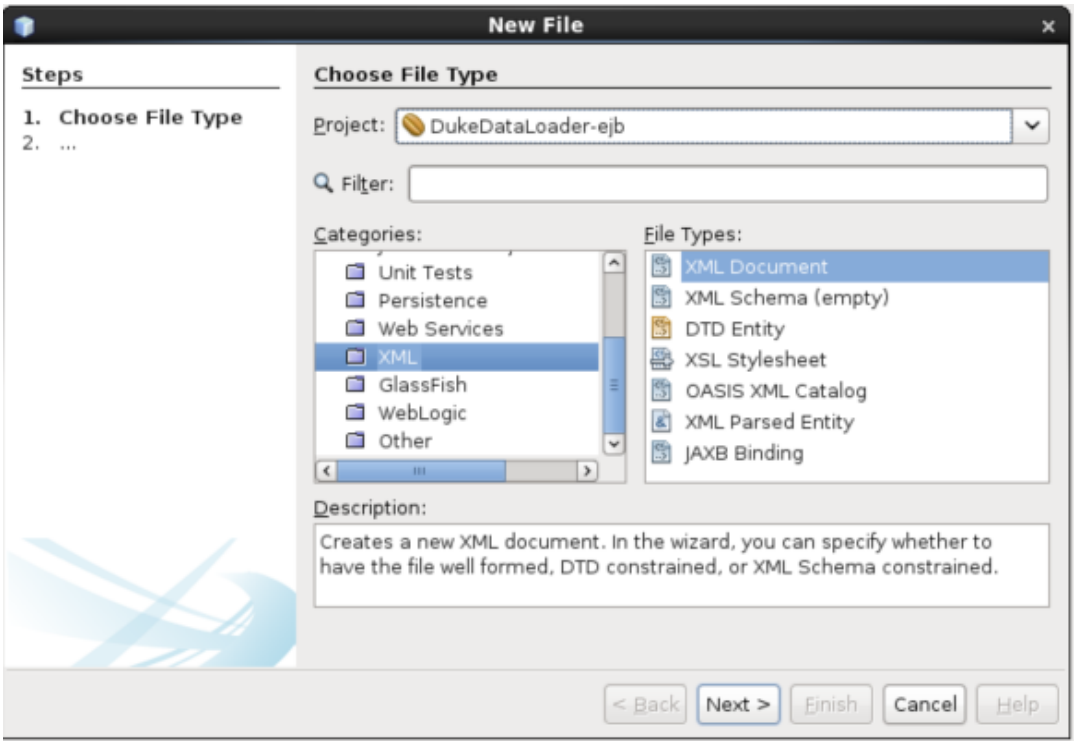


   After a short delay, you should see two projects open on the Projects tab.

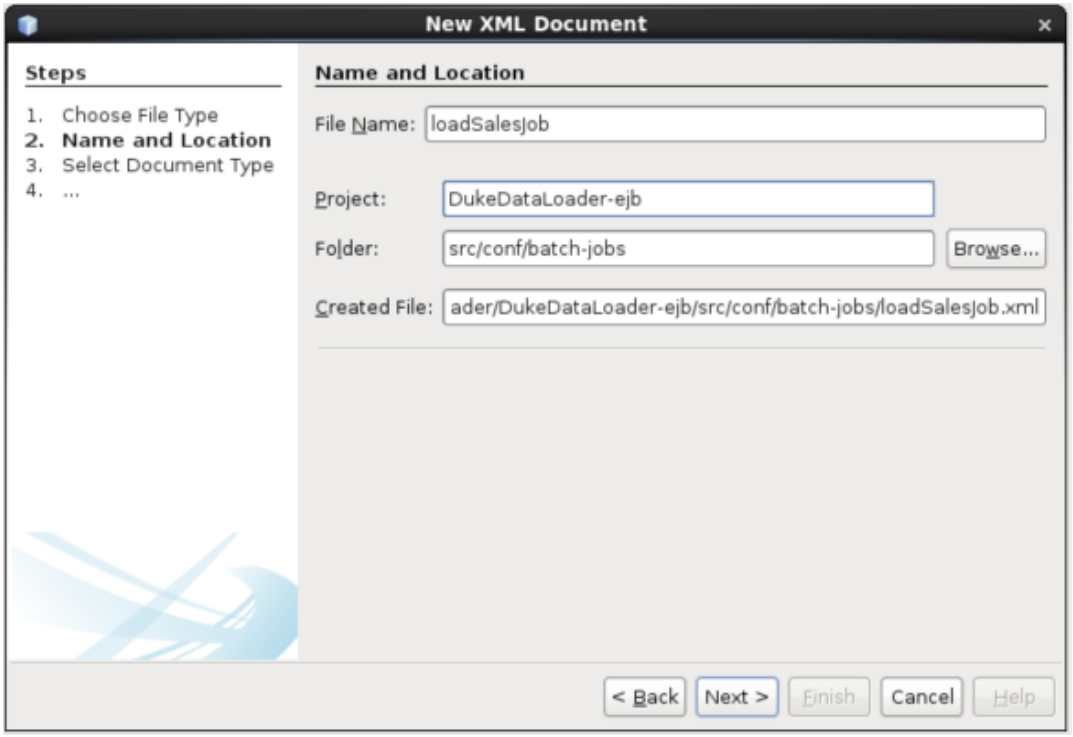
You will be working with the `DukeDataLoader-ejb` project.



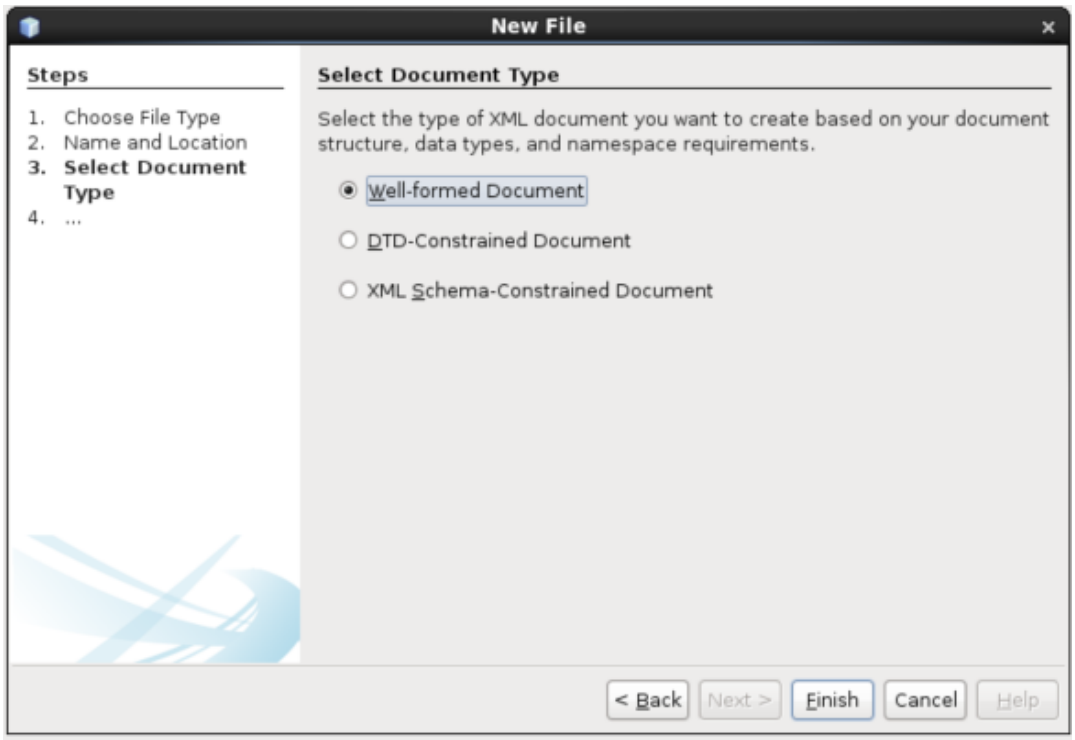
2. Right-click the project and select New > Other.
3. Select the XML category and the XML Document file type. Click Next.



4. Set the file name to `loadSalesJob` and the folder to `src/conf/batch-jobs`. Click Next.



5. Select “Well-formed Document” and click Finish.



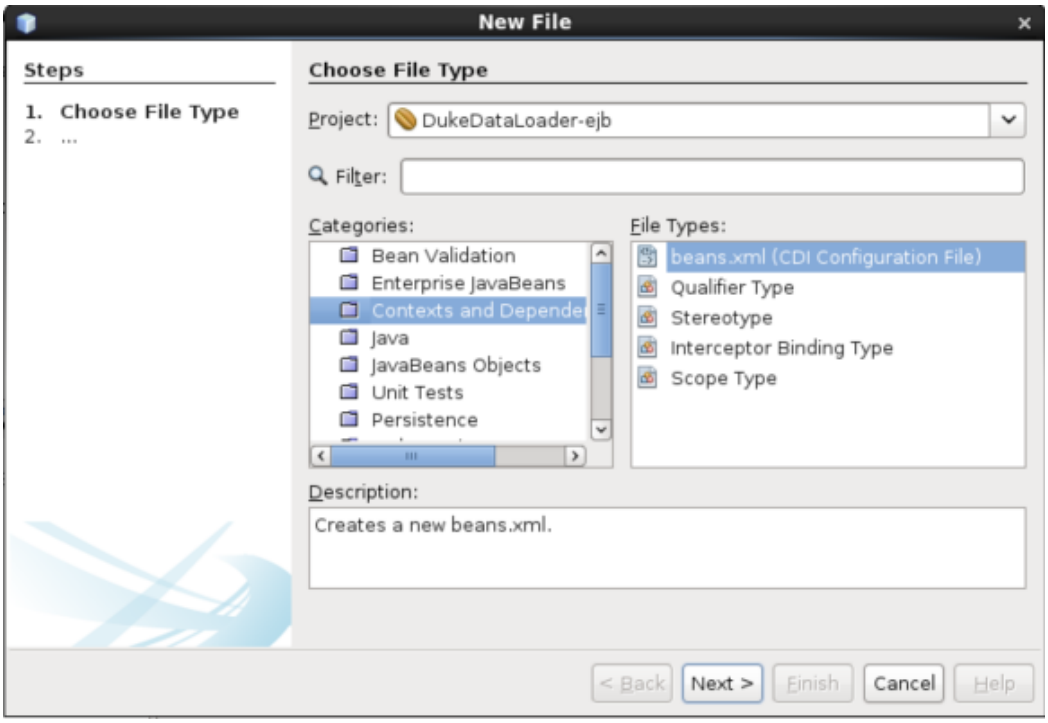
This creates a new empty XML file that you modify to create a batch job.

6. Open the `loadSalesJob.xml` file and replace its contents with the following XML Batch job descriptor:

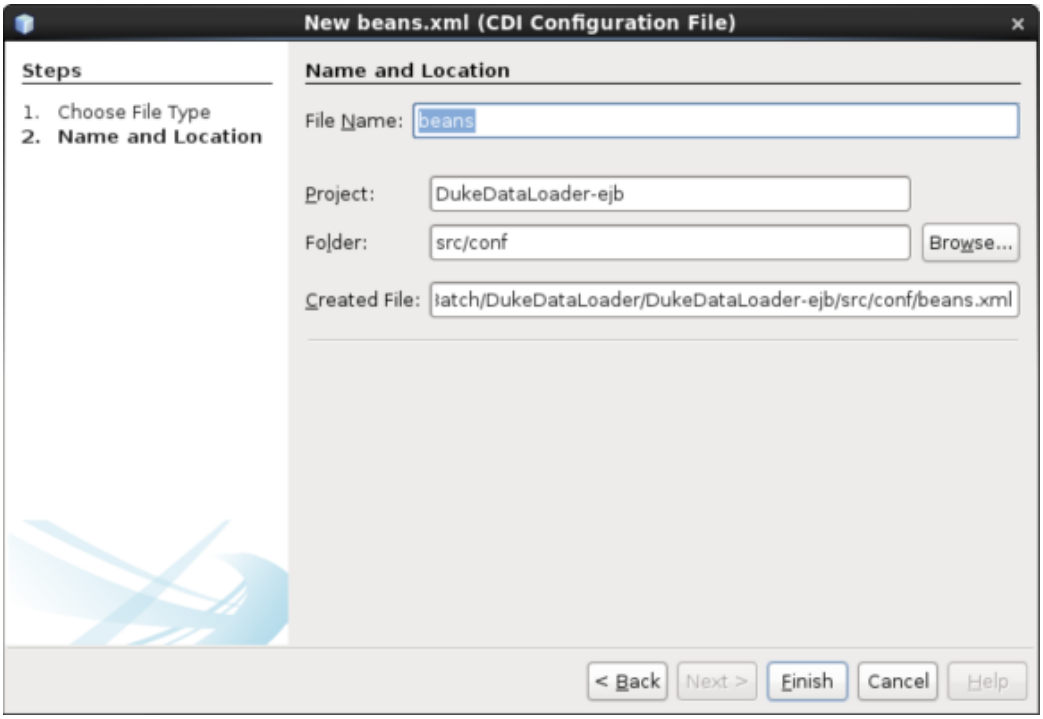
```
<job id="loadSales" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
version="1.0">
    <step id="loadSalesStep">
        <chunk item-count="100">
            <reader ref="salesReader"></reader>
            <processor ref="salesProcessor"></processor>
            <writer ref="salesWriter"></writer>
        </chunk>
    </step>
</job>
```

To enable injection of CDI beans in EJB projects, you must create a `beans.xml` CDI configuration file.

7. Right-click the project and select New > Other.
8. Select the Contexts and Dependency Injection category and the beans.xml (CDI Configuration File) file type. Click Next.



9. Click Finish.



10. Open the `beans.xml` file. Change bean-discovery-mode to all.

```
bean-discovery-mode="all"
```

This concludes the setup of the batch job.

The Item Reader

In this section, you create the Batch Reader that gets the data from the REST service, parses it, and generates the chunks that are later processed.

Using the JAX-RS 2.0 Client API

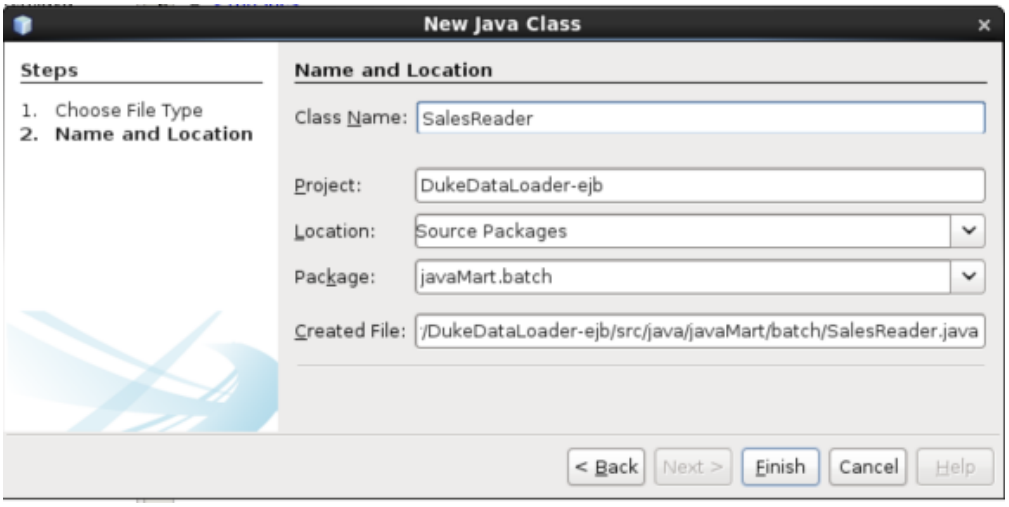
JAX-RS 2.0 includes a client API that makes HTTP requests to remote REST services.

To make a request, perform the following:

- Create a new Client instance by using the `ClientBuilder` class.
- Get a `Target` from the client by using the destination URL.
- Optionally set any headers in the target.
- Make a request to the target, including the expected media type to be returned from the service.
- Optionally make the request asynchronous with the `async` method.
- Call the HTTP method on the request that you want to make, for example, get, post, put, or delete.
- If the request is asynchronous, you get a `Future` object; if it is synchronous, you get the result of the HTTP call.

Because all the calls return a new object by using the builder pattern, they can be chained one after another. You make use of this technique in this practice.

11. Right-click the `javaMart.batch` package and select New > Java Class.
12. Set Class Name to `SalesReader` and click Finish.



13. Add the following imports:

```
import java.io.InputStream;
import java.io.Serializable;
import java.util.concurrent.TimeUnit;
import javax.batch.api.chunk.AbstractItemReader;
import javax.inject.Named;
import javax.json.Json;
import javax.json.stream.JsonParser;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
```

```
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
```

14. Add the `javax.inject.Named` annotation to the class.
15. Extend the `javax.batch.api.chunk.AbstractItemReader` class.

```
@Named
public class SalesReader extends AbstractItemReader {
```

16. Add a `javax.json.stream.JsonParser` parser instance variable. With this, you can parse the JSON input from the remote service.

```
private JsonParser parser;
```

17. Override the `public void open(Serializable checkpoint) throws Exception` method.

```
@Override
public void open(Serializable checkpoint) throws Exception {
```

18. To make a REST call by using the JAX-RS 2.0 Client API, you need to create a `Client` instance by using the `newClient` method from the `ClientBuilder` class.

```
Client client = ClientBuilder.newClient();
```

19. Make the JAX-RS request by using the client object methods in the following order:
    - Set the target to `http://localhost:7005/BatchServices/rest/sales`.
    - Set the request to `MediaType.APPLICATION_JSON`.
    - Call the `async` method.
    - Call the `get` method to execute the REST call.
    - Call the `get` method with a timeout of 2 seconds.

    Use the following code for reference:

```
Response response =
client.target("http://localhost:7005/BatchServices/rest/sales")
    .request(MediaType.APPLICATION_JSON)
    .async()
    .get()
    .get(2, TimeUnit.SECONDS);
```

20. Create the parser instance object by creating a `JsonParser` with the `Json` class. Read the following token from the `JsonParser` to start parsing the JSON structure:

```
parser =
Json.createParser(response.readEntity(InputStream.class));
parser.next();
```

    This concludes the open method.

The JSON-P Streaming API

Because the data returned by the REST call is large, the JAX-RS Client call returns an `InputStream` that can be read sequentially to get the data.

To parse the raw data into JSON, you use the JSON-P streaming API, which is designed to read long streams of JSON data.

All JSON-P objects are created by using the `Json` class's methods. The `JsonParser` reads from an input stream sequentially. Each read operation in the `JsonParser` returns a `JsonParser.Event`.

Events describe what is being read from the JSON stream: start of an object, start of an array, a key, a value, and so on. The actual value of the event is retrieved from the parser.

In this practice, a convenience method is included in the `JsonUtilities` class that reads a `JsonObject` from the stream.

If you want to learn more about the JSON-P API, look at the code of the `JsonUtilities` class. You can also refer to the Java EE7 tutorial at: http://docs.oracle.com/javaee/7/tutorial/.

21. Override the `readItem` method.

```
@Override
public Object readItem() throws Exception {
```

22. Inside the `readItem` method, use the `JsonParser` to read the next `Json` event.

```
JsonParser.Event event = parser.next();
```

23. If the event is equal to `Event.START_OBJECT`, return the result of invoking the `JsonUtilities` class `readObject` method; otherwise, return `null`.

```
if (event == JsonParser.Event.START_OBJECT) {
  return JsonUtilities.readObject(parser);
}
return null;
```

The `JsonUtilities.readObject` method uses the parser to read the next JSON object in the parser. See the code in the `readObject` method for an example on how to use the Streaming API and the Object Model API.

This concludes the `readItem` method.

24. Override the `close` method and close the parser.

```
@Override
public void close() throws Exception {
  parser.close();
}
```

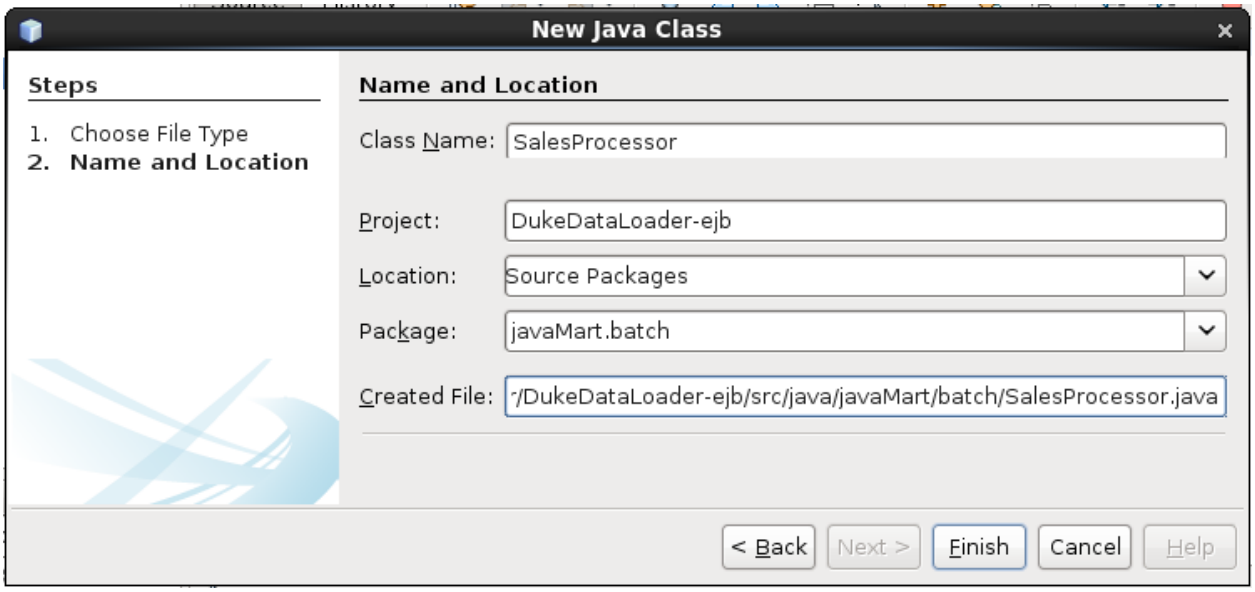
This concludes the `SalesReader` class.

The Item Processor

The item processor converts the Java Json objects into the `Sale` and `Item` objects.

> The JSON-P Object Model API
>
> JSON-P includes an Object Model API to work with JSON data. For every element in the JSON data, there is a Java Object to represent it. The Java interfaces that represent the JSON elements are JsonArray, JsonObject, JsonNumber, JsonString, and JsonValue. Note that all the interfaces inherit from JsonValue.
>
> Due to the nature of JSON data, you need to know the data type of the element that you want to retrieve. Elements from an array are retrieved by index, whereas elements in an object are retrieved by key name. The value type inside the array or object defines the method that you use to retrieve it.
>
> Iterating a JsonArray object returns JsonValue objects. You need to cast these objects to their real type.

25. Right-click the `javaMart.batch` package and select New > Java Class.
26. Set Class Name to `SalesProcessor` and click Finish.



27. Add the following imports:

```
import java.util.Date;
import javax.batch.api.chunk.ItemProcessor;
import javax.inject.Named;
import javax.json.JsonObject;
import javax.json.JsonValue;
```

28. Add the `javax.inject.Named` annotation to the class.
29. Implement the `javax.batch.api.chunk.ItemProcessor` interface.

```
@Named
public class SalesProcessor implements ItemProcessor {
```

30. Override the `Object processItem(Object item)` method.

```
@Override
public Object processItem(Object item) throws Exception {
```

31. Cast the `item` object to a `JsonObject` named `jsonSale`.

```
JsonObject jsonSale = (JsonObject) item;
```

32. Create a new `sale` object.

```
Sale sale = new Sale();
```

33. Set the date of the `sale` object to a new `DateObject` with the time stamp from the `jsonSale` object. Get the time stamp from the `jsonSale` object by using `jsonSale.getJsonNumber("timestamp").longValue()`.

```
sale.setDate(new
Date(jsonSale.getJsonNumber("timestamp").longValue()));
```

34. Set the `status` of the `sale` object to the `status` property of the `jsonSale` object.

```
sale.setStatus(jsonSale.getInt("status"));
```

35. Set `sale total` to the `total` attribute from the `jsonSale` object.

```
sale.setTotal(jsonSale.getJsonNumber("total").doubleValue());
```

36. Iterate the `items` property inside the `JsonSale` object as a `JsonArray`. Note that iterating a Json array returns a collection of `JsonValue` objects.
37. Cast the `jsonItemVal` object to a `JsonObject` named `jsonItem`.
38. Create a new `Item` object and perform the following:
    - Set the `id` to the `productId` property of the `jsonItem` object.
    - Set the `quantity` to the `productCount` property of the `jsonItem` object.
    - Add the `item` to the items list inside the `sale` object.

```
for (JsonValue jsonItemVal : jsonSale.getJsonArray("items")) {
  JsonObject jsonItem = (JsonObject) jsonItemVal;
  Item saleItem = new Item();
  saleItem.setId(jsonItem.getString("productId"));
  saleItem.setQuantity(jsonItem.getInt("productCount"));
  sale.getItems().add(saleItem);
}
```

39. Finally return the `sale` object.

```
return sale;
```

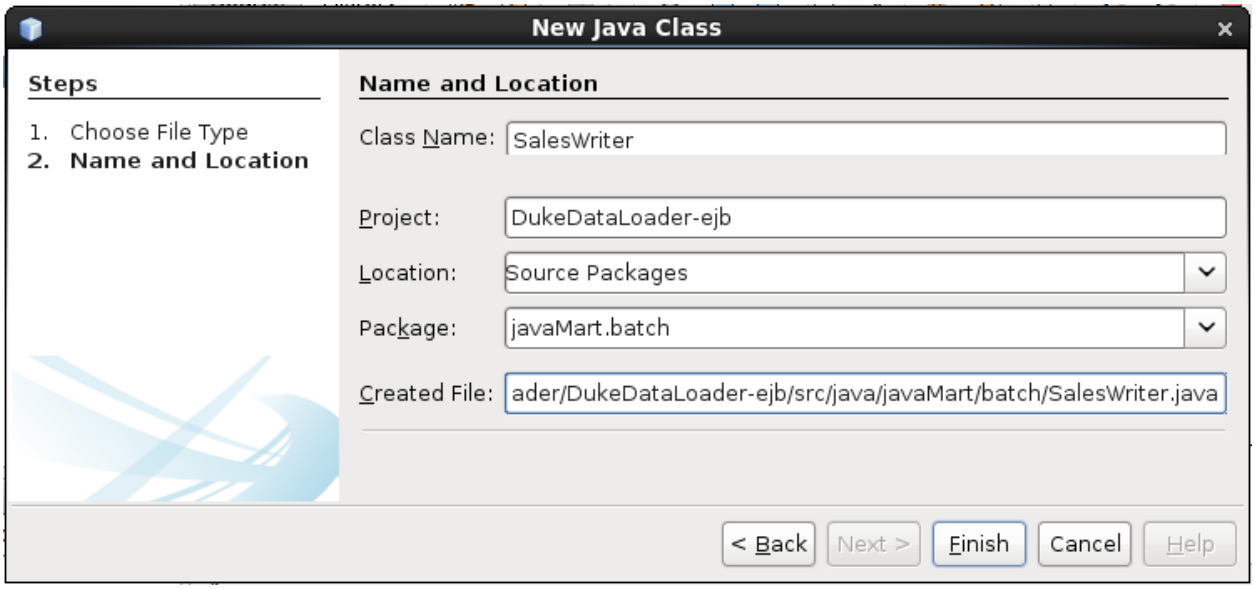
This concludes the `processItem` method and the `SalesProcessor` class.

The Item Writer

The item writer stores the processed sale objects in the database.

You use the same data source from practice 12-2 titled “Writing Data Access Objects with JDBC.” You need to complete the practice or configure the data source in GlassFish.

40. Right-click the `javaMart.batch` package and select New > Java Class.
41. Set the Class Name to `SalesWriter` and click Finish.



42. Add the following imports:

```
import java.io.Serializable;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Timestamp;
import java.util.List;
import javax.annotation.Resource;
import javax.batch.api.chunk.AbstractItemWriter;
import javax.inject.Named;
import javax.sql.DataSource;
```

43. Add the `javax.inject.Named` annotation to the class.
44. Extend the `javax.batch.api.chunk.AbstractItemWriter` class.

```
@Named
public class SalesWriter extends AbstractItemWriter {
```

45. Inject the `DataSource` resource with the JNDI name `jdbc/JavaMartDB`.

```
@Resource(lookup = "jdbc/JavaMartDB")
private DataSource dataSource;
```

46. Create an unassigned `Connection` instance variable.

```
private Connection connection;
```

47. Create an unassigned `PreparedStatement` variable named `insertItem`.

```
private PreparedStatement insertItem;
```

48. Create an unassigned `PreparedStatement` variable named `insertSale`.

```
private PreparedStatement insertSale;
```

49. Override the `open(Serializable checkpoint) throws Exception` method and perform the following:
    - Get a new connection from `dataSource` and set it to the `connection` instance variable.
    - Set `insertSale` to a new prepared statement in the connection by using the following query: `INSERT INTO Sales(Date_Sold, Total_Sale) VALUES(?,?)` Add the `Statement.RETURN_GENERATED_KEYS` parameter.
    - Set `insertItem` to a new prepared statement in the connection by using the following query: `INSERT INTO ProductSales(Sales_ID, Product_ID, Quantity) VALUES(?,?,?)`

```
@Override
public void open(Serializable checkpoint) throws Exception {
  connection = dataSource.getConnection();
  insertSale = connection.prepareStatement("INSERT INTO Sales(
Date_Sold, Total_Sale) VALUES(?,?)",
Statement.RETURN_GENERATED_KEYS);
  insertItem = connection.prepareStatement("INSERT INTO
ProductSales(Sales_ID, Product_ID, Quantity) VALUES(?,?,?)");
}
```

50. Override the `writeItems(List<Object> items) throws Exception` method.
    - Iterate the `items` list.
    - Cast each element to a `sale` object.
    - Call the `insert` method with the sale object (you create this method in the following steps) and get the `id` from the sale.
    - Iterate the items inside `sale.getItems()`.

- For each of the items, call the `insert(saleId, item)` method. (You create this method in the following steps.)

```
@Override
public void writeItems(List<Object> items) throws Exception {
  for (Object o : items) {
    Sale sale = (Sale) o;
    long saleId = insert(sale);
    for (Item item : sale.getItems()) {
      insert(saleId, item);
    }
  }
}
```

51. Create the `long insert(Sale sale) throws SQLException` method. You use the `insertSale` prepared statement to insert elements in the `Sales` table.
    - Call the `setTimestamp` method in the `insertSale` object with the following parameters: `1, new Timestamp(sale.getDate().getTime())`
    - Call the `setDouble` method in the `insertSale` object with the following parameters: `2, sale.getTotal()`
    - Call the `executeUpdate` method in the `insertSale` object.
    - Get a `generatedKeys ResultSet` object from the `insertSale.getGeneratedKeys` method.
    - Advance the `generatedKeys` object by using the `next` method in it.
    - Return the result of the `getLong(1)` method from the `generatedKeys` object.

```
private long insert(Sale sale) throws SQLException {
    insertSale.setTimestamp(1, new
Timestamp(sale.getDate().getTime()));
    insertSale.setDouble(2, sale.getTotal());
    insertSale.executeUpdate();
    ResultSet generatedKeys = insertSale.getGeneratedKeys();
    generatedKeys.next();
    return generatedKeys.getLong(1);
}
```

52. Create the `void insert(long saleId, Item item) throws SQLException` method. You use the `insertItem` prepared statement to insert elements in the `ProductSales` table.
    - Call the `setLong` method in the `insertItem` object with the following parameters: `1, saleId`
    - Call the `setString` method in the `insertItem` object with the following parameters: `2, item.getId()`
    - Call the `setInt` method in the `insertItem` object with the following parameters: `3, item.getQuantity()`
    - Call the `executeUpdate` method in the `insertItem` object.

```
private void insert(long saleId, Item item) throws SQLException
{
    insertItem.setLong(1, saleId);
    insertItem.setString(2, item.getId());
    insertItem.setInt(3, item.getQuantity());
    insertItem.executeUpdate();
}
```

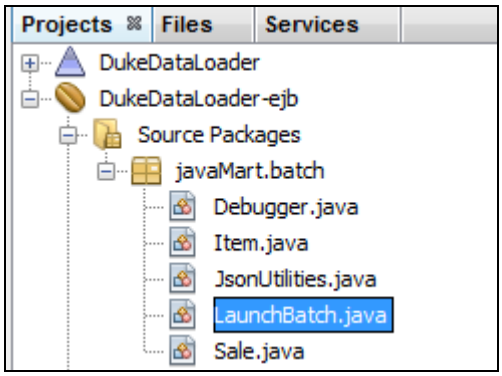
53. Override the `void close() throws Exception` method.
    - Close the `insertSale` object.
    - Close the `insertItem` object.
    - Close the `connection` object.

```
@Override
public void close() throws Exception {
    insertSale.close();
    insertItem.close();
    connection.close();
}
```

This concludes the `SalesWriter` object.

Running the Batch Process by Using a Web Service

54. Open the `LaunchBatch` class inside the `javaMart.batch` package.



55. Add the following Imports:

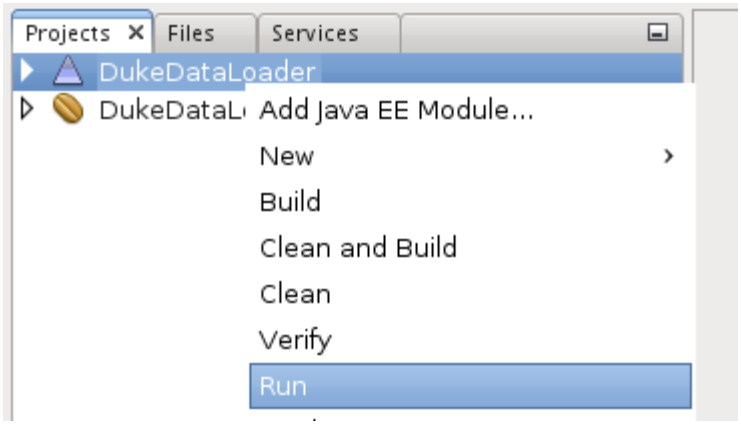
```
import java.util.Properties;
import javax.batch.operations.JobOperator;
import javax.batch.operations.JobSecurityException;
import javax.batch.operations.JobStartException;
import javax.batch.runtime.BatchRuntime;
```

56. Replace the contents of the `runBatch` method with the following:

```
@WebMethod(operationName = "run")
public String runBatch() {
    try {
        JobOperator jo = BatchRuntime.getJobOperator();
        long jobId = jo.start("loadSalesJob", new Properties());
        return "Launched job: " + jobId;
    } catch (JobStartException | JobSecurityException ex) {
        return "error: " + ex.getMessage();
    }
}
```

Testing the Application

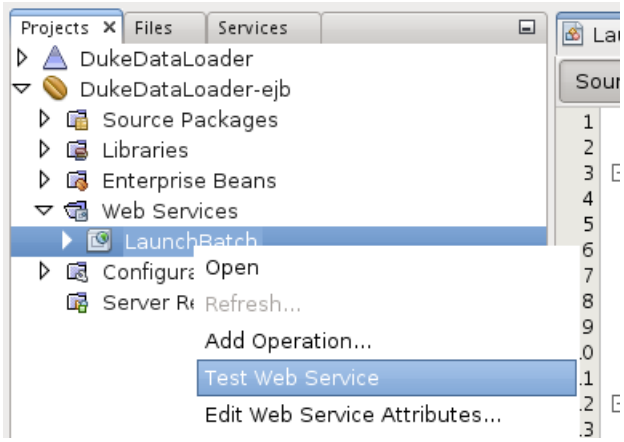
57. Right-click the `DukeDataLoader` project and select **Run** from the context menu.



After a while, the server starts and deploys your application.

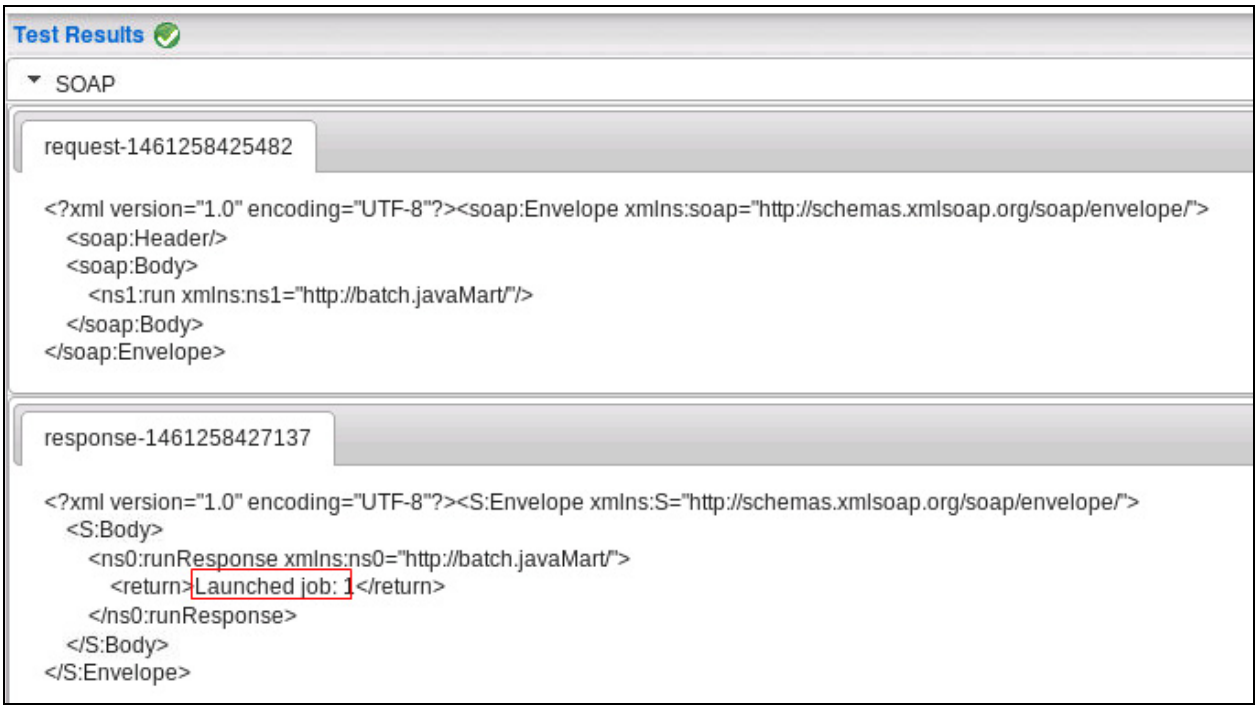
58. Open a web browser and go to `http://localhost:7005/BatchServices`. This page lets you see how much sales data is registered in the database. When you complete and run the batch service, this number increases by 10,000 each time the batch is run.

59. In NetBeans, open the `WebServices` node inside the `DukeDataLoader-ejb` project.
60. Right-click the `LaunchBatch` Web Service and select Test Web Service.

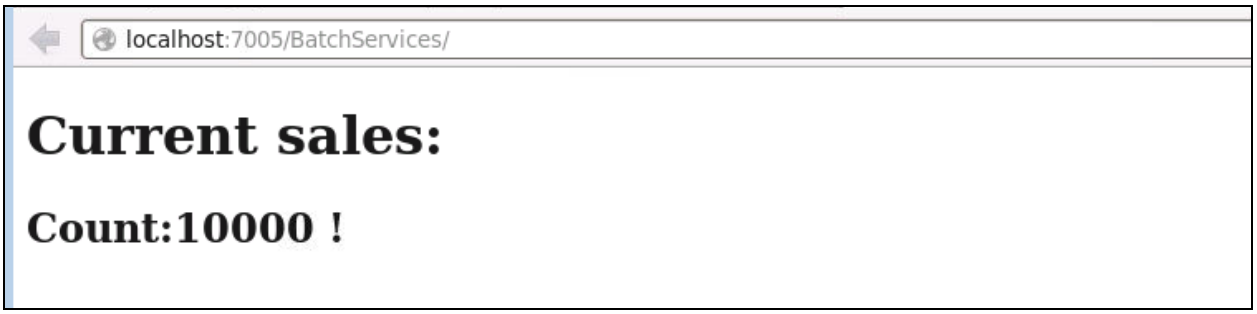


61. A browser opens with the web service tester. Click **Test**.

62. Click **Invoke** at the bottom of the page. The result of the invocation is displayed.



63. Open the browser window that has the Sales count page: http://localhost:7005/BatchServices. Refresh the page. The sales count increases gradually until it reaches 10,000.



If you already inserted sales before running the batch process, the final number might be different, but the increment after running the batch process must be a sales count of 10,000.

Practices for Lesson 17: Securing Enterprise Beans

Chapter 17

Practices for Lesson 17: Overview

Practices Overview

In these practices, you develop and configure enterprise beans to secure them. Using security annotations, you configure authorization directly in the source code.

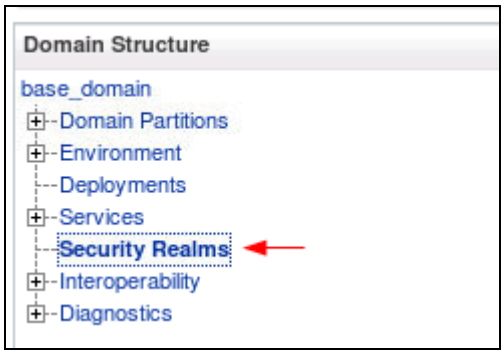
Practice 17-1: Creating a Security Group on the Application Server

Overview

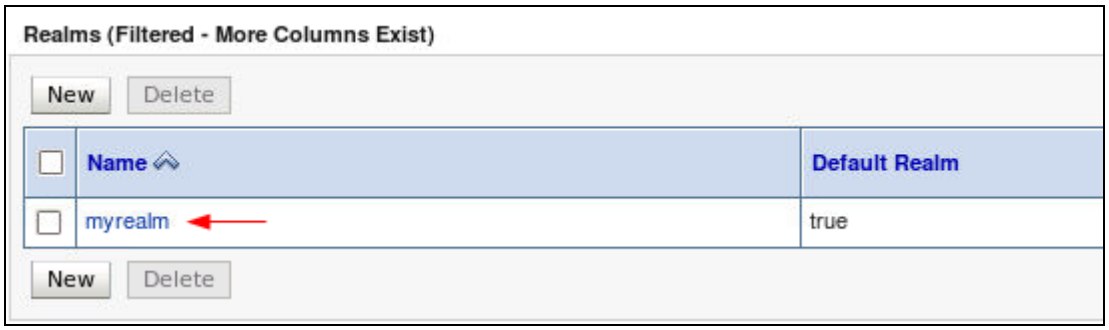
In this practice, you create a group `store_users` that has access to the enterprise bean. You create the `admin` user in the `store_users` group.

Tasks

1. Create a user account by using the WebLogic console.
   a. Open the `http://localhost:7001/console/` URL in a web browser and log in. The username is `weblogic`. Enter the password as specified in the security credentials page.
   b. In the Domain Structure panel, click Security Realms.



   c. In the Realms table, click the `myrealm` realm.

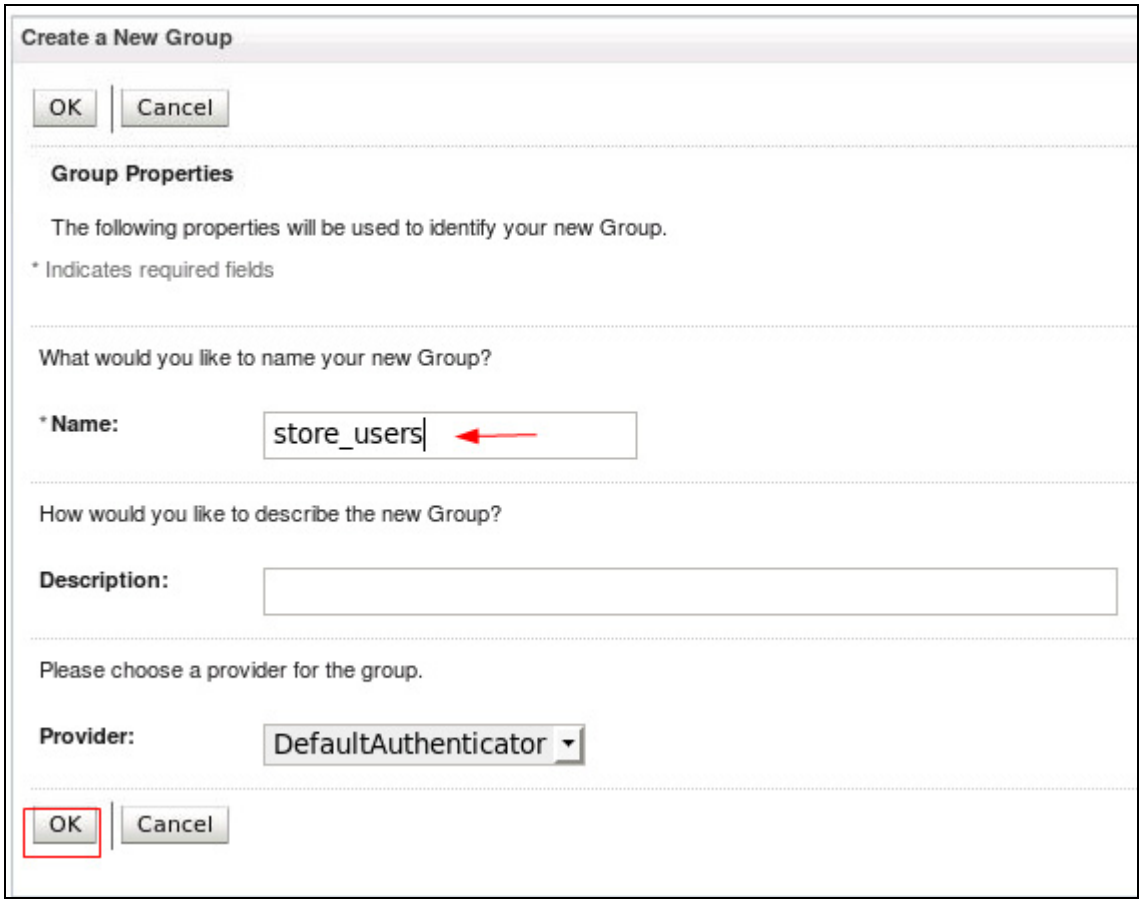


   d. Click the Users and Groups tab.

e. Create a new group:

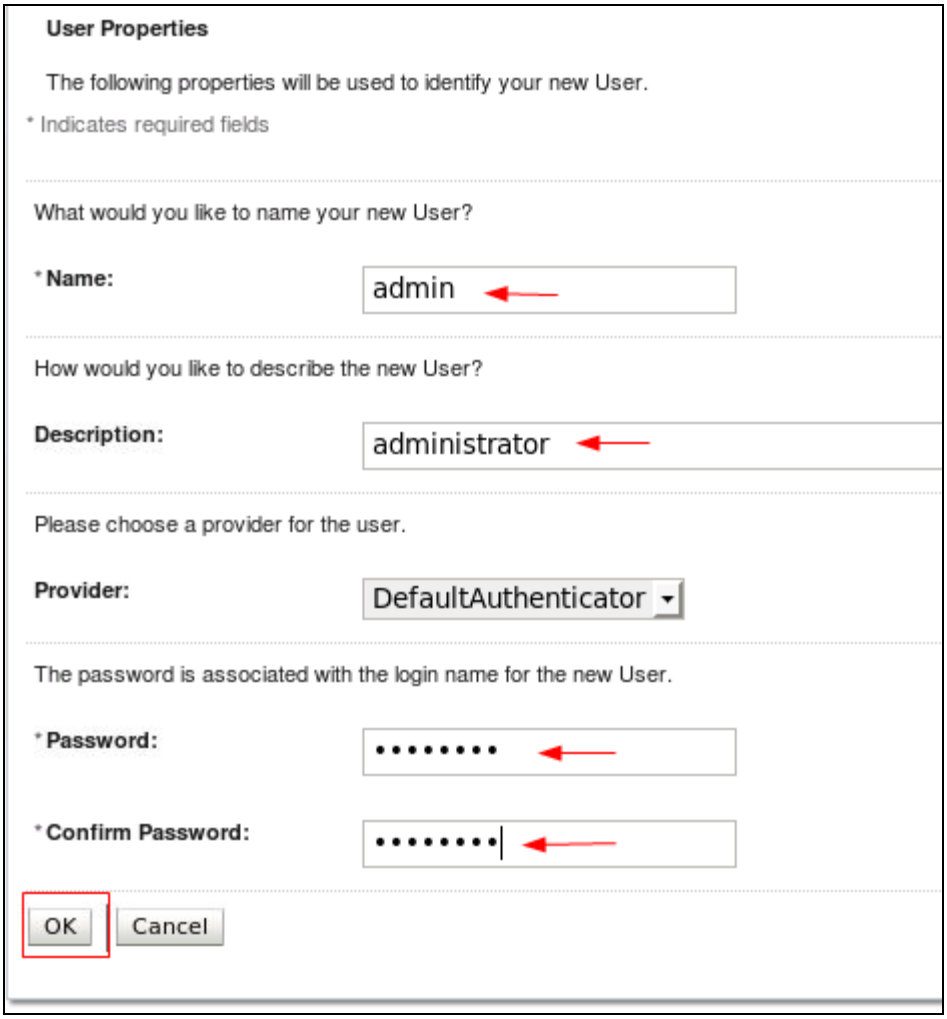
- On the Groups subtab, click New.

- A dialog box appears, as shown in the following image. Enter the Name as `store_users` and click OK.



f. Create a new user:

- On the Users subtab, click New.
- Enter the following:
  1) Name: `admin`
  2) Description: `administrator`
  3) Enter the password. Refer to the security credentials page.
  4) Click OK.

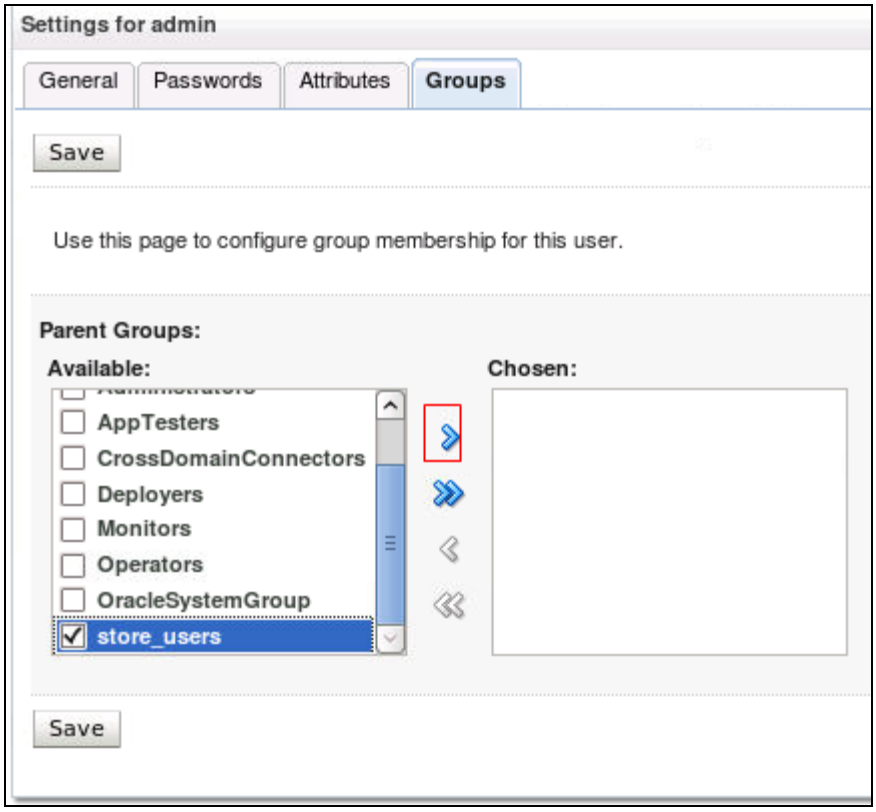


g. Add the user `admin` to the `store_users` group:

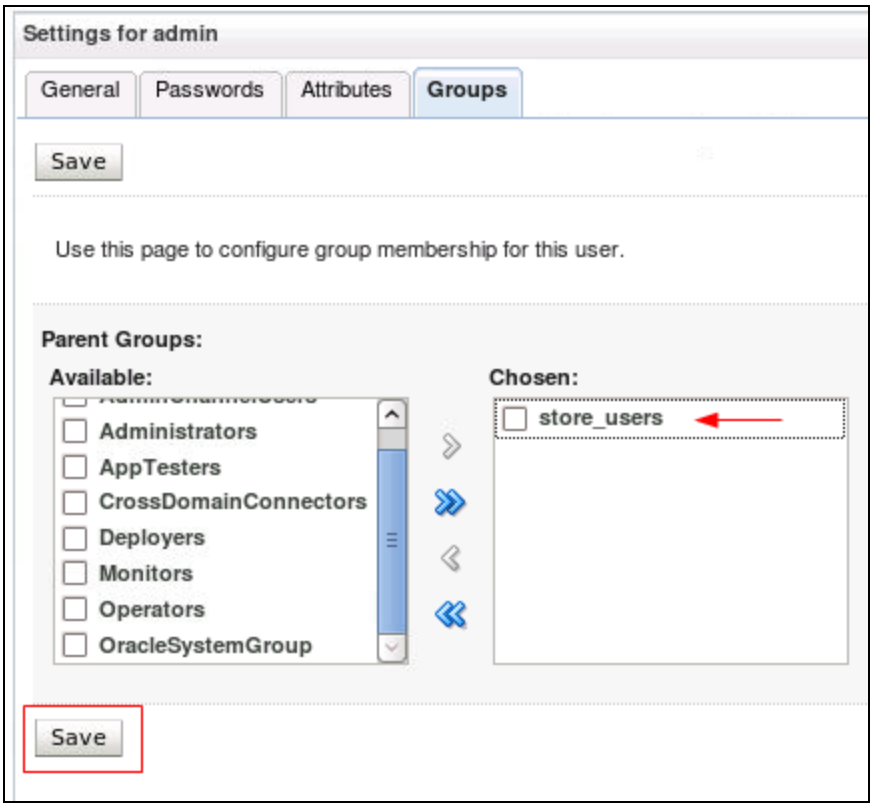
- On the Users subtab, click `admin`.
- Select the Groups tab.

All the groups available in the WebLogic Authentication provider's database appear in the Available list box.

- In the Available list box, select `store_users`.
- Click the highlighted arrow to move the group from the Available list box to the Chosen list box.



- Click Save.



Practice 17-2: Creating a Java Class for the Remote Interface

Overview

In this practice, you create a Java class library project that contains the remote interfaces for the session bean.

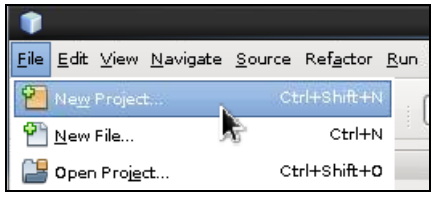
The compiled class library JAR is added to the classpath of the EJB module and the application client that is used to call the session bean.

Assumptions

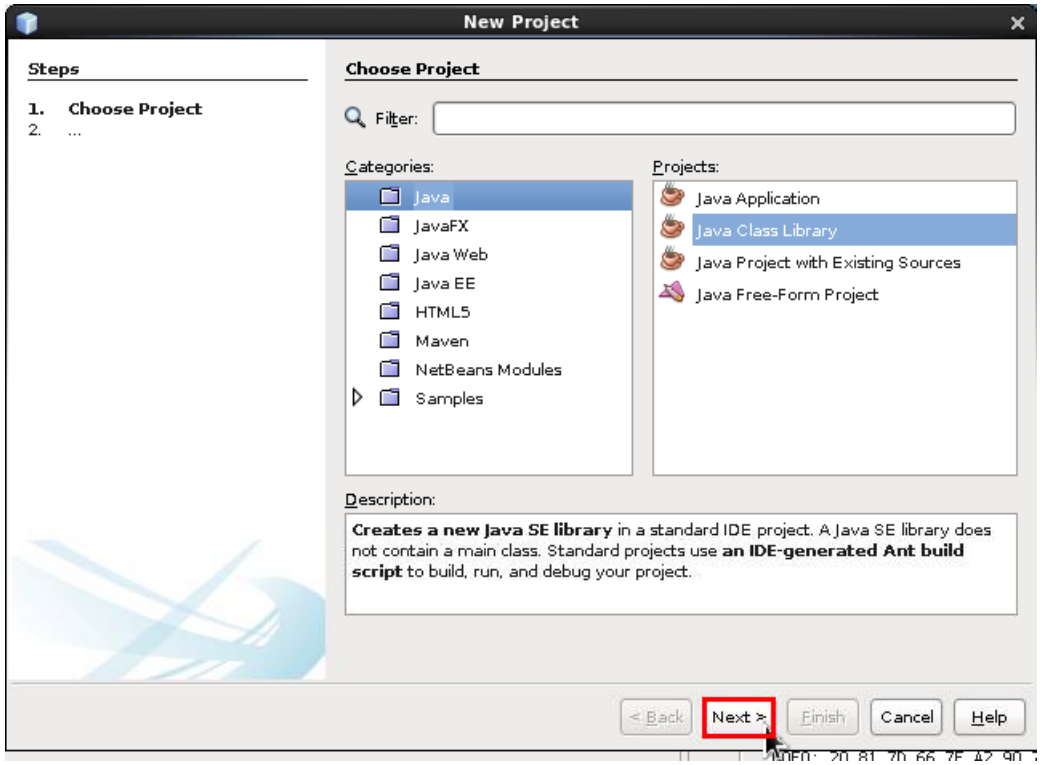
You have finished Practice 17-1.

Tasks

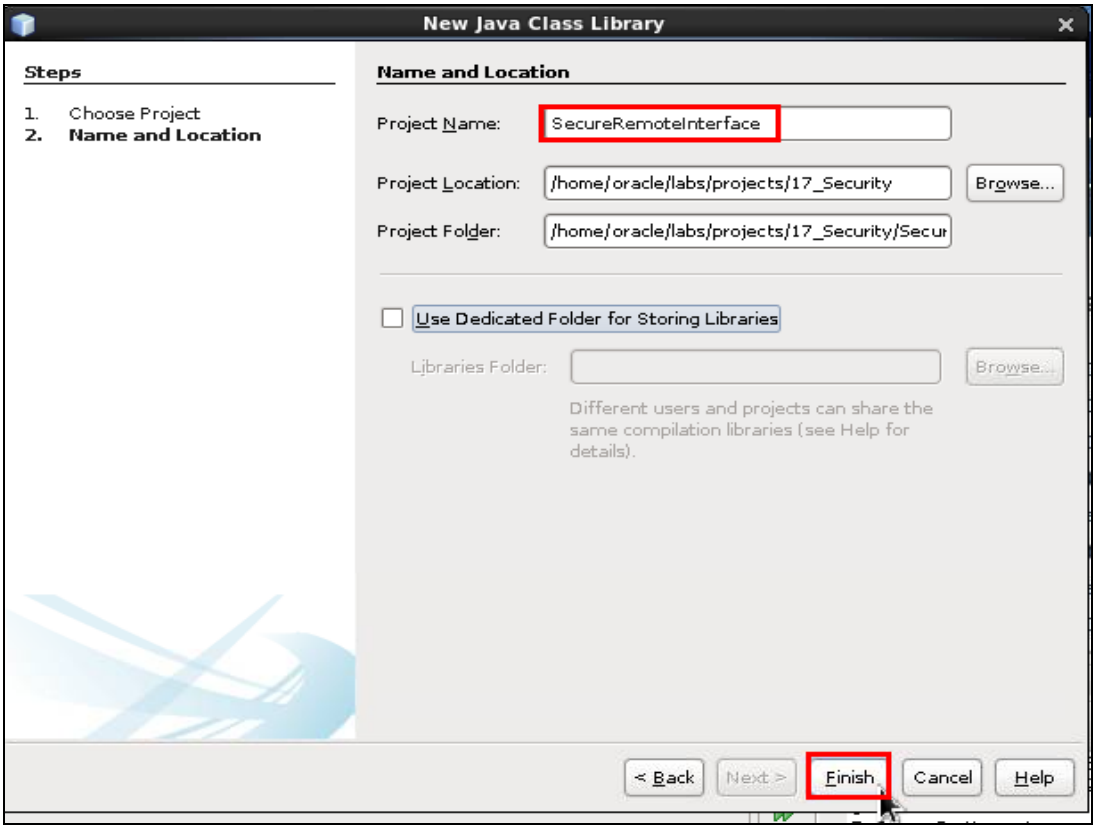
1. Select File > New Project.



2. Select Java from Categories and Java Class Library from Projects. Click **Next**.



3. Enter `SecureRemoteInterface` as Project Name. Click **Finish**.



In the next practice, you create and secure a session bean in an enterprise application. The session bean is accessed via a remote interface.

Practice 17-3: Creating and Securing the Enterprise Application

Overview

In this practice, you create a simple session bean that you access via a remote interface in the class library project.

Assumptions

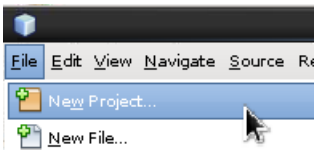
You have finished Practices 17-1 and 17-2.

Tasks

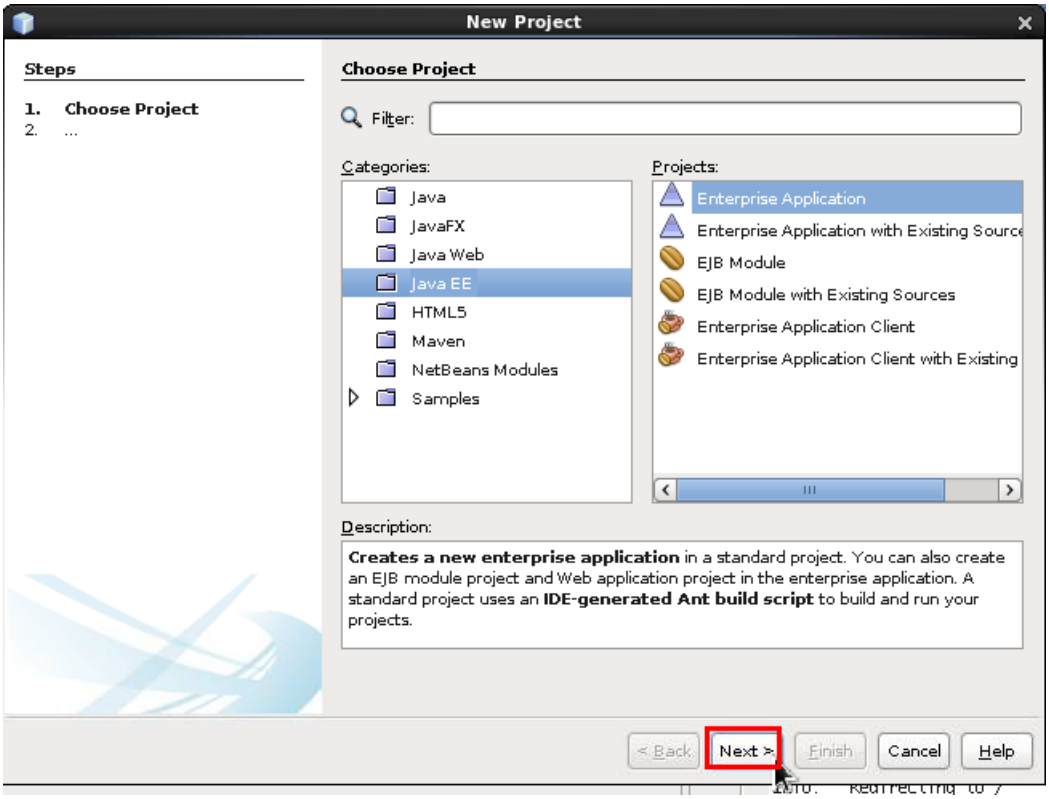
Creating the Enterprise Application Project

In this section, you create an enterprise application that contains an EJB module.

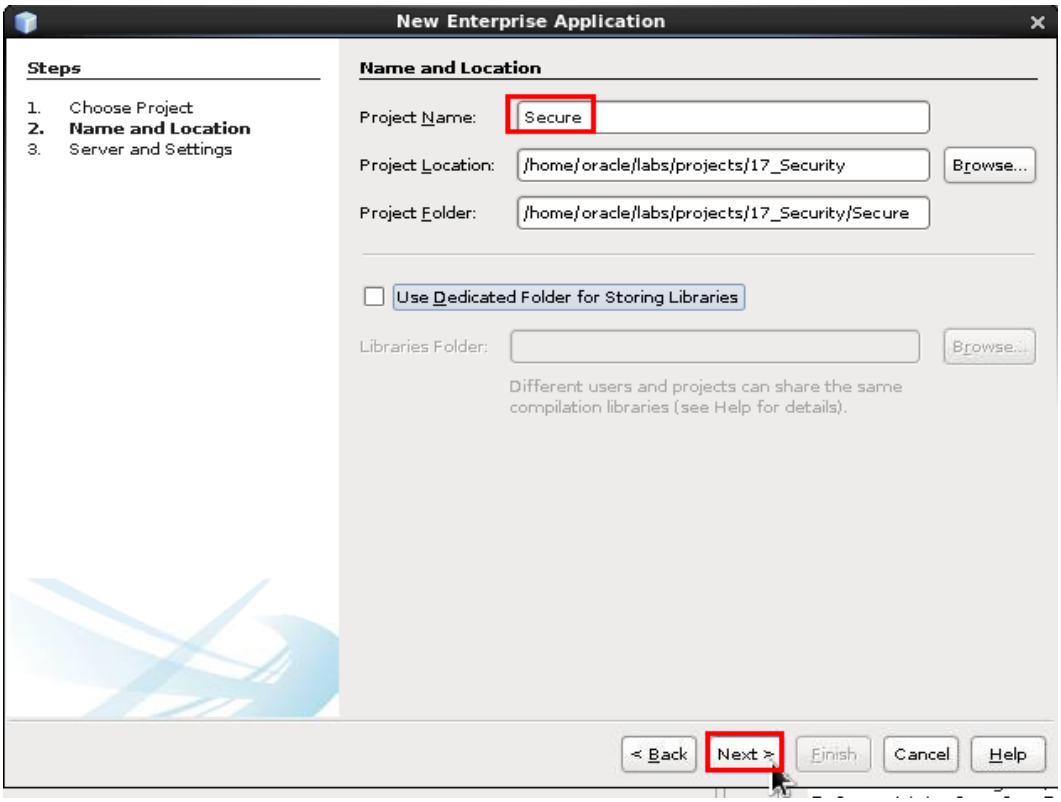
1. Select File > New Project.



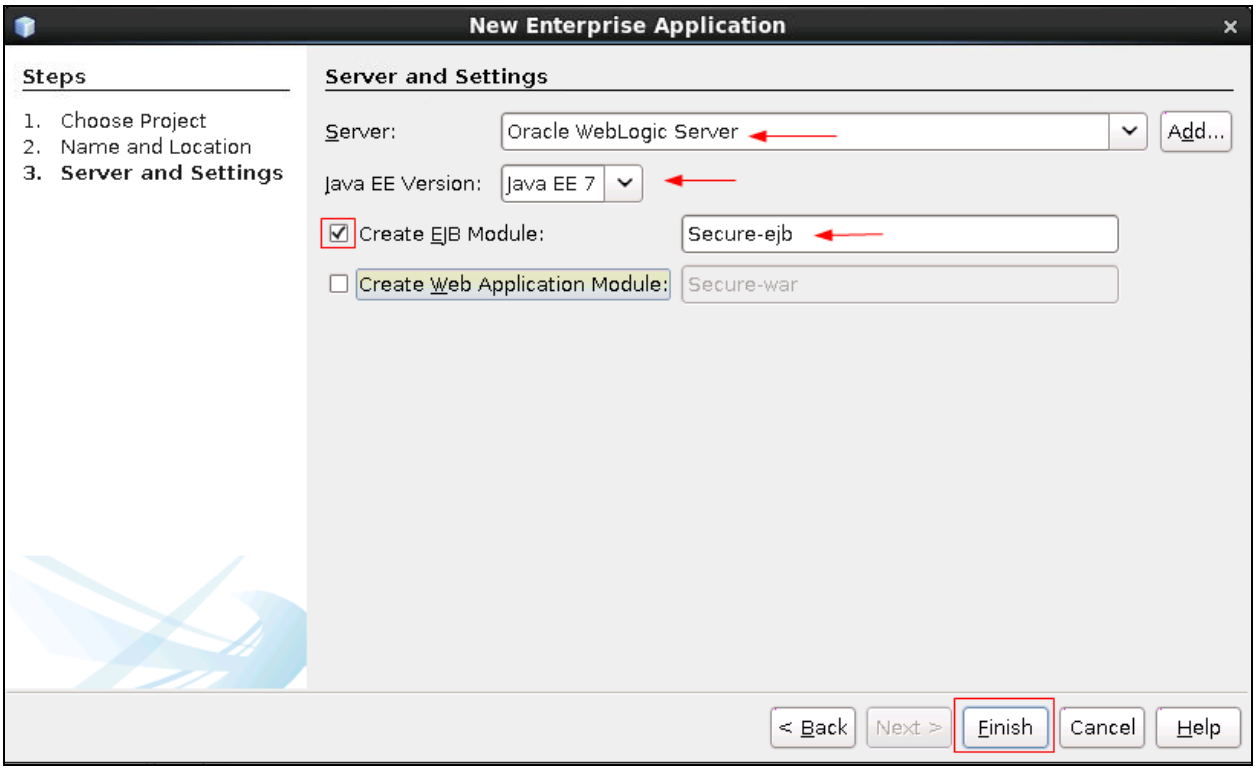
2. Select Java EE from Categories and Enterprise Application from Projects. Click **Next**.



3. Enter `Secure` as Project Name. Click **Next**.



4. Make sure that WebLogic Server is selected as the server and Java EE 7 is the Java EE Version. Select Create EJB Module and deselect Create Web Application Module. Click **Finish**.

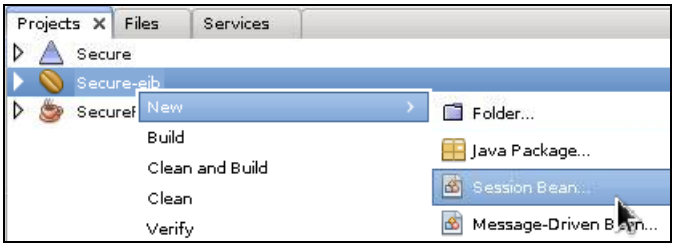


The `Secure` EJB Enterprise Application project is created. Now you create and secure a session bean in the EJB module project.

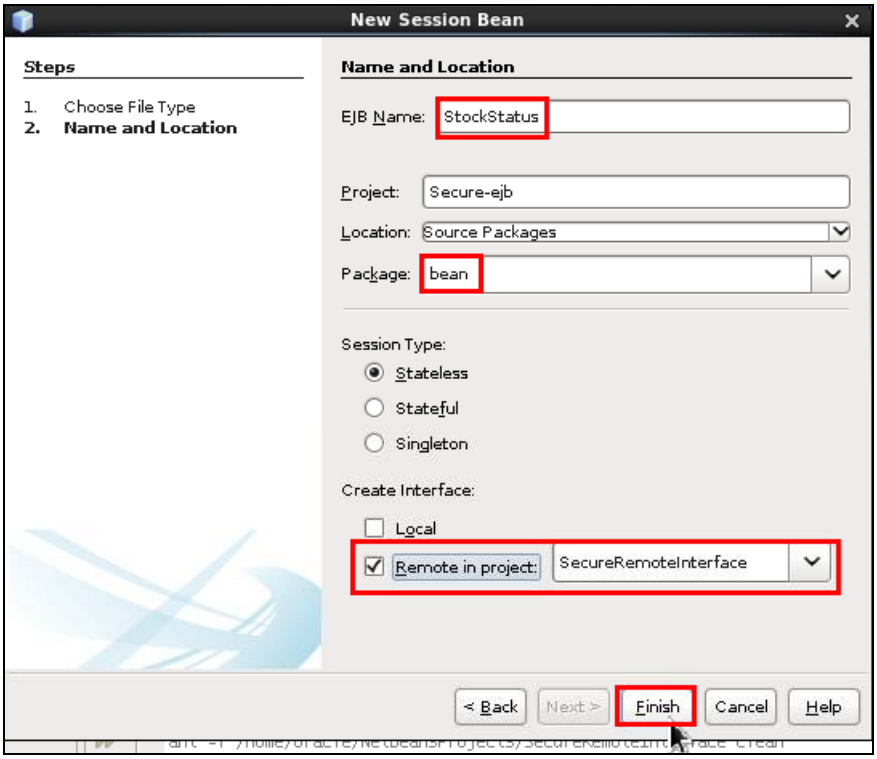
Securing a Method in a Session Bean

In this section of the practice, you create a session bean in the EJB module project and create a `getStatus` method, securing it by annotating it with `@RolesAllowed`.

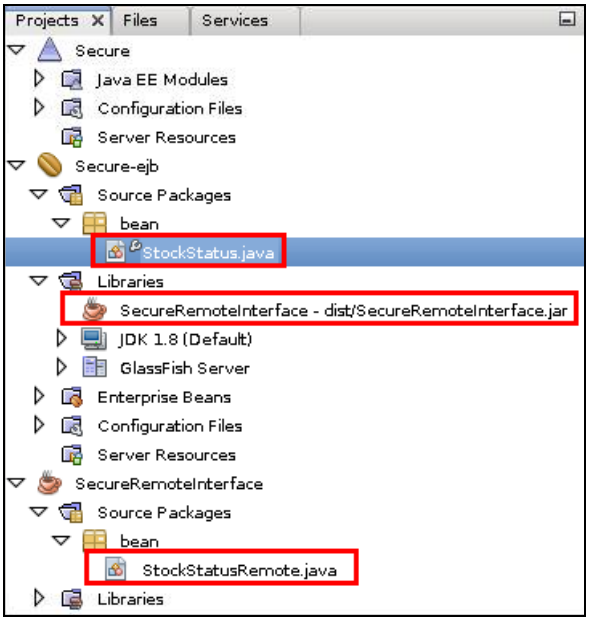
5. Right-click the `Secure-ejb` module in the Projects window. Select New > Session Bean.



6. Perform the following actions in the New Session Bean window:
   - Enter `StockStatus` as EJB Name.
   - Enter `bean` for Package.
   - Select “Remote” and select **`SecureRemoteInterface`** from the drop-down list.
   - Click **Finish**.

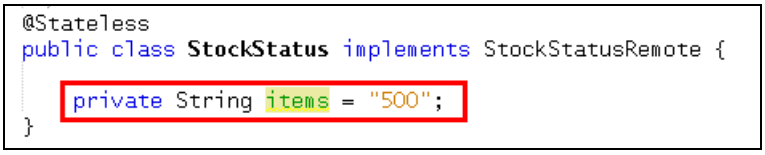


After you click Finish, the IDE creates the `StockStatus` class and opens the file in the source editor. It also creates the `StockStatusRemote` remote interface for the bean in the bean package under the `SecureRemoteInterface` project. In addition to this, it adds the `SecureRemoteInterface` JAR to the classpath of the `Secure-ejb` project. These three files are highlighted in red in the following screenshot:

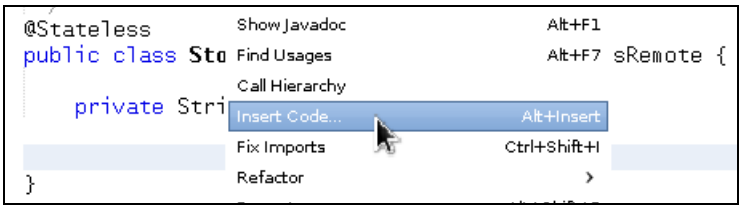


7. In the source editor, add the following line of code to `StockStatus`:

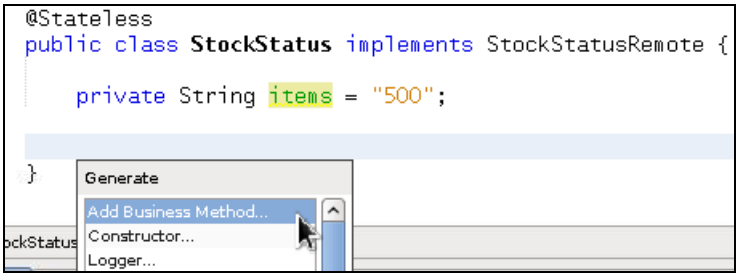
```
private String items = "500";
```



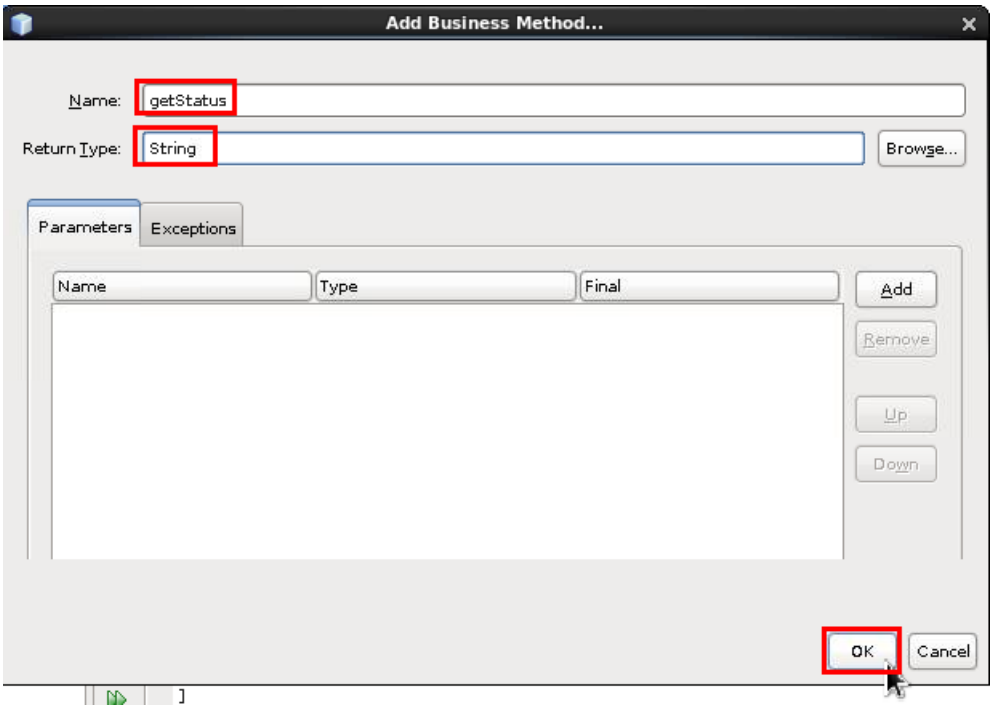
8. In the source editor, right-click in the class and select Insert Code.



9. Select Add Business Method.



10. Enter `getStatus` as Name and `String` as Return Type, and click **OK**.



The IDE automatically exposes the business method in the `StockStatus` class.

```
@Stateless
public class StockStatus implements StockStatusRemote {

    // Add business logic below. (Right-click in editor and choose
    // "Insert Code > Add Business Method")
    private String items = "500";

    @Override
    public String getStatus() {
        return null;
    }
```

11. In the source editor, for the `StockStatus` class, add the following line of code to the `getStatus` method, replacing the current return line:

```
return "The stock contains " + items + " items";
```

```
@Stateless
public class StockStatus implements StockStatusRemote {

    private String items = "500";

    @Override
    public String getStatus() {
        return "The stock contains " + items + " items";
    }


}
```

12. Add the following annotation to the `getStatus` method:

```
@RolesAllowed({"USERS"})
```

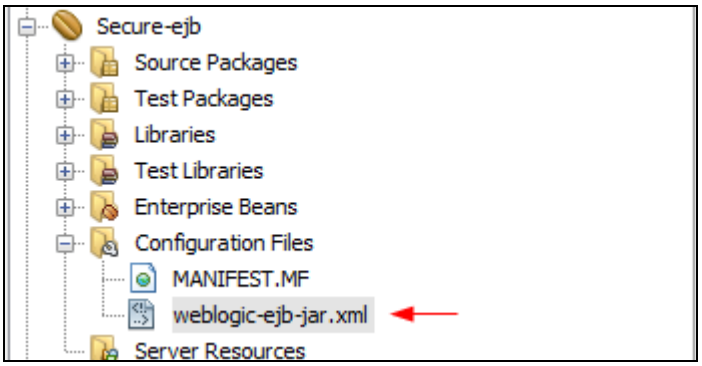
```
@Override
@RolesAllowed({"USERS"})
public String getStatus() {
    return "The stock contains " + items + " items";
}
```

13. Right-click in the editor and select Fix imports (or press Ctrl + Shift + I) to add the line `import javax.annotation.security.RolesAllowed;` at the beginning of the code.
14. Save your work.

Configuring the Deployment Descriptors

In this section of the practice, you map the security roles that are used in the enterprise application (`USERS`) to the users and groups that you configured on the WebLogic application server. In Practice 17-1, you created the group `store_users` on the application server. Now you need to map this group to the security role `USERS` in the enterprise application.

1. Expand the `secure-ejb` project. Then expand `Configuration Files`, select `weblogic-ejb-jar.xml`, which is the WebLogic EJB deployment descriptor, and open it in the editor.



2. Edit the `weblogic-ejb-jar.xml` file:
   a. Delete the IDE generated code.
   b. Add the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-ejb-jar>
<security-role-assignment>
        <role-name>USERS</role-name>
        <principal-name>store_users</principal-name>
 </security-role-assignment>
</weblogic-ejb-jar>
```

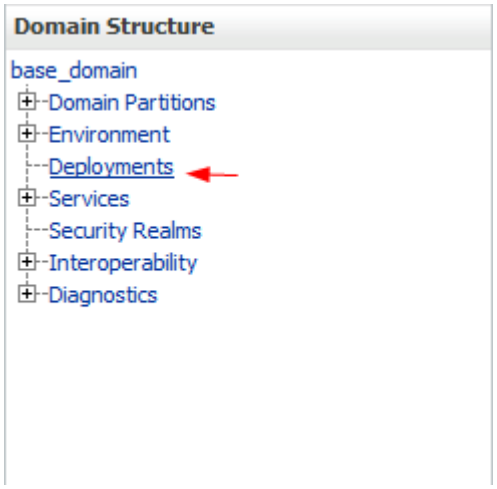
   c. Save the file.

   This maps the EJB roles to users or groups in WebLogic Security Realm (myrealm). The <`role-name`> must match with the roles names used in the `RolesAllowed` annotation, and the <`principal-name`> must match with a user or role in WebLogic Security Realm.

   The `getStatus` method is now secure and only those users in the `store_users` group that you specified on the server can access the method.

3. Verify the permissions for the EJB in WebLogic server. Perform the following steps:
   a. Deploy the `Secure` application in WebLogic Server.
   b. Log in to the WebLogic Server Admin console.

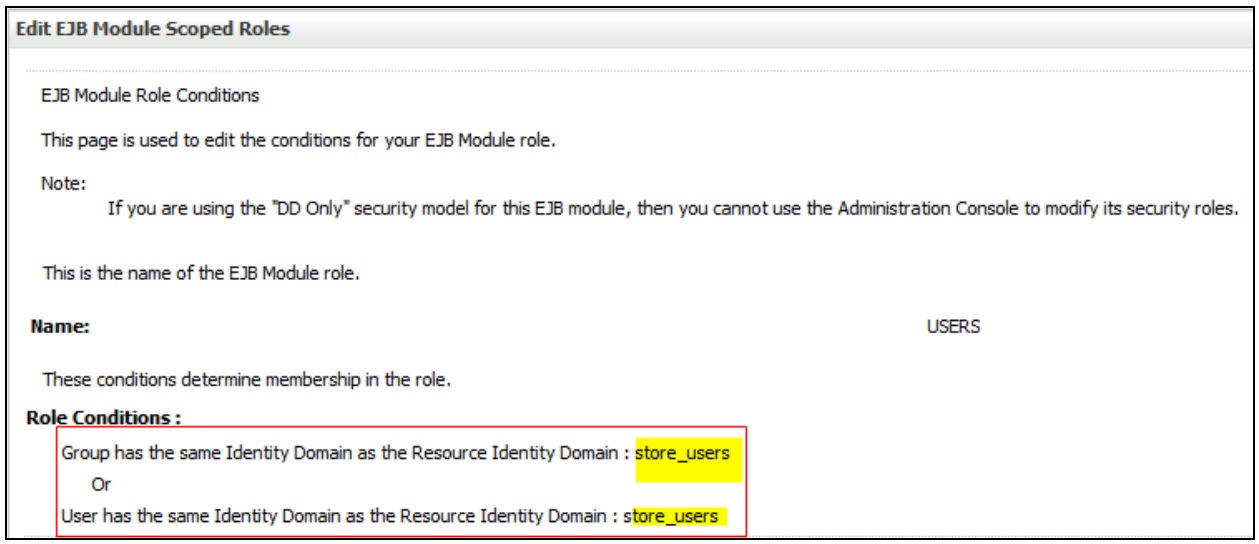
c. On the Admin Console page, click Deployments.



d. In the Deployments table, select the `Secure` application, expand the project, and click `Secure-ejb.jar`.

e. On the Settings page, click the **Security** tab and then click the **Roles** tab. The USERS role is displayed.

f. Click USERS. The groups connected to this EJB role are displayed.



Practice 17-4: Creating and Running the Application Client

Overview

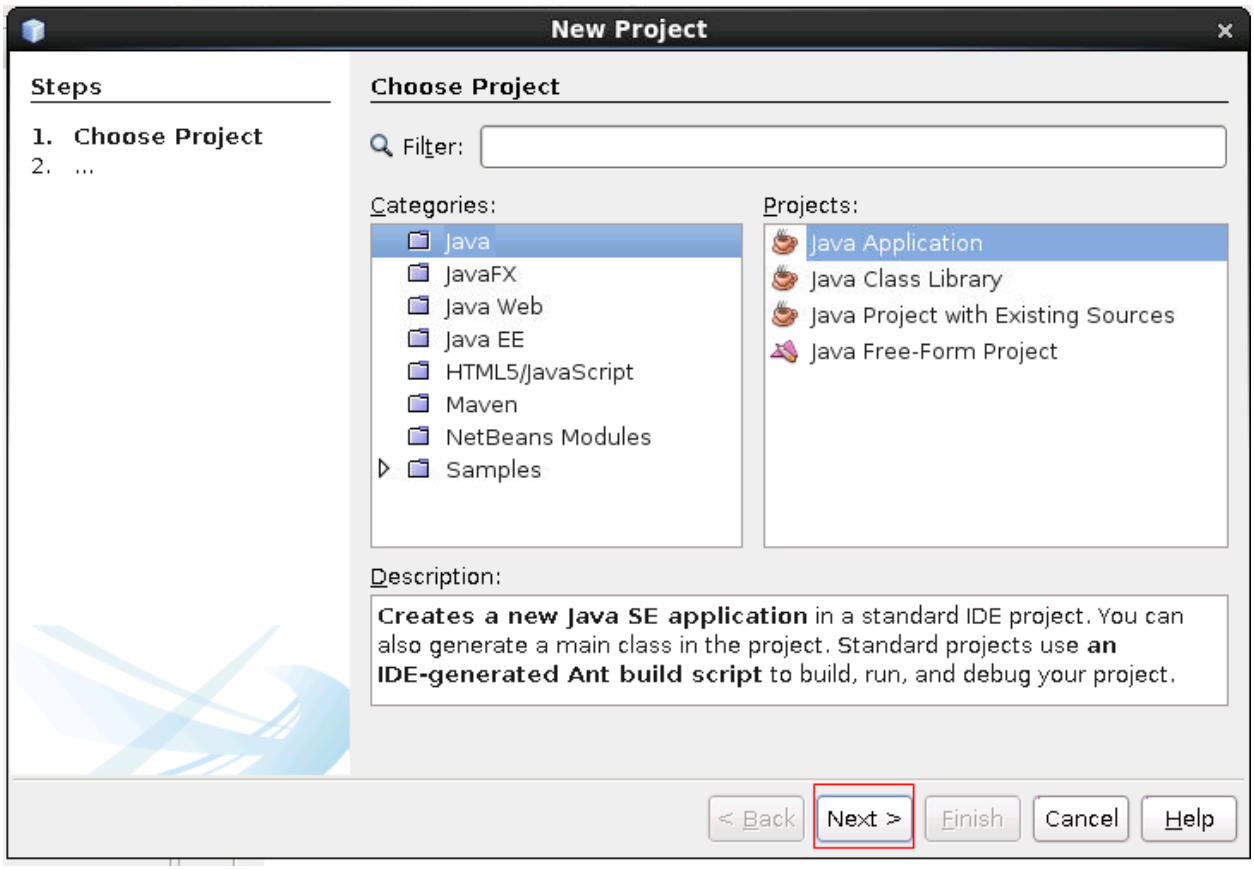
In this practice, you create a simple application client to access the `StockStatus` session bean.

Assumptions

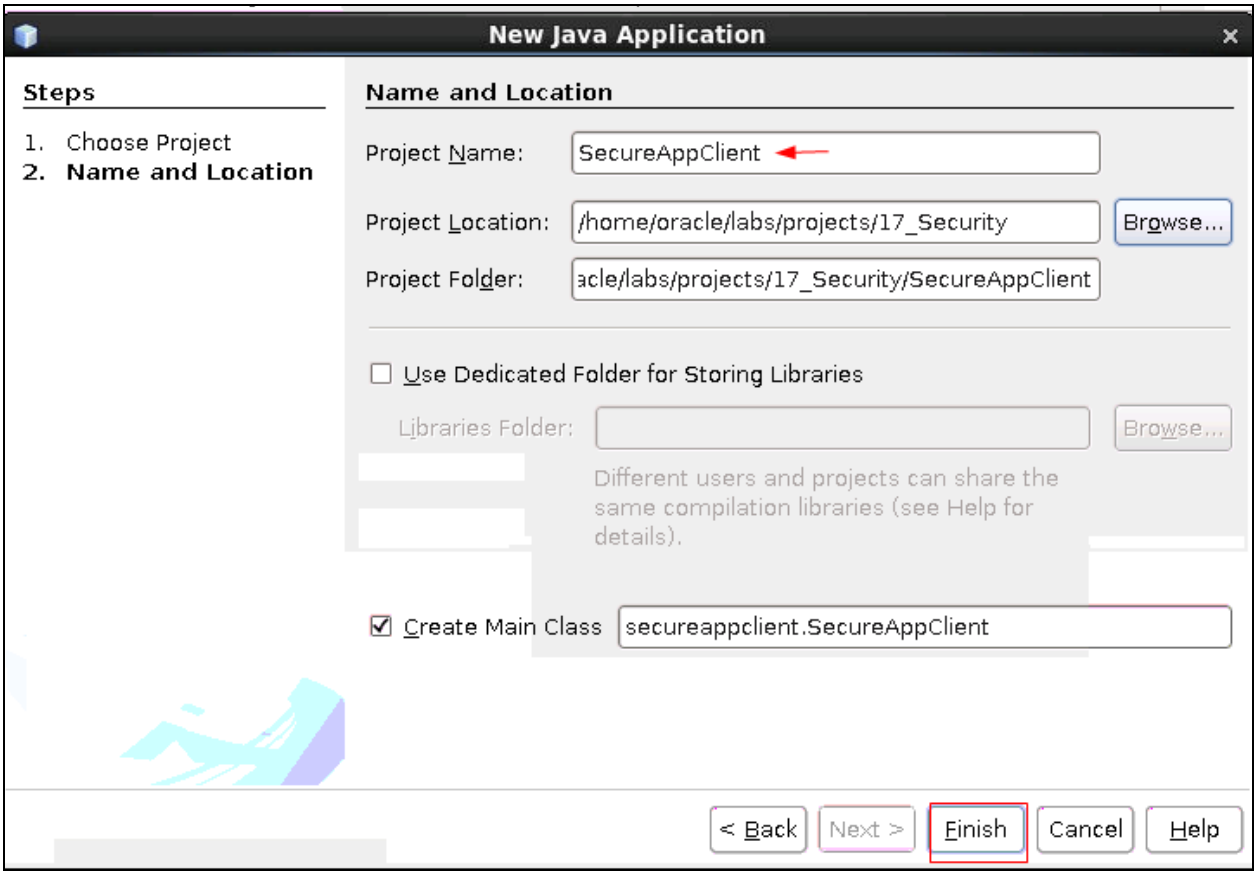
You have finished Practices 17-1, 17-2, and 17-3.

Tasks

1. Select File > New Project, and select Java from Categories and Java Application from Projects. Click **Next**.



2. Enter `SecureAppClient` as Project Name. Click **Finish**.



When you click Finish, the IDE creates the `SecureAppClient.java` class and opens the file in the source editor.

Install client libraries

You will need to configure the `SecureAppClient` project to reference the client libraries of the `Secure` project. Additionally, you will need to configure the `EnterpriseCalculatorClient` project to reference the WebLogic runtime libraries. This will enable you to use the WebLogic JNDI provider implementation.

1. Add the SecureRemoteInterface library to `SecureAppClient`.
   a. Right-click SecureAppClient and select Properties > Libraries.
   b. Click Add Project.
   c. In the Add Project dialog box, browse to `/home/oracle/labs/projects/17_Security`.
   d. Select the SecureRemoteInterface project.
   e. Click Add Project Jar Files.
   f. Click OK.

2. Add the WebLogic runtime libraries to `SecureAppClient`.
   a. Right-click SecureAppClient and select Properties > Libraries.
   b. Click Add Jar/Folder.
   c. In the file chooser, browse to `/home/oracle/weblogic/wls12210/wlserver/server/lib`.
   d. Select `weblogic.jar`.
   e. Click OK.
3. To perform the JNDI global lookup of the `StockStatus` bean and invoke the `getStatus` method, complete the following steps:
   a. Edit SecureAppClient.java.
   - Include the following import statements:

```
import  bean.StockStatusRemote;
import java.util.Hashtable;
import javax.naming.*;
```

   - Modify the `main` method to invoke the EJB.

```
String jndiPath = "java:global/Secure/Secure-ejb/StockStatus";
try {
    final Context cxt = getInitialContext();
    System.out.println("standaloneapp.main: looking up bean at:
" + jndiPath);
   StockStatusRemote StockStatus = (StockStatusRemote)
cxt.lookup(jndiPath);
  System.out.println(StockStatus.getStatus());
   }catch (Exception ex) {
      ex.printStackTrace();
  }
```

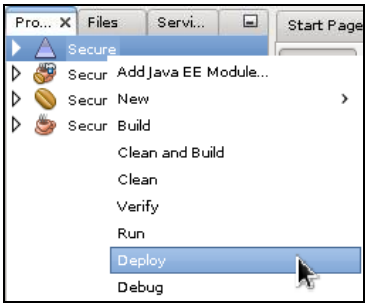
   - Add the `getInitialContext` method to obtain a reference to JNDI.

```
private static InitialContext getInitialContext() throws
NamingException {
        Hashtable env = new Hashtable();
       env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
        env.put(Context.PROVIDER_URL, "t3://localhost:7001");
        env.put(Context.SECURITY_PRINCIPAL, "admin");
        env.put(Context.SECURITY_CREDENTIALS, "welcome1");
        return new InitialContext(env);
```

Running the Application

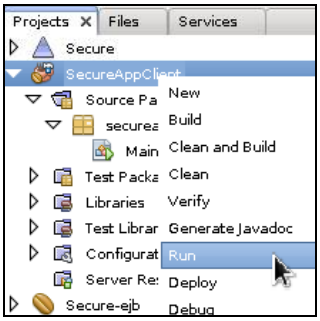
In this section, you deploy the enterprise application to the server and run the application client to test that the method in the enterprise application is secure and that the user roles are mapped correctly.

4. Right-click the `Secure` enterprise application project and select **Deploy**.

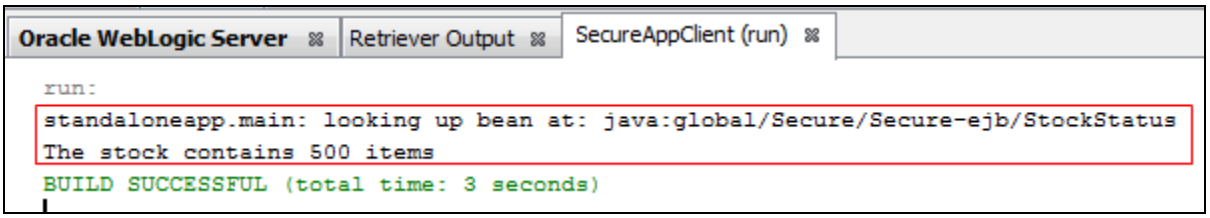


The IDE builds the enterprise application and deploys the EAR archive to the server.

5. Right-click the `SecureAppClient` project in the Projects window and select **Run**.



The following appears in the Output window in NetBeans:



6. You can specify any unauthorized role and test the application. For example, you can specify the principal role as `admin1`.

```
private static InitialContext getInitialContext() throws NamingException {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
    env.put(Context.PROVIDER_URL, "t3://localhost:7001");
    env.put(Context.SECURITY_PRINCIPAL, "admin1");
    env.put(Context.SECURITY_CREDENTIALS, "welcome1");

    return new InitialContext(env);
}
```

7. If you execute the `SecureAppClient` application, it generates the following exception:

```
javax.naming.AuthenticationException: User failed to be
authenticated.
```