ORACLE University

Integrated Cloud Applications & Platform Services

Developing Applications for the Java EE 7 Platform

Activity Guide

D98815GC10
Edition 1.0 | January 2018 | D100414

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

ORACLE

Author

Vasily Strelnikov

Technical Contributors and Reviewers

Phil Franklin
Girija C
Daniel Milne
Dorin Paraschiv
Jacobo Marcos
Nikolai Elistratov

Editors

Raj Kumar
Vijayalakshmi Narasimhan

Graphic Designer

Kavya Bellur

Publishers

Syed Ali
Pavithran Adka
Raghunath M
Asief Baig
Srividya Rameshkumar
Sumesh Koshy

Table of Contents

Practices for Lesson 1: Course Introduction

Practices for Lesson 1: Overview

Overview

The practices in this course build upon each other. If you don't complete a practice, you can load a solution project into NetBeans to "catch up" so that you can proceed to the next practice.

In addition, all practices depend on the successful completion of "Practices for Lesson 2," during which you create NetBeans, WebLogic, and Derby Database configurations that all other practices rely upon. If you are unable to complete "Practices for Lesson 2," you would not be able to load any solution projects to proceed to the following practices.

In NetBeans, do the following to load a solution project:

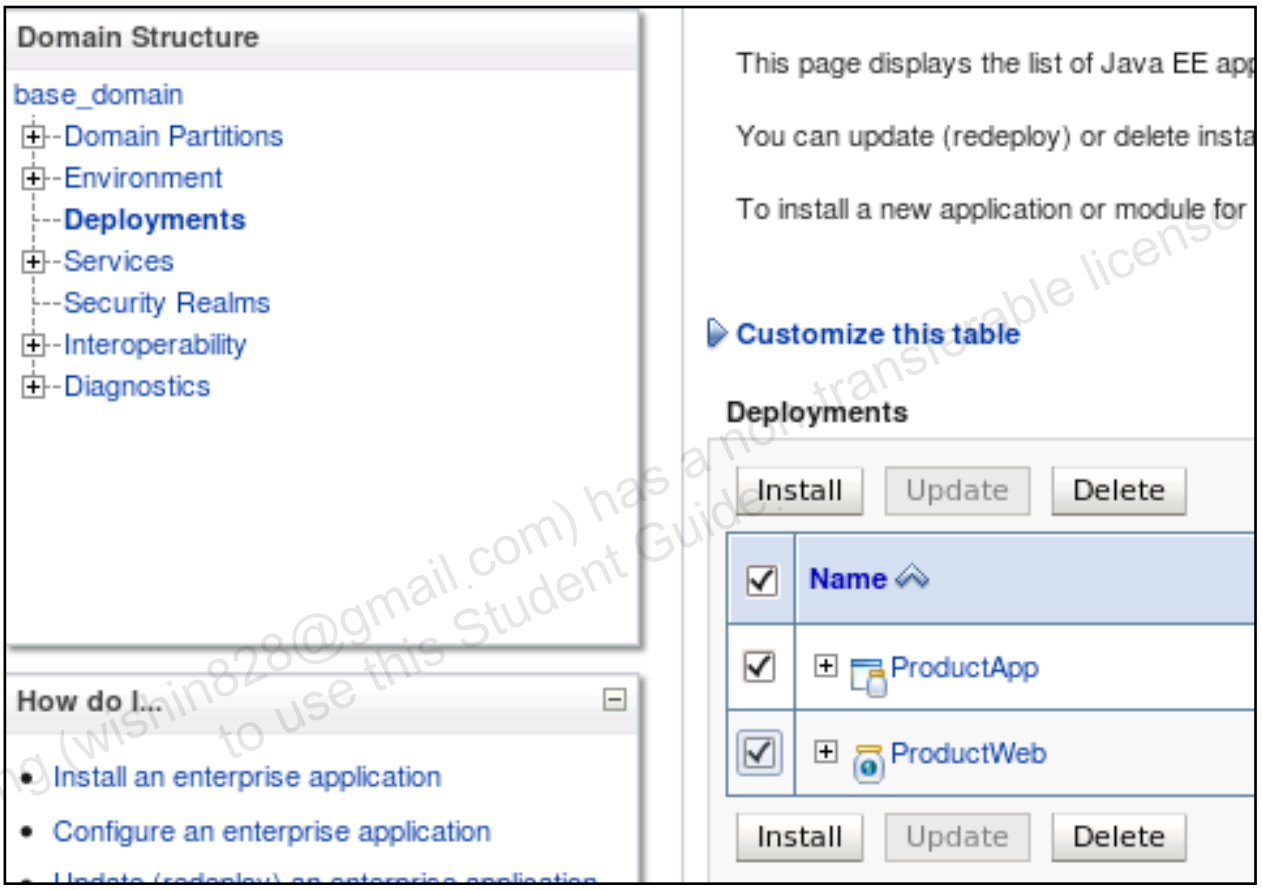
1. If you want to open the solution *for the practice you are currently working on*, do the following:
   a. Close all currently opened projects in NetBeans: Select **File > Close All Projects**.
   b. Select **File > Open Project**.
   c. Navigate to the /home/oracle/labs/solutions folder and to the appropriate practice subfolder. For example, to open the solution for "Practices for Lesson 7," open all projects from the **practice7** subfolder.
   d. Select **all projects** from appropriate subfolder.
   e. Make sure you select the **Open Required Projects** check box.
   f. Click **Open Project**.
2. If you want to *start the next practice form a fresh solution project*, do the following:
   a. Close all currently opened projects in NetBeans: Select **File > Close All Projects**.
   b. Select **File > Open Project**.
   c. Navigate to the /home/oracle/labs/solutions folder and to the appropriate practice subfolder. For example, to start "Practices for Lesson 11" from a fresh project set, open solution projects from the practice10 subfolder.

   Note: This rule has two exceptions: To start the practices for lessons 4 and 7 from fresh projects, you need to open projects from the practice4_start and practice7_start folders.
   d. Select **all projects** from appropriate subfolder.
   e. Make sure you select the **Open Required Projects** check box.
   f. Click **Open Project**.

Naming Conflict: When you deploy a project that is taken from the solutions folder, you may get an error that indicates that an application with this name already exists on your server. This is what this error may look like:

```
Deployment failed. The message was: [J2EE:160257] Application with
name "ProductApp" cannot be deployed because the name is already in
use in Domain named "base_domain"
```

3. To resolve this naming conflict when deploying a solution project, do the following:
   a. Open a browser and navigate to **WebLogic Console**. It is available on this URL: `http://localhost:7001/console`
   b. Log in as user `weblogic` with password `welcome1`
   c. Click the **Deployments** link in the **Domain Structure** panel.
   d. Select **all applications**.



   e. Click the **Delete** button.
   f. Retry the deployment of your projects from NetBeans.

Note: If you did not successfully complete “Practices for Lesson 2,” or Practice 5-1, you would not be able to deploy solutions for subsequent practices, because these are server configuration practices.

Practices for Lesson 2: Introduction to Java EE

Practices for Lesson 2: Overview

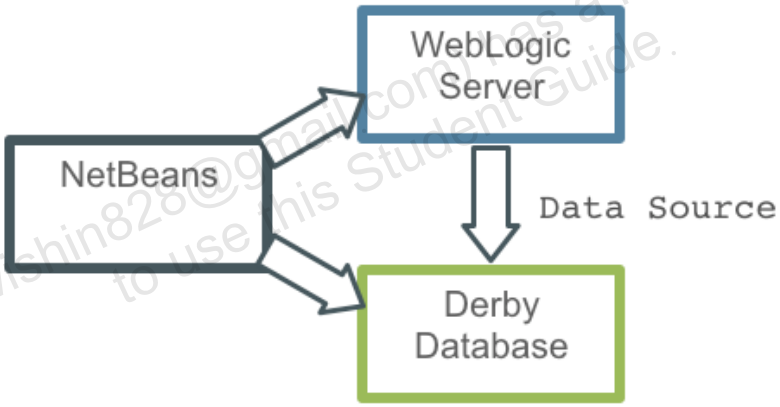
Overview

In these practices, you configure your environment to develop Java EE 7 applications. In Practice 2-1, you configure and start a WebLogic Server instance. In Practice 2-2, you configure a Derby Database and populate it with data. In Practice 2-3, you configure a JDBC data source.

In this practice, you configure the following:

- NetBeans connection to WebLogic Server
- NetBeans connection to Derby Database

  WebLogic data source with connection pool to Derby Database



Practice 2-1: Configuring and Starting WebLogic Server

Overview

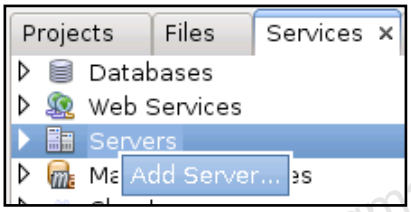
In this practice, you configure WebLogic Server as a service in NetBeans and start this server.

Assumptions

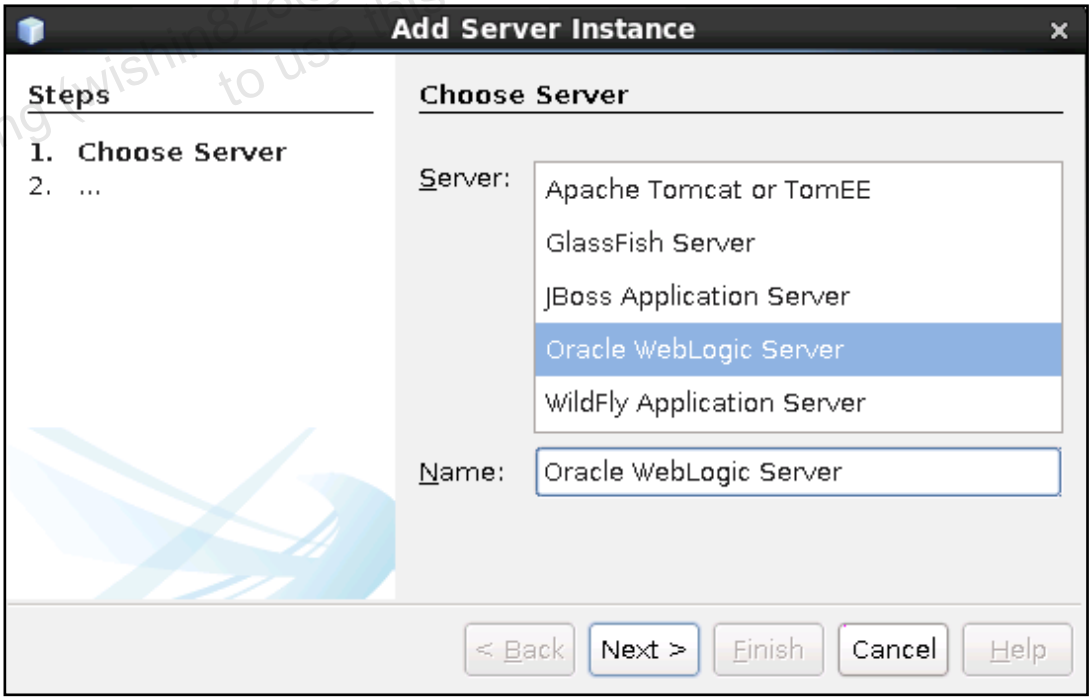
- JDK 8 is installed.
- NetBeans 8.1 EE is installed.
- WebLogic Server is installed.

Tasks

1. Start NetBeans.
2. Configure NetBeans to start WebLogic Server:
   a. Select **Window > Services** to open the Services panel in NetBeans.
   b. Right-click **Servers** and select **Add Server…**.



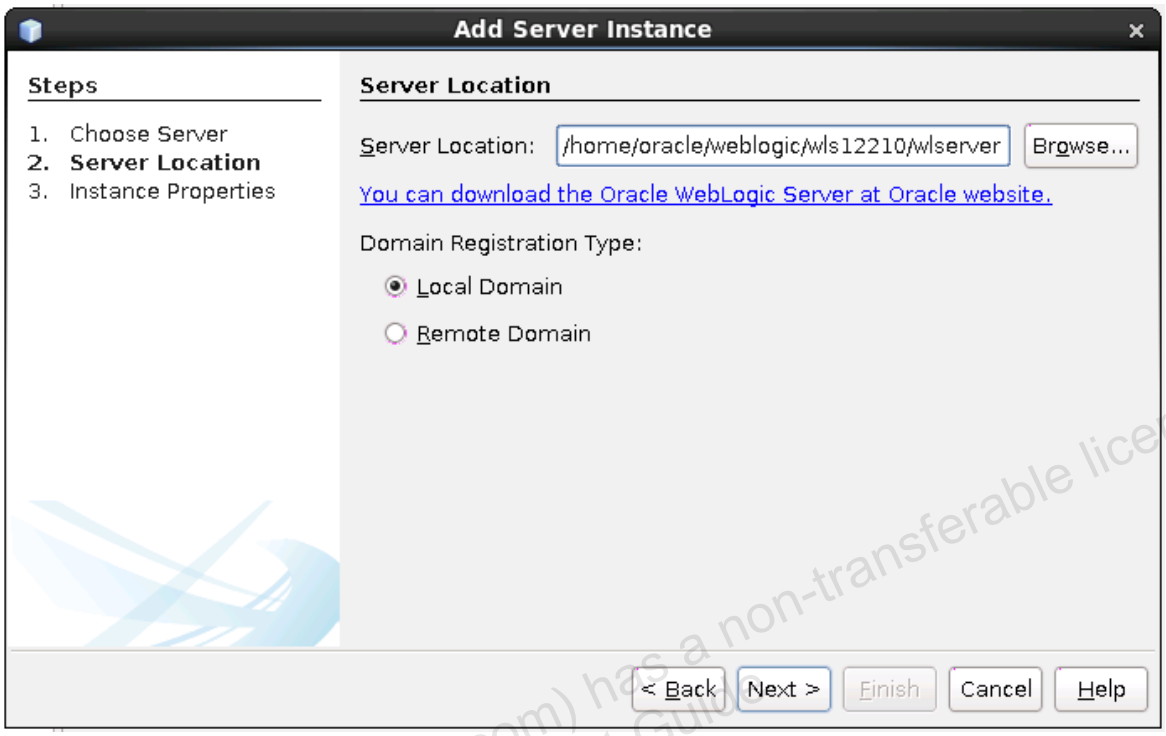
   c. Select **Oracle WebLogic Server**.



   d. Click **Next**.

e. Enter the Server Location as: `/home/oracle/weblogic/wls12210/wlsserver`

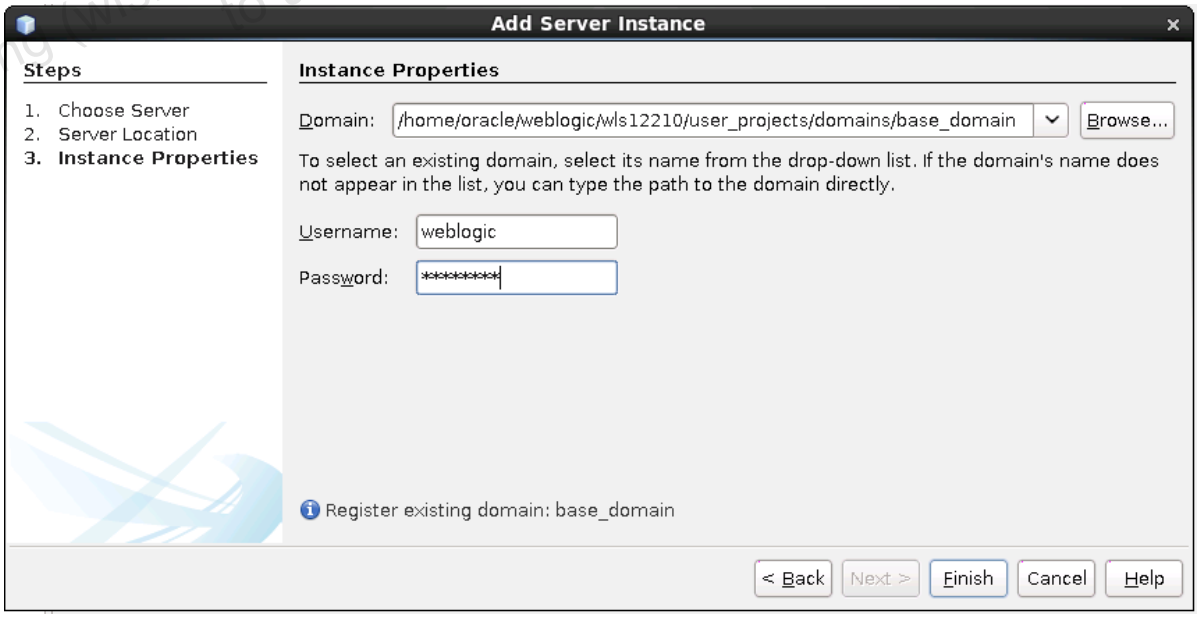
f. Ensure that **Local Domain** is selected as Domain Registration Type.



g. Click **Next**.

h. Enter the domain path: `/home/oracle/weblogic/wls12210/user_projects/domains/base_domain`

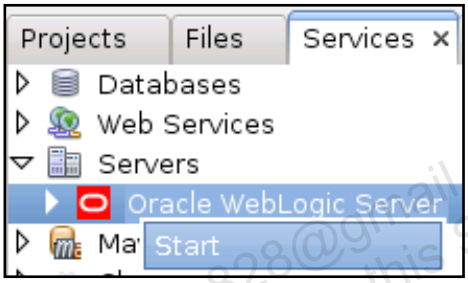
i. Enter `weblogic` as the username and `welcome1` as the password.



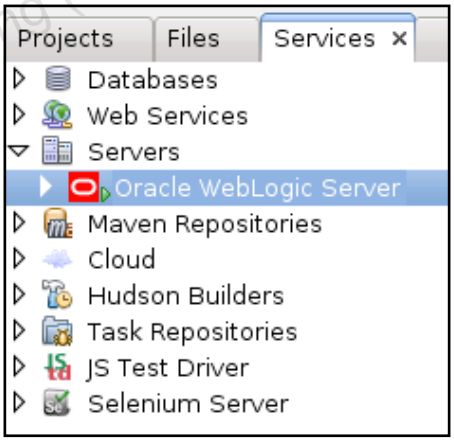
j. Click **Finish**.

Note: If you are prompted for a default keyring password, enter the password `oracle` and click **OK**.

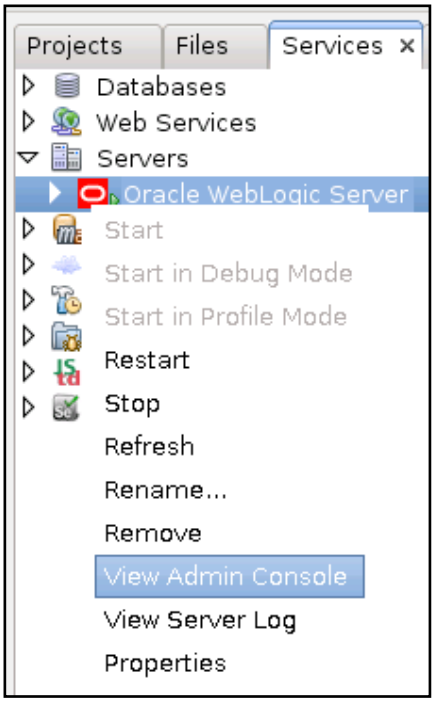
3. Start WebLogic Server:
   a. Expand the **Servers** folder.
   b. Right-click **Oracle WebLogic Server** and select **Start**.



Note: After the server completes the startup process, you should see a small green triangle beside the server name, indicating that the server started successfully.



4. Open WebLogic Server Console:
   a. Right-click **Oracle WebLogic Server** and select **View Admin Console**.



   The console opens in a new browser window.

   Note: Alternatively, open a browser and navigate to:
   `http://localhost:7001/console`

   b. Enter the username `weblogic` and password `welcome1`.

   You should be able to see WebLogic server console, which means the server domain has been configured successfully.

Practice 2-2: Configuring a Java Database

Overview

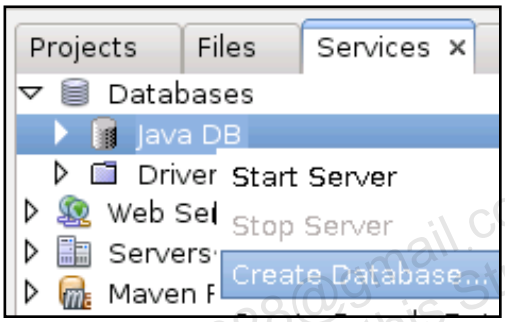
In this practice, you create a database for the practices' application and populate it with sample data.

Assumptions

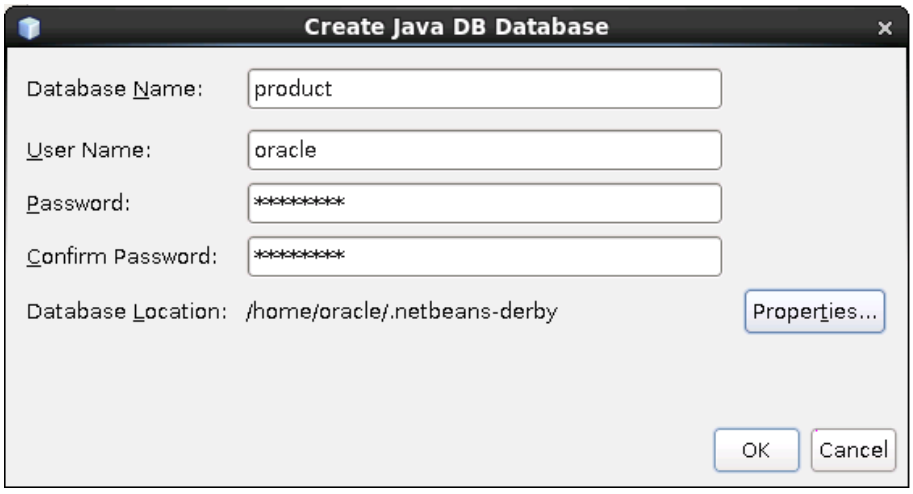
You have successfully competed Practice 2-1.

Tasks

1. Create the `product` database:
   a. Open the **Services** panel.
   b. Expand the **Databases** section.
   c. Right-click **Java DB** and select **Create Database**.



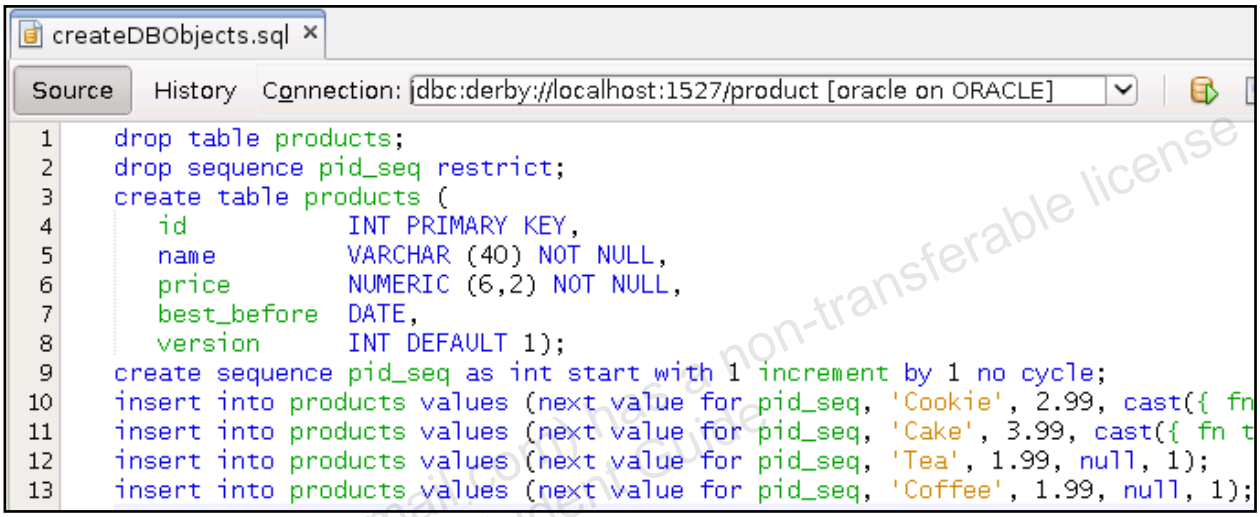
   d. In the Create Java DB Database dialog box, set the following properties:
      - Database Name: `product`
      - User Name: `oracle`
      - Password: `welcome1`
      - Confirm Password: `welcome1`



   e. Click **OK**.

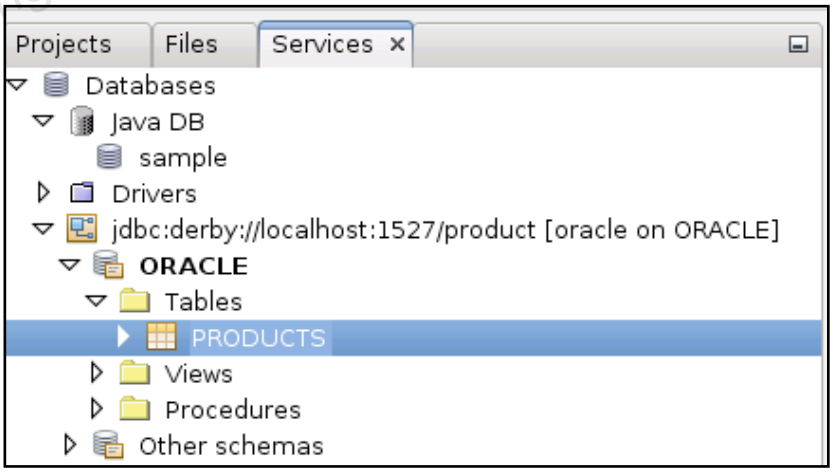
Note: You have just created an empty database called `product`. In addition, NetBeans has added a connection for this database to the Services tab.

2. Create database objects and populate the `products` table:
   a. Double-click the connection to connect to the database server.
   b. Select **File > Open File**, and then open the `createDBObjects.sql` file located in the /home/oracle/labs/resources folder.
   c. Select the **jdbc derby** database connection from the drop-down list.

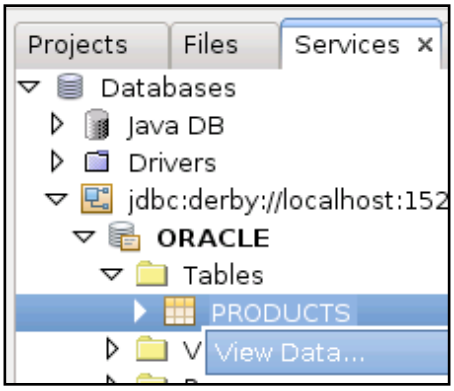


   d. Click the **Run SQL** button.

3. Verify that the database has been successfully populated:
   a. Expand the **ORACLE** schema under the derby database connection.

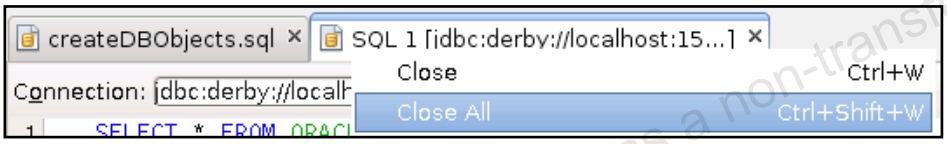


b. Right-click the **PRODUCTS** table and select **View Data**.



Note: You should be able to see four records added to this table.

4. Close both open SQL tabs for the View Data and the `createDBObject.sql` file:
   a. Right-click any of the opened SQL tabs and select **Close All**.



Practice 2-3: Configuring a JDBC Data Source

Overview

In this practice, you configure a JDBC data source on a WebLogic server, to be used by the JPA components in the EJB Module project.

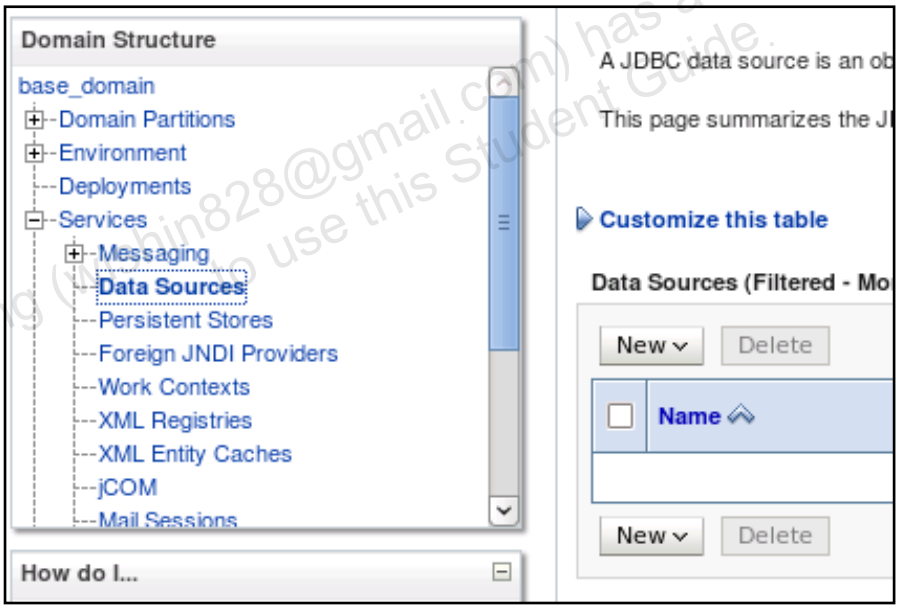
Assumptions

You have successfully completed all previous practices.

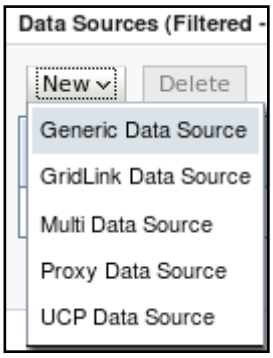
Tasks

1. Log in to WebLogic Server Console:
   a. Open a browser and navigate to: `http://localhost:7001/console`
   b. Log in using `weblogic` as the username and `welcome1` as the password.

2. Create a JDBC data source:
   a. Expand the **Services** section in the Domain Structure panel, and click the **Data Sources** node.

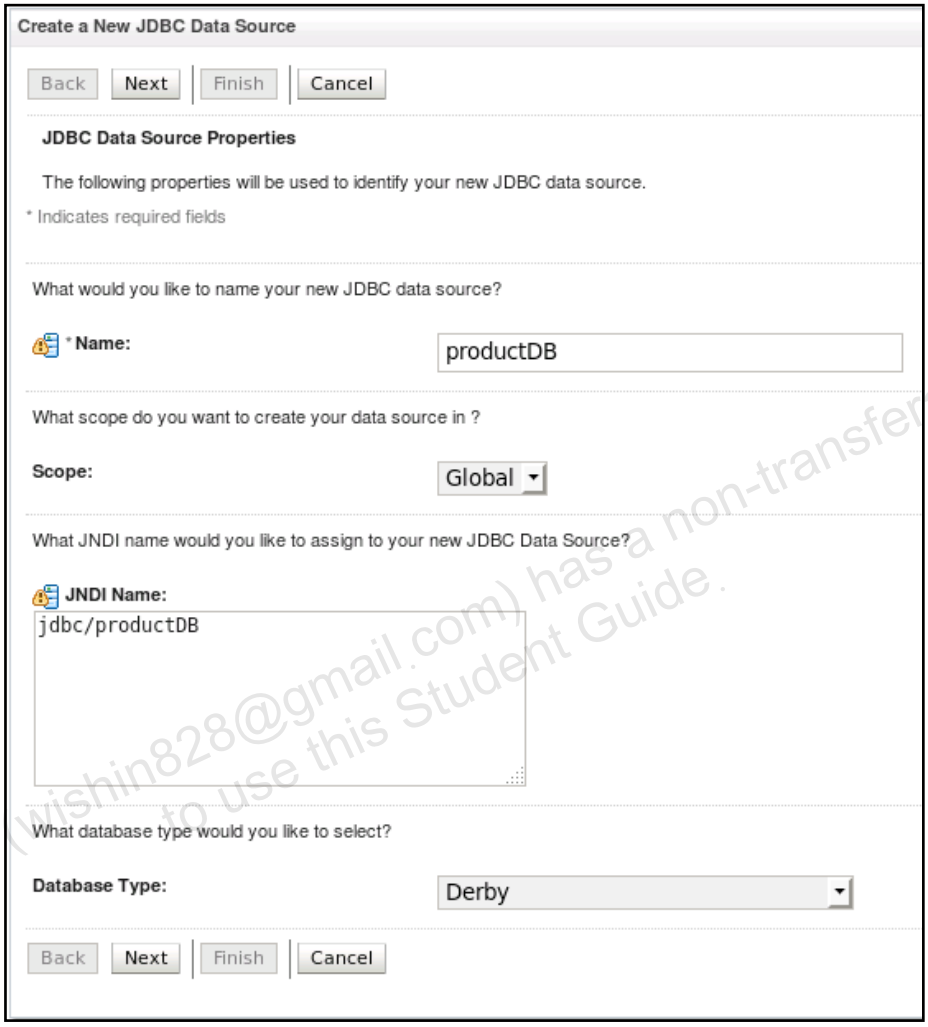


   b. Click the **New** button and select **Generic Data Source**.



c. Set Data Source properties:

- Name: `productDB`
- JNDI Name: `jdbc/productDB`
- Database Type: `Derby`



d. Click **Next**.

e. Confirm that the selected driver type is:
`Derby's Driver (Type 4 XA)  Versions:Any`

f. Click **Next**.

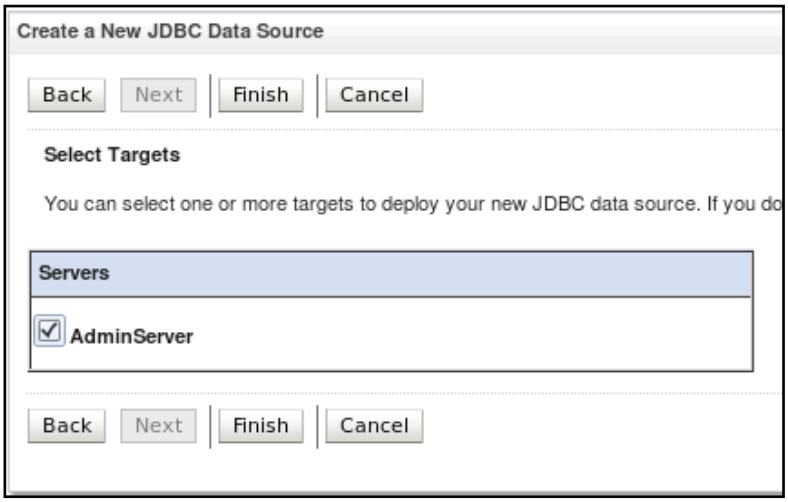
g. There are no settings on the Transaction Options page to be set; just click **Next**.

h. Set Connection Properties:
   - Database Name: `product`
   - Host Name: `localhost`
   - Port: `1527`
   - Database User Name: `oracle`
   - Password and Confirm Password fields: `welcome1`

i. Click **Next**.

j. On the Test Database Connection page, just click **Next** button.

k. On the Select Targets page, select the **AdminServer** check box.



l. Click **Finish**.

Note: You will start using this data source configuration from “Practices for Lesson 4” onwards.

Practices for Lesson 3: Managing Persistence by Using JPA Entities

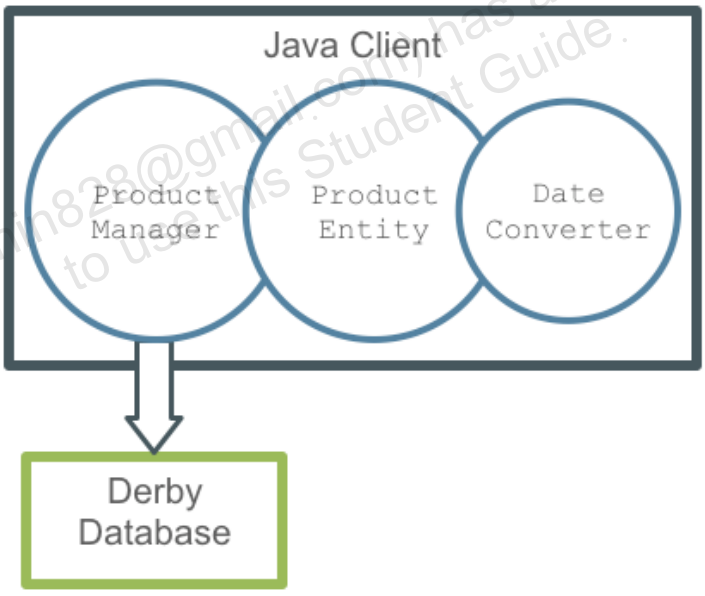
Practices for Lesson 3: Overview

Overview

In these practices, you create a JPA entity, `Product`, mapped to the `PRODUCTS` database table. Then you define a JPA controller to handle entity persistence. Finally, you test your JPA application logic, using the Java SE command line application client.

In this practice, you create a Java application running on the client tier. The application performs operations against the Derby database using:

- The `ProductManager` class, which utilizes `EntityManager` and controls persistence
- The `Product` class, which is mapped with JPA annotations to the `PRODUCTS` database table
- The `Product` entity, which is validated with the help of Bean Validation annotations
- Custom `DateConverter`, which handles data type conversions



Practice 3-1: Creating a JPA Entity

Overview

In this practice, you create a new NetBeans project to contain JPA entities mapped to database objects. This project will run as a Java SE client application.

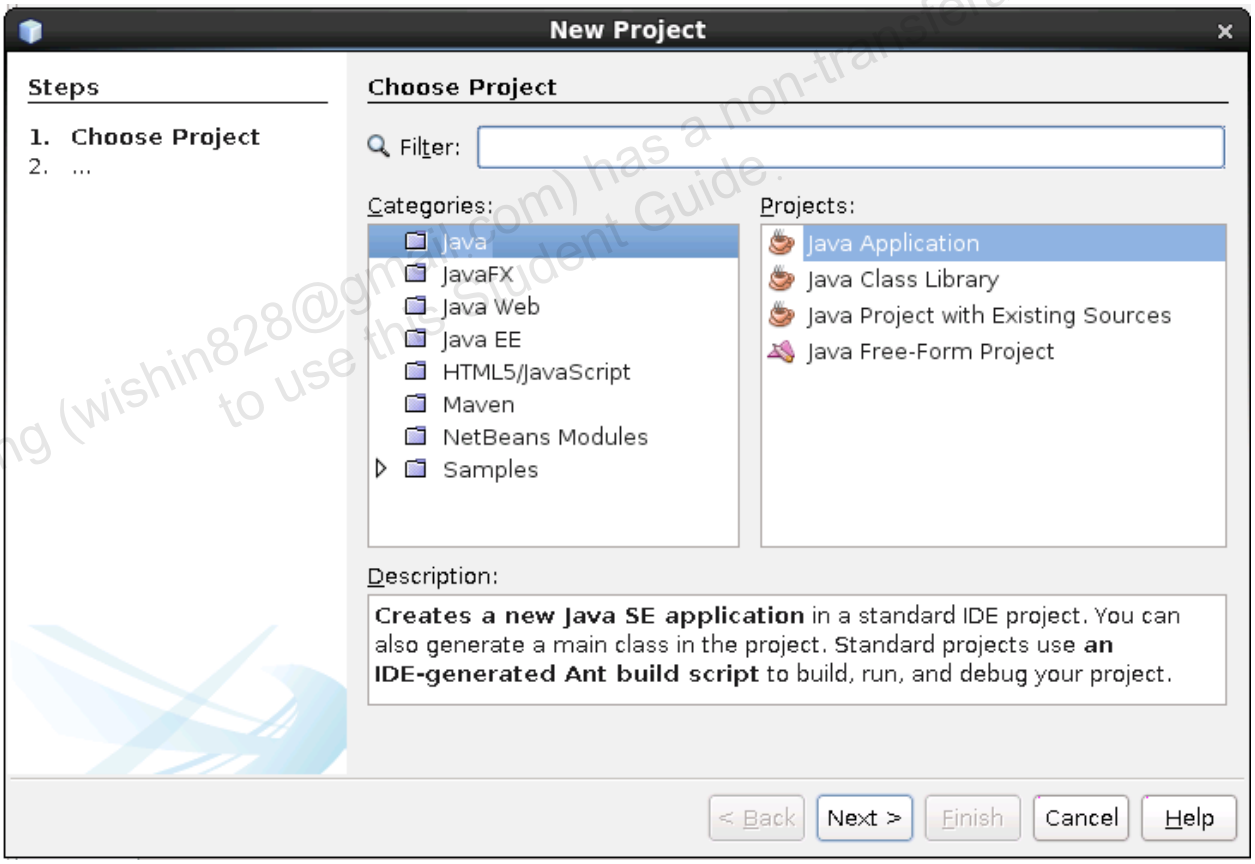
Note: In the next practice (“Practices for Lesson 4”) you will use the same JPA entities in a Java EE EJB application.

Assumptions

You have successfully completed all practices for Lesson 2.

Tasks

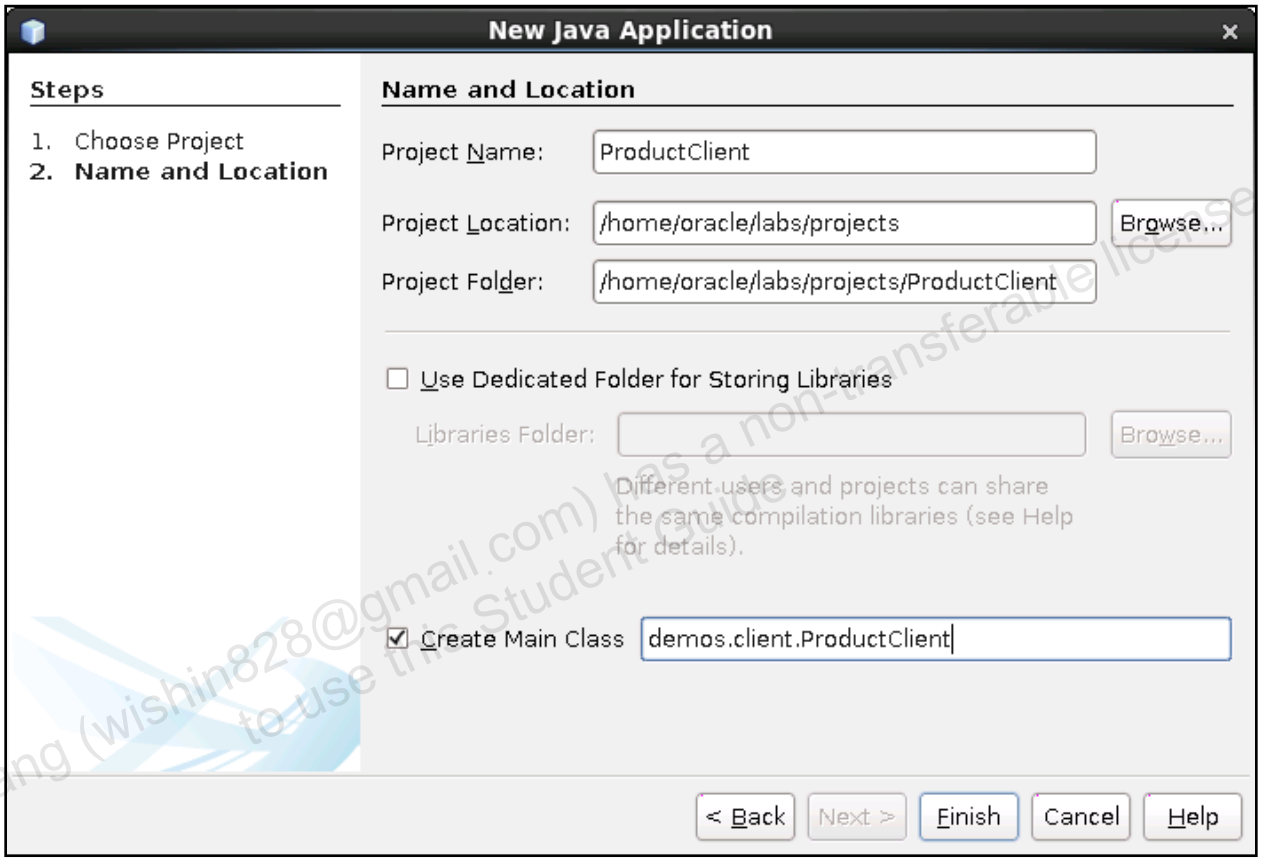
1. Create a new Java Application Project:
   a. In NetBeans, select **File > New Project**.



   b. Select **Java** from the Categories list and **Java Application** in the Projects list.
   c. Click **Next**.

d. In the New Java Application dialog, set following properties:

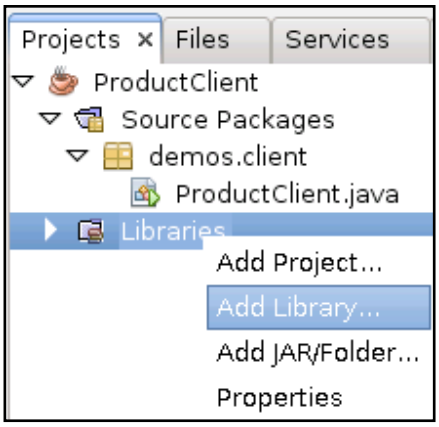
- Project Name: `ProductClient`
- Project Location: `/home/oracle/labs/projects/`
- Project Folder: `/home/oracle/labs/projects/ProductClient`
- Select the **Create Main Class** check box and set the Class Name to: `demos.client.ProductClient`



e. Click **Finish**.

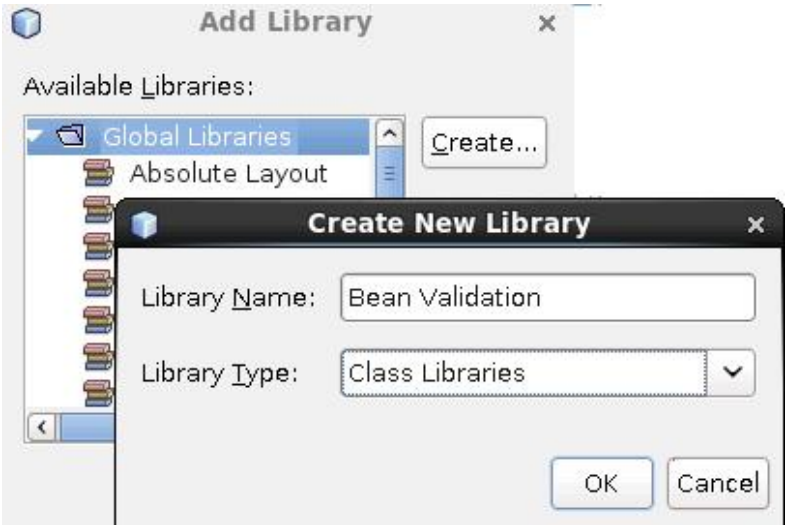
2. Add libraries to the project:

a. Right-click the **Libraries** node in the **Projects** panel, and select **Add Library**.



b. In the **Add Library** dialog click on **Creat**e button to define a new library to support Bean Validation in Java SE Project

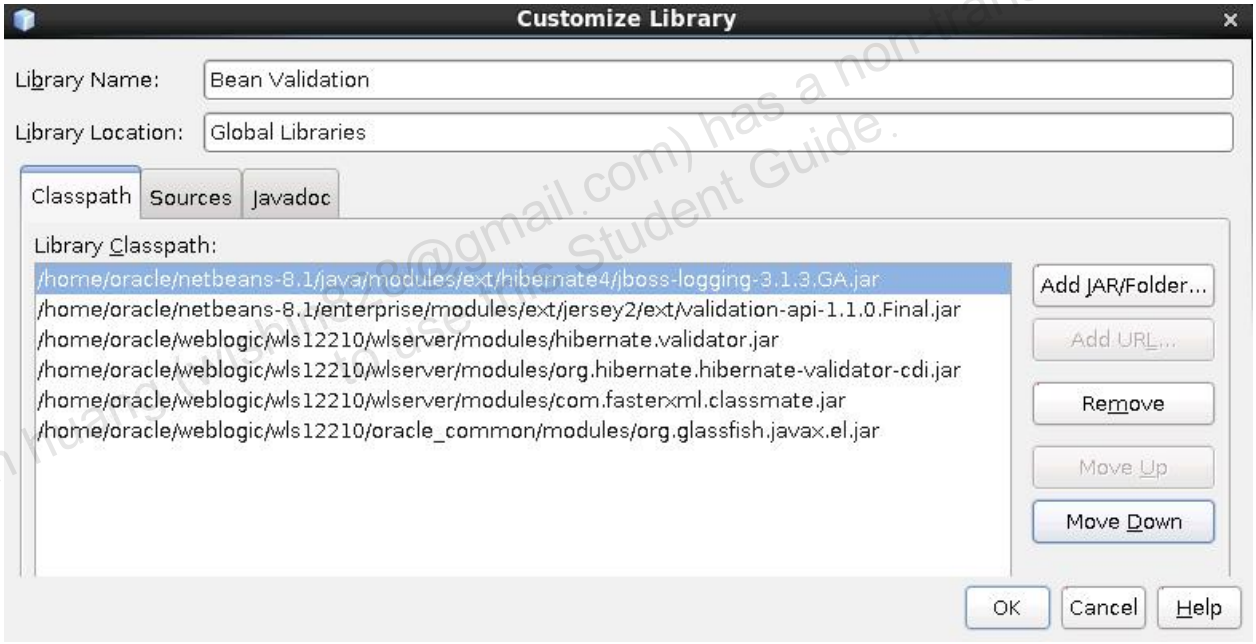
c. In the **Create New Library Dialog** set Library Name: `Bean Validation`



d. Click **Ok**.

e. In the **Customize Library** dialog add following jar files to this library (use **Add JAR/Folder** button to add each jar) :

```
/home/oracle/netbeans-8.1/java/modules/ext/hibernate4/
jboss-logging-3.1.3.GA.jar
/home/oracle/netbeans-8.1/enterprise/modules/ext/jersey2/ext/
validation-api-1.1.0.Final.jar
/home/oracle/weblogic/wls12210/wlserver/modules/
hibernate.validator.jar
/home/oracle/weblogic/wls12210/wlserver/modules/
org.hibernate.hibernate-validator-cdi.jar
/home/oracle/weblogic/wls12210/wlserver/modules/
com.fasterxml.classmate.jar
/home/oracle/weblogic/wls12210/oracle_common/modules/
org.glassfish.javax.el.jar
```



f. Click **OK**.

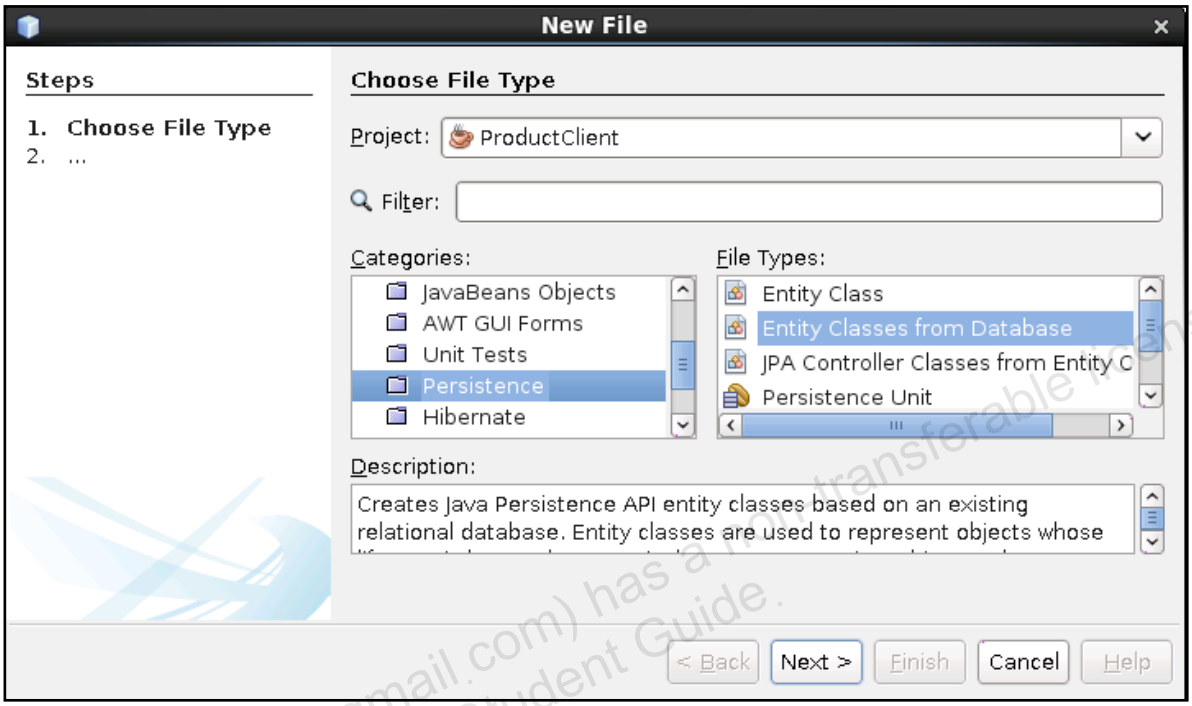
g. In the **Add Library** dialog box, select:

- Java DB Driver
- Java EE 7 API Library
- Bean Validation

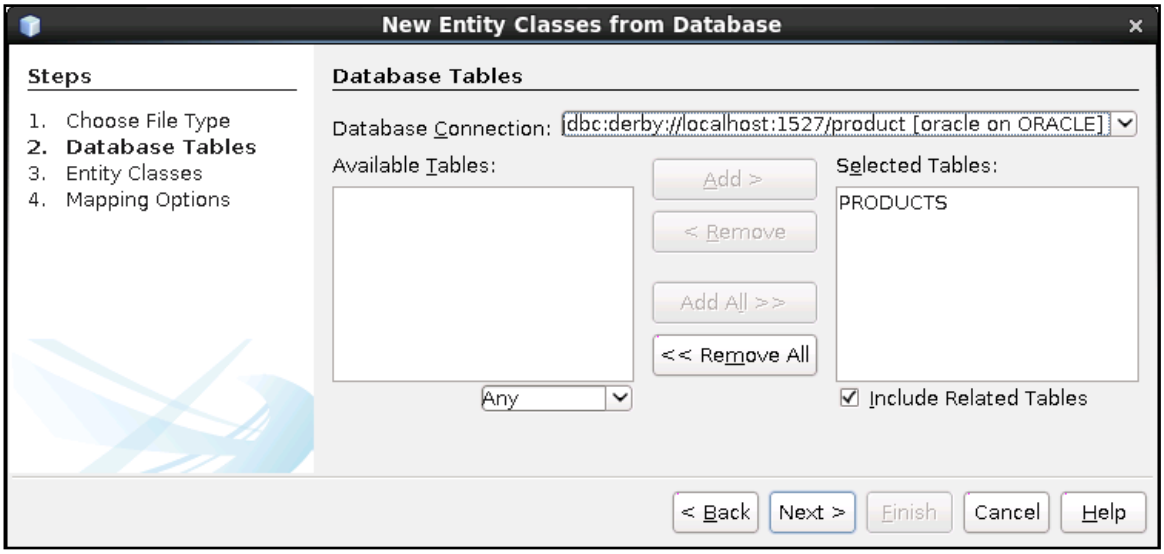
h. Click the **Add Library** button.

Note: In this practice, you have created what is essentially a Java SE project. However, you are going to use it to create some Java EE code and test components that you will later deploy into the Java EE server. This is why you had to add additional libraries and jar files to it.

3. Create a JPA entity class.
   a. Select **File > New File**.
   b. Select **Persistence** from the Categories list and **Entity Classes from Database** from the Files Types list.



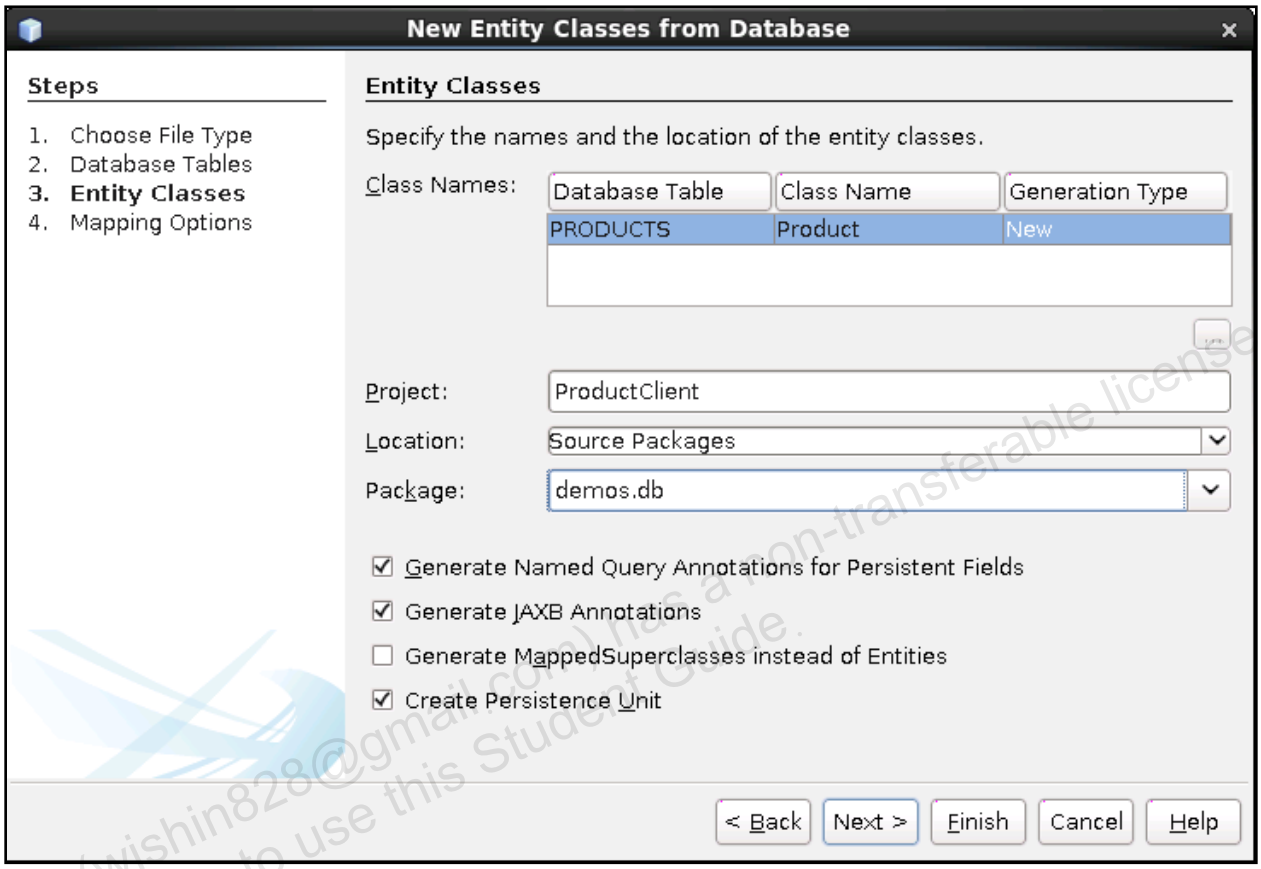
   c. Click **Next**
   d. Select the database connection you created in the previous practice from the drop-down list. The `PRODUCTS` table should appear in the list of Available Tables.
   e. Select the `PRODUCTS` table and click the **Add** button to move it to the Selected Tables list.



   f. Click **Next**.

g. Set Entity Classes generation properties:

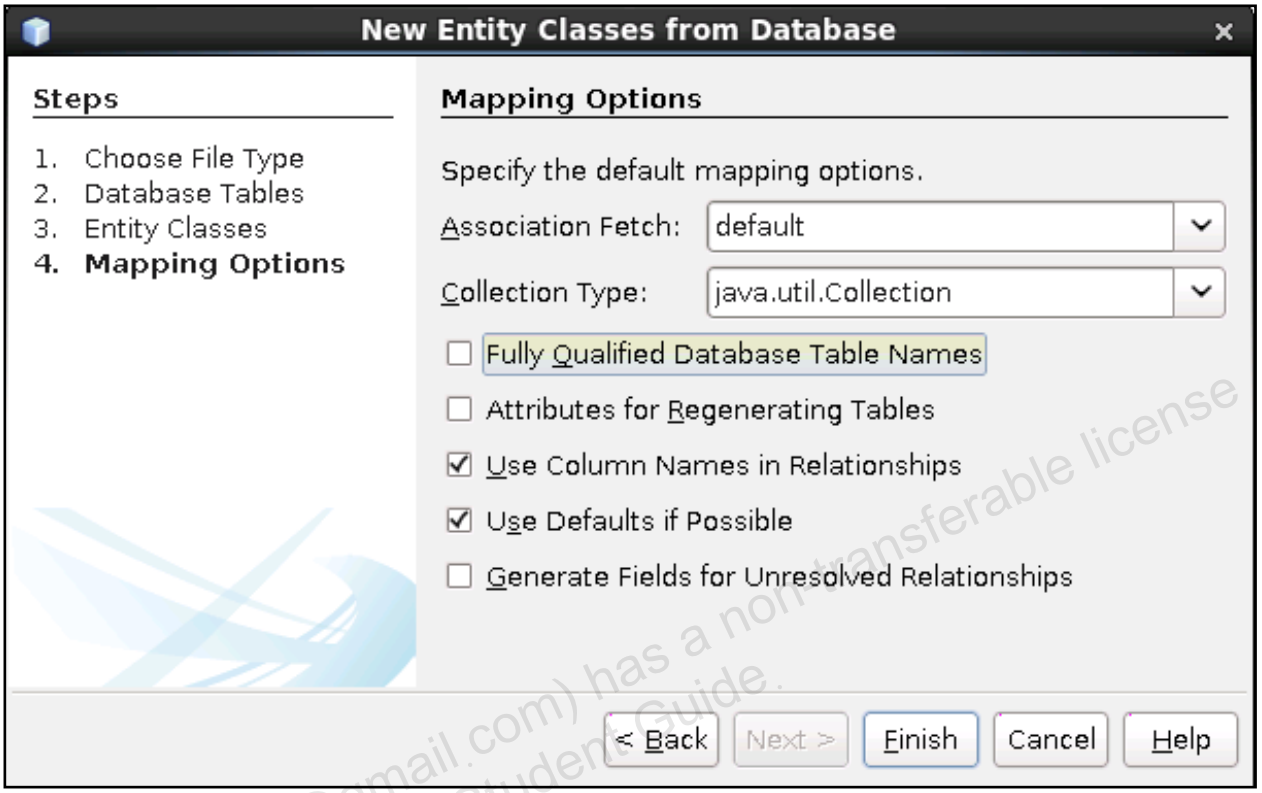
- Change Class Name to: `Product`
- Change Package to: `demos.db`



h. Click **Next.**

i. Set Mapping Options:

- Select the **Use Defaults if Possible** check box.

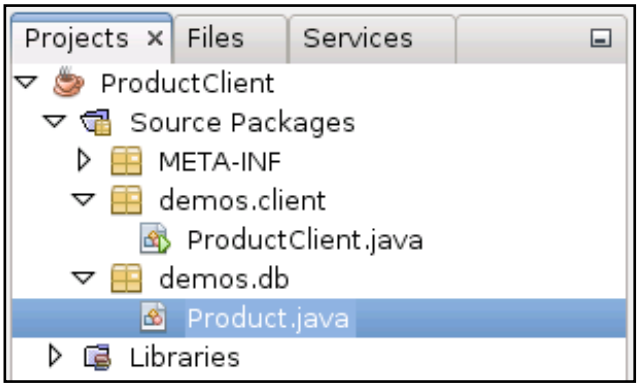


j. Click **Finish**.

Note: A Persistence Unit configuration file, `persistence.xml`, was created in a `META-INF` folder, and a JPA Entity class, `demos.db.Product`, was created.

4. Modify the `toString` method of the `Product` class to return information about all of the product properties.

a. Expand the `demos.db` package folder.

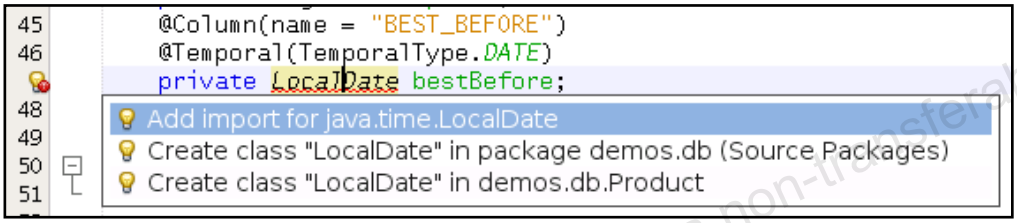


b. Double-click the `Product.java` file to open the editor.

c. Find the `toString` method in the `Product` class and modify it to return concatenation of all product properties:

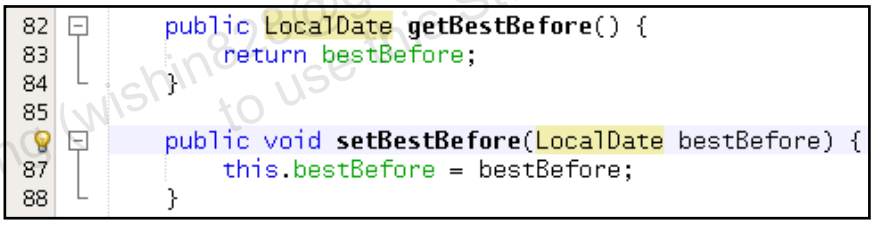
```
@Override
public String toString() {
    return id+" "+name+" "+price+" "+bestBefore+" "+version;
}
```

5. Change the type of the `bestBefore` field to: `LocalDate`
   a. Type `LocalDate` instead of `Date`.
   b. Click the light bulb icon that appears on this line of code.
   c. Click **Add import for `java.time.LocalDate`**.



Note: The type of the `bestBefore` field has been changed. However, getter and setter methods for this field are still referring to the old type, `Date`.

   d. Change the return type of the getter method and the parameter type of the setter method for the `bestBefore` field to `LocalDate`.



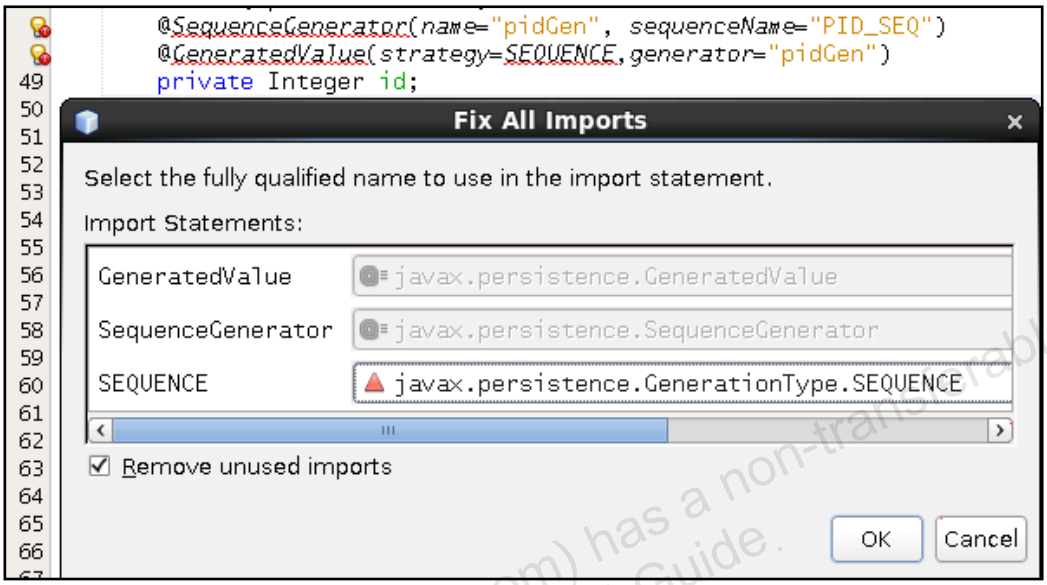
Note: Although you have modified the Product entity class to use `LocalDate` instead of the Date class, the current version of the JPA ORM provider (EclipseLink) does not support mapping of this type. You will address this problem by providing a converter for this type later in this practice.

6. Enable the generation of a primary key value from the sequence:
   a. Add code just before the declaration of the `private Integer id;` field:

```
@SequenceGenerator(name="pidGen",sequenceName="PID_SEQ",
                   allocationSize=1)
@GeneratedValue(strategy=SEQUENCE,generator="pidGen")
```

b. Right-click anywhere in the source code editor, press **CTRL+SHIFT+I** to invoke the Fix Imports menu, and add the following imports:

```
javax.persistence.GeneratedValue
javax.persistence.SequenceGenerator
javax.persistence.GenerationType.SEQUENCE
```



Note: Remember these ways of adding imports to Java source code: using the Fix Imports menu or clicking the light bulb icon on the right side of the line of code.

7. Add version attribute handling to support optimistic locking:
   a. Add the `@Version` annotation on a new line of code just before the declaration of the `private Integer version;` field.
   b. Add an import: `javax.persistence.Version`

8. Add Beans Validation constraints to the fields:
   a. Add a new line of code in front of the `private Integer id;` field, and add the annotation `@NotNull` to it.
   b. Add an import: `javax.validation.constraints.NotNull`.
   c. Add two new lines of code in front of the `private String name;` field and add the following annotations:

```
@NotNull
@Size(min=3, max=40, message="{prod.name}")
```

   d. Add an import: `javax.validation.constraints.Size`
   e. Uncomment the line of code in front of the `private BigDecimal price;` field, and modify the Max and Min annotations:

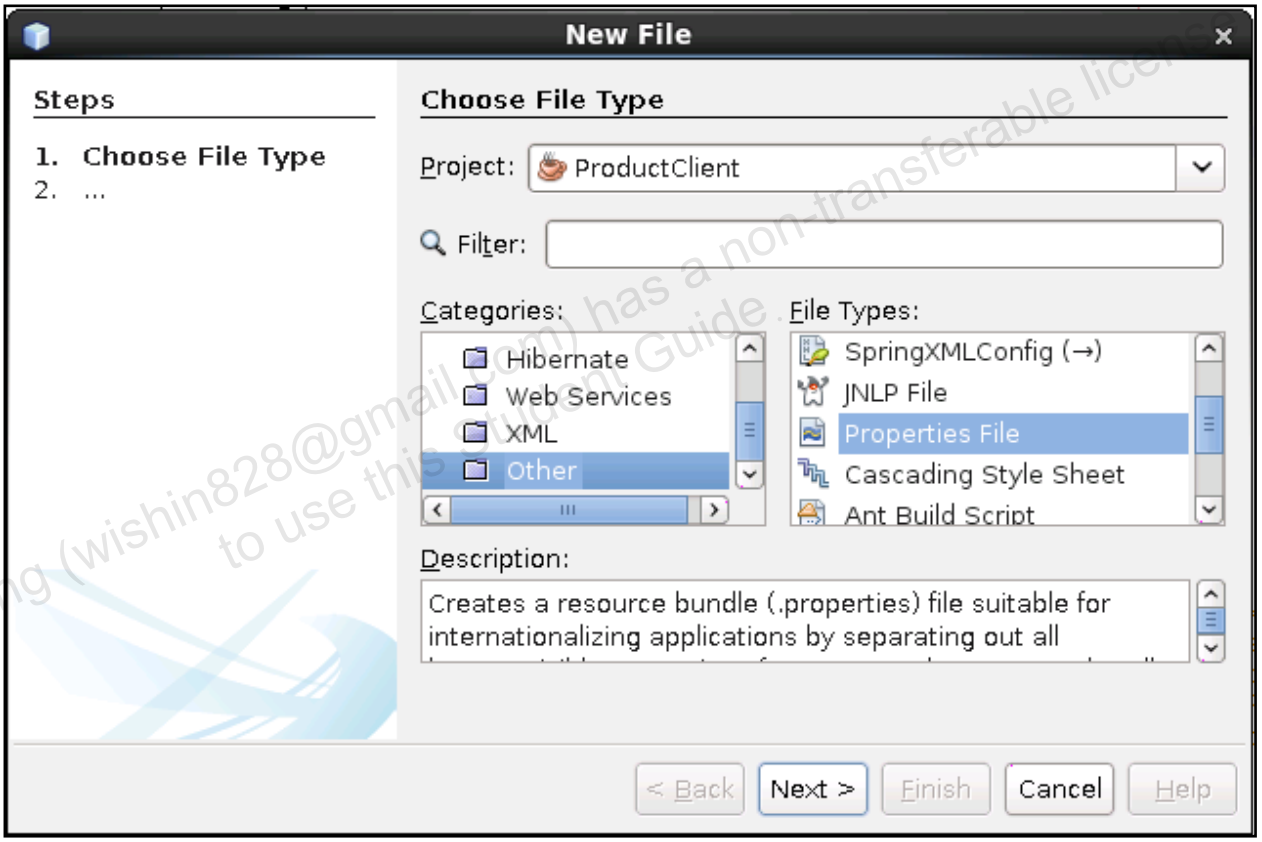
```
@Max(value=1000, message="{prod.price.max}")
@Min(value=1, message="{prod.price.min}")
```

f. Add two imports:

```
javax.validation.constraints.Max
javax.validation.constraints.Min
```

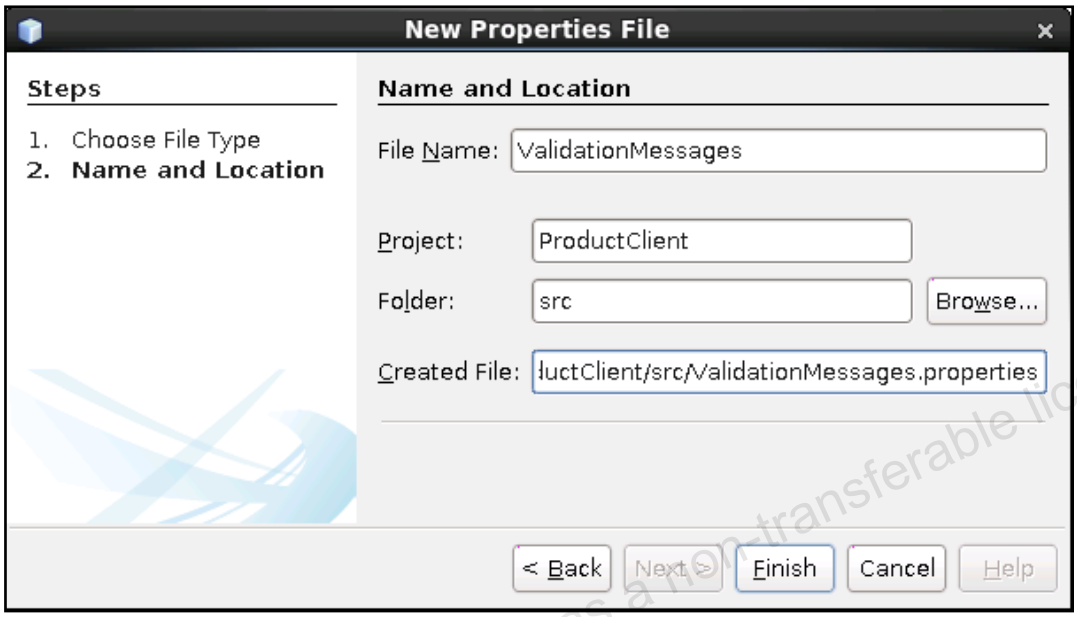
Note: The Bean Validation constraints that you have just introduced to the Product class refer to resource bundle messages. You will create this resource bundle and add messages to it next.

9. Create resource bundle to customize Bean Validation messages:
   a. Select **File > New File**.
   b. Select **Other** in the Categories list and **Properties File** in the File Types list.



   c. Click **Next.**

d. Set New Properties File properties:
   - File Name: `ValidationMessages`
   - Folder: `src`

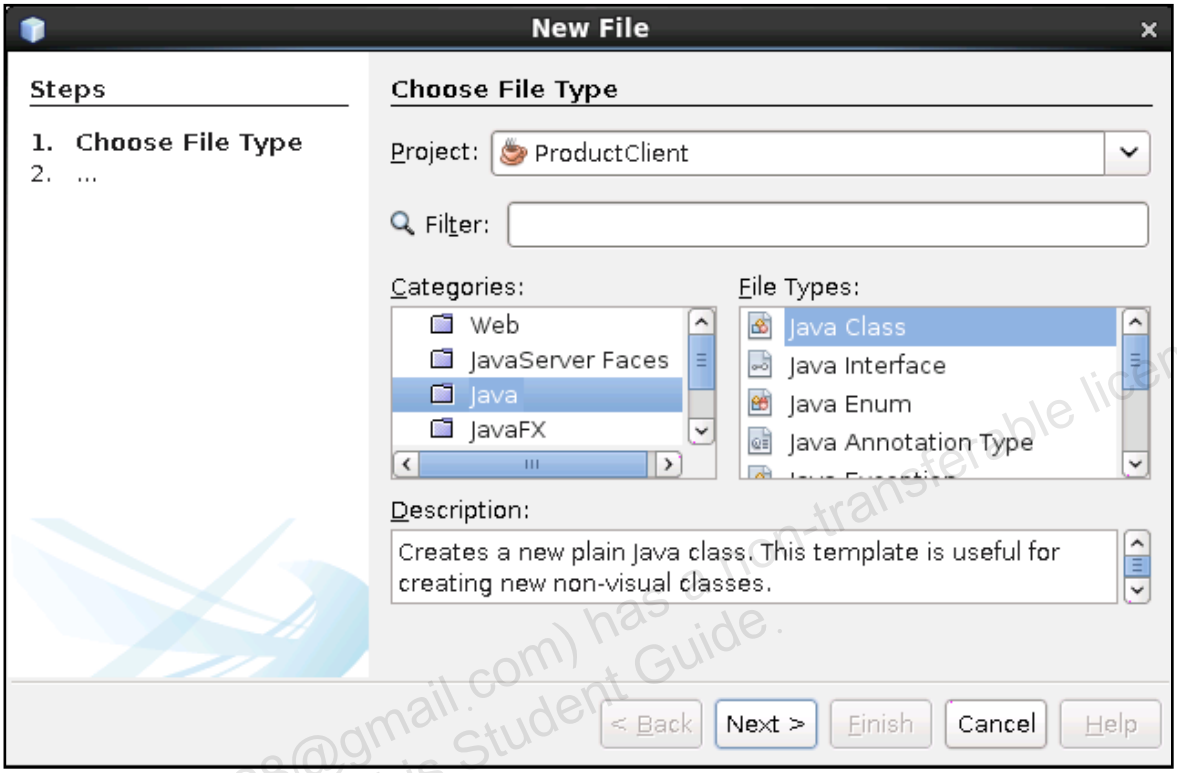


e. Click **Finish**.

f. Add the following messages to the `ValidationMessages.properties` file:

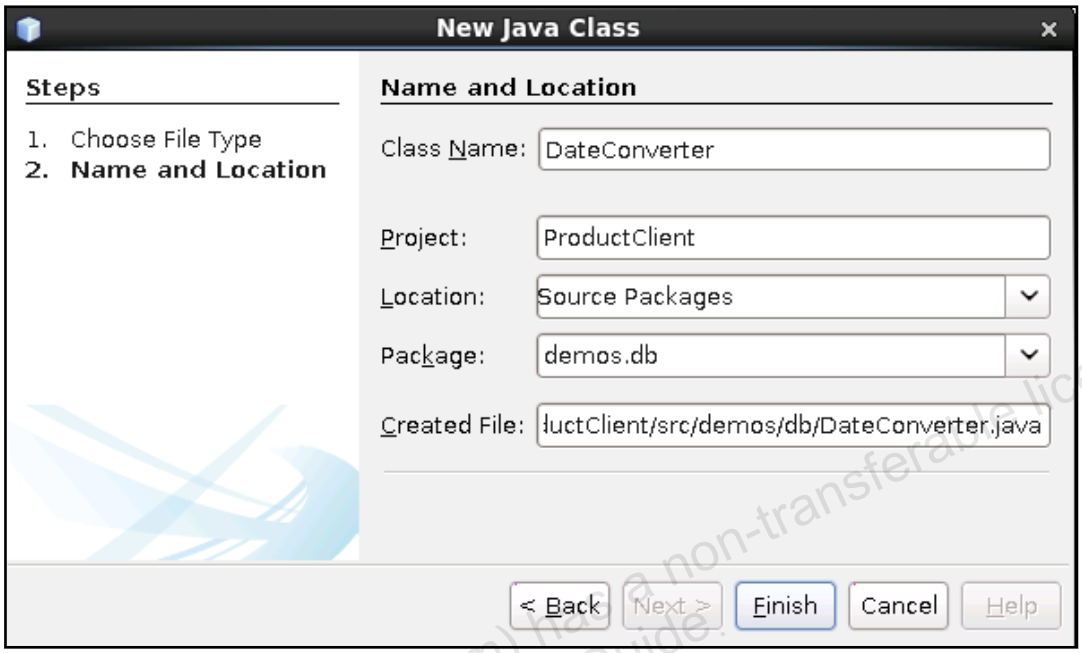
```
prod.name=Name cannot be shorter than {min} or longer than {max}
prod.price.max=Price must be smaller than {value}
prod.price.min=Price must be greater than {value}
```

10. Create a converter class to handle conversion between `LocalDate` and `Date` types.
    a. Select **File > New File**.
    b. Select **Java** from the Categories list and **Java Class** from the File Types list.



    c. Click **Next**.

d. Set New Java Class properties:
   - Class Name: `DateConverter`
   - Package: `demos.db`



e. Click **Finish**.

f. Open the `DateConverter` class.

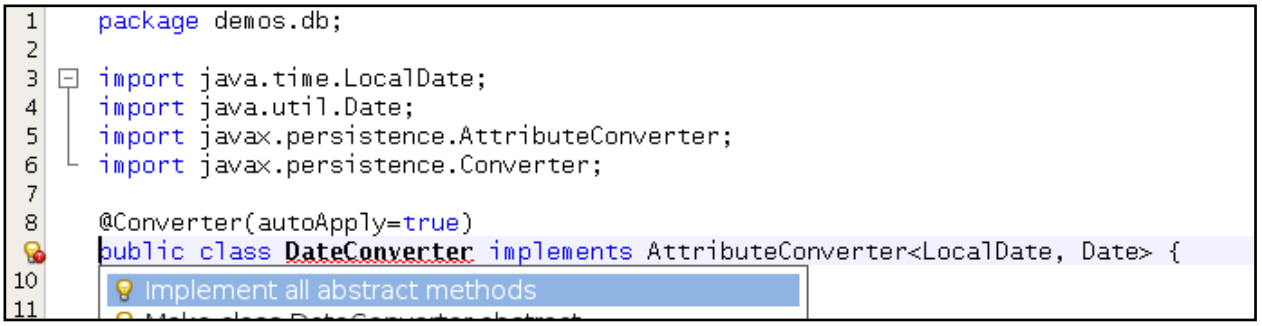
g. Make **`DateConverter`** implement the **`AttributeConverter`** interface to perform conversions between **`Date`** and **`LocalDate`** types. Also annotate this class with the **`Converter`** annotation and set the value of **`autoApply`** to true:

```
@Converter(autoApply=true)
public class DateConverter
             implements AttributeConverter<LocalDate, Date> {
    // you will add implementation code here
}
```

h. Add the following imports:

```
javax.persistence.AttributeConverter
javax.persistence.Converter
java.time.LocalDate
java.util.Date
```

i. Click the light bulb error icon located on the left side of the class definition line of code, and invoke the **implement all abstract methods** menu.



j. Rename the method parameters so that they are both called **`value`**.

This is what these two conversion method signatures should look like:

```
@Override
public Date convertToDatabaseColumn(LocalDate value){
    // you will add conversion of LocalDate to Date code here
}
@Override
public LocalDate convertToEntityAttribute(Date value){
    // you will add conversion of Date to LocalDate code here
}
```

Note: Comments inside these operations represent code you are now going to replace with actual implementation logic.

k. Replace code inside the **`convertToDatabaseColumn`** method body with the code that converts **`LocalDate`** to **`Date`**. Application date attributes can be null, so in this case do not attempt to perform conversion.

```
return (value==null) ? null :
Date.from(value.atStartOfDay(ZoneId.systemDefault()).toInstant());
```

l. Add an import: `java.time.ZoneId`

m. Replace code inside the `convertToEntityAttribute` method body with the code that converts `Date` to `LocalDate`. Application date attributes can be null, so in this case do not attempt to perform conversion.

```
return (value==null) ? null : Instant.ofEpochMilli(value.getTime())
.atZone(ZoneId.systemDefault()).toLocalDate();
```

n. Add an import: `java.time.Instant`

11. Register the converter class with persistence unit configuration:
    a. Expand **Source Packages > META-INF node**.
    b. Double-click the **`persistence.xml`** file.
    c. Click the **Source** button.

d. Add a new line of code immediately after the line that registers the Product class `<class>demos.db.Product</class>`, to register the `DateConverter` class:

`<class>demos.db.DateConverter</class>`

Note: `DateConverter` class is not an entity, but it contains annotations that have to be processed by the ORM provider that handles entities.

12. Modify the Named Query defined by the Product Entity class to select products with similar names:
    a. Locate a line of code that defines a Named Query called `Product.findByName` and change it to use an operator like in the `WHERE` clause:

    `"SELECT p FROM Product p WHERE p.name like :name"`

    b. Add a new Named Query definition to the list of named queries. This query should be called `Product.findTotal`. This query should return a counter of products' ID values and a sum of product price values for all products with ID matching ID values supplied as a parameter.

```
@NamedQuery(name="Product.findTotal", query=
"SELECT count(p.id), sum(p.price) FROM Product p WHERE p.id in :ids")
```

13. Compile the ProductClient project:
    a. Click the **Clean and Build Project** icon on the toolbar, or press **SHIFT+F11**.

    b. The output window should display a BUILD SUCCESSFUL message.

Note: You may ignore warning messages. But if you get compiler errors, these will have to be fixed.

Practice 3-2: Creating a JPA Controller

Overview

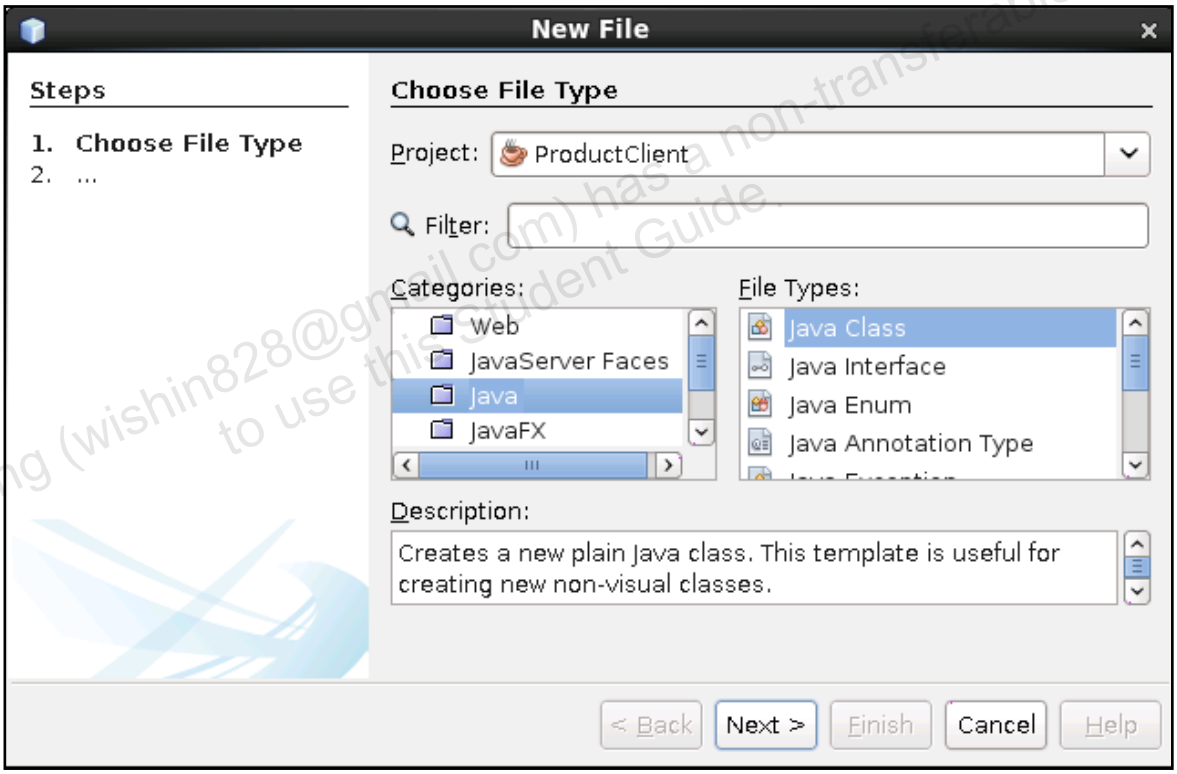
In this practice, you create a new JPA Controller to handle interactions with the database using the JPA entity Product.

Assumptions

You have successfully completed all previous practices.

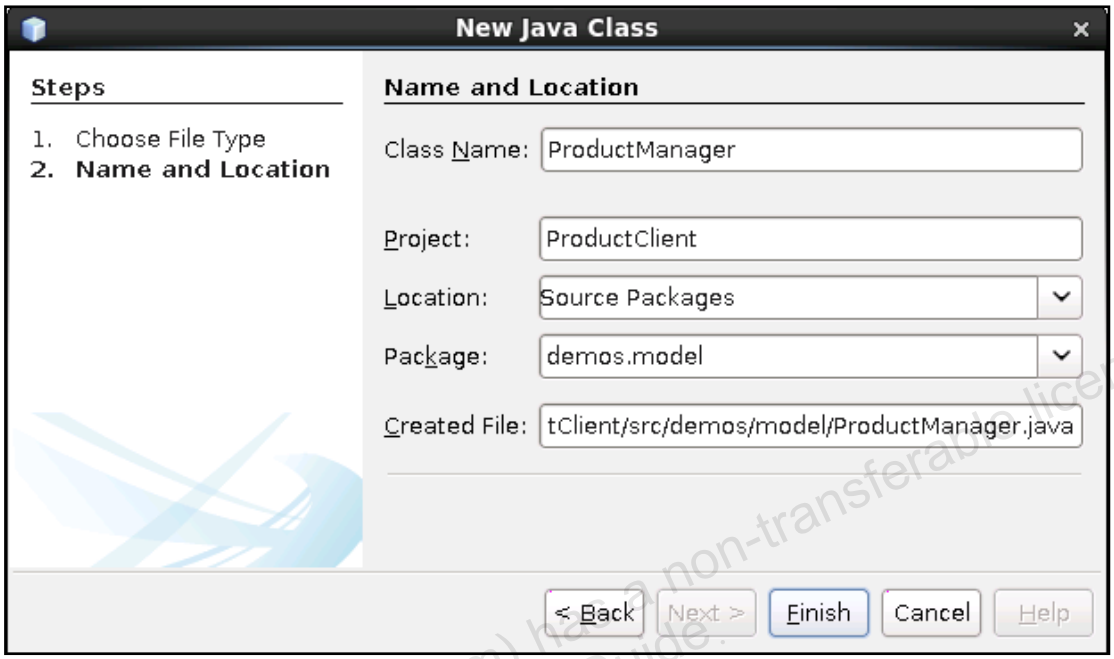
Tasks

1. Create a new JPA Controller class to handle Product Entity database interactions:
   a. Select **File > New File**.
   b. Select **Java** from the Categories list and **Java Class** from the File Types list.



   c. Click **Next**

d. In the New Java Class dialog box, set following properties:
   - Class Name: `ProductManager`
   - Package: `demos.model`



e. Click **Finish**.

2. Modify the `ProductManager` class to add instance variables:
   a. Add EntityManager instance variable to ProductManager class:
      `private EntityManager em;`
   b. Add an import: `javax.presistence.EntityManager`
   c. Add Query instance variable to the ProductManager class:
      `private Query productNameQuery;`
   d. Add an import: `javax.presistence.Query`

3. Add initialization and closure code for the EntityManager object and for one of the Named Queries defined by the Product entity:
   a. Add constructor to the ProductManager class to initialize the EntityManager and Query objects. This constructor should accept the persistence unit name as a String parameter:

```
public ProductManager(String persistenceUnit) {
EntityManagerFactory emf =
      Persistence.createEntityManagerFactory(persistenceUnit);
em = emf.createEntityManager();
productNameQuery = em.createNamedQuery("Product.findByName");
}
```

b. Add two imports:

```
javax.persistence.Persistence
javax.persistence.EntityManagerFactory
```

c. Add an operation to close the EntityManager:

```
public void closeEntityManager() {
    em.close();
}
```

4. Modify the ProductManager class to add operations to create, update, delete, and find products:

a. Add a `create` operation that accepts a Product object as an argument. This operation should use EntityManager object to begin the transaction, and then persist the Product entity object and commit the transaction:

```
public void create(@Valid Product product) {
    em.getTransaction().begin();
    em.persist(product);
    em.getTransaction().commit();
}
```

b. Add imports of `demos.db.Product` and `javax.validation.Valid` classes.

c. Add an `update` operation that accepts a Product object as an argument. This operation should use the EntityManager object to begin the transaction, and then merge the Product entity object and commit the transaction:

```
public void update(@Valid Product product) {
    em.getTransaction().begin();
    product = em.merge(product);
    em.getTransaction().commit();
}
```

d. Add a `delete` operation that accepts a Product object as an argument. This operation should use the EntityManager object to begin the transaction, and then remove the Product entity object and commit the transaction:

```
public void delete(Product product) {
    em.getTransaction().begin();
    em.remove(product);
    em.getTransaction().commit();
}
```

e. Add a `findProduct` operation that accepts an Integer product ID as an argument. This operation should use a EntityManager object to find the Product with a specific ID:

```
public Product findProduct(Integer id) {
    return em.find(Product.class,id);
}
```

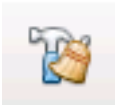
f. Add a `findProductByName` operation that accepts a String product name as an argument. This operation should use EntityManager set `productNameQuery` parameter name, and then execute the query and return the results as a list.

```
public List<Product> findProductByName(String name) {
    productNameQuery.setParameter("name",name);
    return productNameQuery.getResultList();
}
```

g. Add an import: `java.util.List`

Note: Java EE frameworks that handle user interface–related functionalities, such as Java Server Faces, can automate the handling of the JSR-303 Bean Validation constraints. Later in the course you will be using these frameworks. However, in this project your UI is going to be a simple Java SE application launched from the command line; therefore, it is not going to use any of the sophisticated frameworks that automatically implement constraint validations. In this practice, you added a `@Valid` annotation to parameters of the create and update operations to trigger Bean Validation before JPA attempts to perform database insert or update.

5. Compile the ProductClient project.
   a. Click the **Clean and Build Project** icon on the toolbar, or press **SHIFT+F11**.



   b. The output window should display a BUILD SUCCESSFUL message.

      Note: You may ignore warning messages. But if you get compiler errors, these will have to be fixed.

Practice 3-3: Testing JPA Functionalities

Overview

In this practice, you create code in the `ProductClient` class to test functionalities of the JPA application.

Assumptions

You have successfully completed all previous practices.

Tasks

1. Add code to the `ProductClient` class to set up Logger:
   a. Add and initialize static Logger object reference to be used by this class:
   ```
   private static final Logger logger =
   Logger.getLogger(ProductClient.class.getName());
   ```
   b. Add import: `java.util.logging.Logger`.

2. Add exception handling logic to `ProductClient` class to catch errors related to constraint validations, optimistic locking, and other possible problems.
   a. Inside this main method of the `ProductClient` class, create a try block.
   ```
   try {
   // product handling logic will be placed here
   }
   ```
   Note: The comment line in the example indicates the place where the code that will test JPA product entity should be added later in this practice.
   b. After the `try` block add a `catch` block that handles all exceptions. You should check what the cause of the exception is. It could be `ConstraintViolationException`, `OptimisticLockException`, or anything else.
   ```
   catch (Exception ex) {
        Throwable cause = ex.getCause();
        // exception handling will be placed here
   }
   ```
   Note: The comment line in the example indicates the place where the code that will handle exceptions should be added later in this practice.

c. Inside the catch, after you have extracted the exception cause, add an `if` condition to check if `ConstraintViolationException` was the cause of the problem. There could be more than one Constraint Violation, so you need to iterate through the set of constraint violations and write information about them to the log.

```
if (cause instanceof ConstraintViolationException) {
     ConstraintViolationException e =
                  (ConstraintViolationException)cause;
     e.getConstraintViolations().stream()
          .forEach(v -> logger.log(Level.INFO,v.getMessage()));
}
```

d. Add imports of `javax.validation.ConstraintViolationException` and `java.util.logging.Level` classes.

e. After the end of this `if` block, add an `else` block. This `else` block should contain another `if` block that should check if the cause of the exception was an `OptimisticLockException`. You should also write information about this exception to the log file.

```
} else if (cause instanceof OptimisticLockException) {
     OptimisticLockException e =(OptimisticLockException)cause;
     logger.log(Level.INFO, e.getMessage());
}
```

f. Add an import: `javax.persistence.OptimisticLockException`

g. After the end of this `if` block add another `else` block. This last `else` block will be triggered if the exception was caused by any other problem. You should just write information about the exception to the log.

```
} else {
     logger.log(Level.SEVERE, "Product Manager Error",ex);
}
```

Note: Constraint violations and optimistic lock exceptions are relatively "normal" behaviors of the program. These are the issues that user can "fix" by supplying correct values, or by re-selecting the object from the database. However, any other issues indicate a more serious problem with the code. This is the reason why you should use different logging levels for reporting these exceptions.

3. Add code to the method main of the `ProductClient` class to perform initialization and closure of the `ProductManager` JPA controller class:

a. Inside the `try` block of the main method of the `ProductClient` class, add the following code:

```
ProductManager pm = new ProductManager("ProductClientPU");
     // all your test code will be added here
pm.closeEntityManager();
```

b. Add an import: `demos.model.ProductManager`

4. Add code that finds and prints product with ID 1:
   a. After the line of code that initializes the ProductManager object and before the line of code that closes the Entity Manager, add code that calls the `findProduct` operation of the ProductManager to find product with ID 1 and then print this product to the console.

```
Product p = pm.findProduct(1);
System.out.println(p);
```

   b. Add an import: `demos.db.Product`

5. Execute the `ProductClient` Java class to test the application.

   a. Click the **Run Project** button, or press **F6**.
   b. You should see information about a product Cookie printed onto the console.

6. Replace code that printed a single product with code that prints all products which name starts with "Co":
   a. Comment out code created in the previous step.

```
// Product p = pm.findProduct(1);
// System.out.println(p);
```

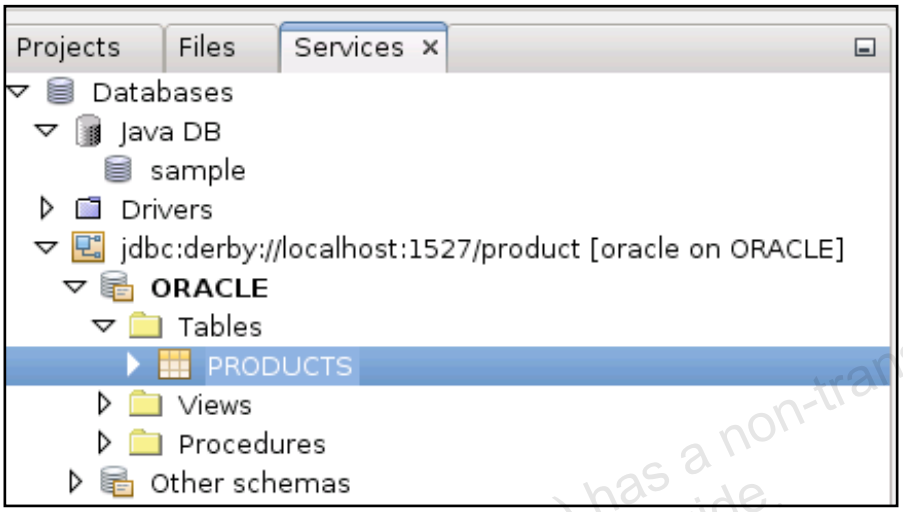
   Note: To comment or uncomment a number of lines of code in NetBeans, you can select these lines and press **CTRL+/** on the keyboard.
   b. Add code that invokes the `findProductsByName` method of the `ProductManager` class and process the list of results.

```
List<Product> products = pm.findProductByName("Co%");
products.stream().forEach(p -> System.out.println(p));
```

   c. Add an import: `java.util.List`

7. Run this project again. You should see information about products Coffee and Cookie printed onto the console.

8. Open SQL console to view data in the `PRODUCTS` table:
   a. Click the Services panel tab.
   b. Expand the Databases node.
   c. Expand jdbc connection to the product database.
   d. Expand the `ORACLE` schema.
   e. Expand Tables.



   f. Right-click the **`PRODUCTS`** table and select **View Data**.
   g. Notice details of the first row. In the next practice step you will write code in the `ProductClient` class to update this record and then observe changes.

9. Replace code that queried all products which name starts with "Co", with the code that finds, changes, validates and updates the first product:
   a. Comment out code created in practice 3-3 step 6.b.
   b. Use the `findProduct` method to find product with ID 1. Set this product price to be 2.5 and best before date to be tomorrow.

```
Product p = pm.findProduct(1);
p.setPrice(BigDecimal.valueOf(2.5));
p.setBestBefore(LocalDate.now().plusDays(1));
pm.update(p);
```

   c. Add imports of `java.math.BigDecimal` and `java.time.LocalDate`.
   d. Run this project again.
   e. Click the SQL console and rerun the query using the **Run SQL** button, or by pressing **CTRL+SHIFT+E**.
   f. Observe changes to the first record. Price, best before, and version columns were modified.

10. Change values so that the product update will trigger validation errors.
    a. Set this product price to be 0.1 and product name to be "x", leave the rest of the code the same.

```
Product p = pm.findProduct(1);
p.setPrice(BigDecimal.valueOf(0.1));
p.setName("x");
p.setBestBefore(LocalDate.now().plusDays(1));
pm.update(p);
```

    b. Run this project again.
    c. Observe messages logged to the Java Console:

```
INFO: Price must be greater than 1
INFO: Name cannot be shorter than 3 or longer than 40
```

    d. You may also rerun SQL query via the SQL console to see that the record has not been changed.

11. Prevent record update due to the Optimistic Lock exception:
    a. Change values within the code of the main method of the ProductClient class so it will find and update record number 1, setting its price to 5.

```
Product p = pm.findProduct(1);
p.setPrice(BigDecimal.valueOf(5));
     // more code will be added here in the next step
pm.update(p);
```

    b. Add code to pause this program between the line of code that makes changes to the product and the actual update:

```
Scanner s = new Scanner(System.in);
System.out.println("Click here and then press enter to continue");
s.nextLine();
```

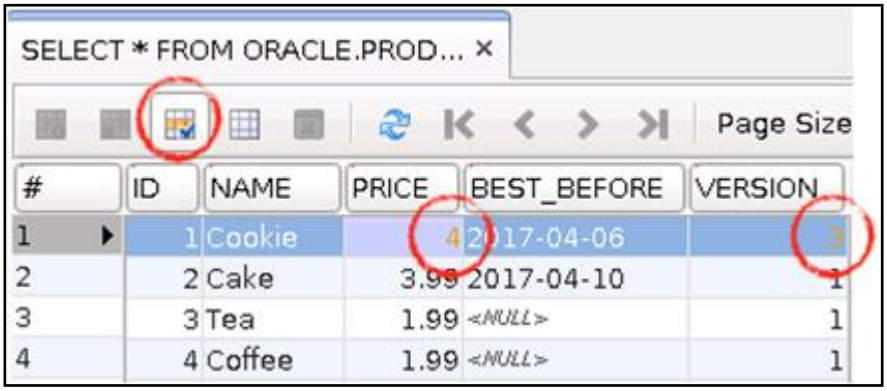
    c. Add import: `java.util.Scanner`
    d. Run this project again. The console will pause.

e. Switch to the SQL Console, select first record, change its price to 4 and change version to be 3.

Note: Version number has to be incremented by one.

f. After you modified the fist record, click it.

g. Click the **Commit Records** button in the SQL console.



h. Click inside the ProductClient(run) log window and press **Enter**.

i. Observe that an Optimistic Lock Exception is produced.

Note: Optionally (if you have time) you may also write code to test create and delete methods of the `ProductManager` class.

Practices for Lesson 4: Implementing Business Logic by Using EJBs

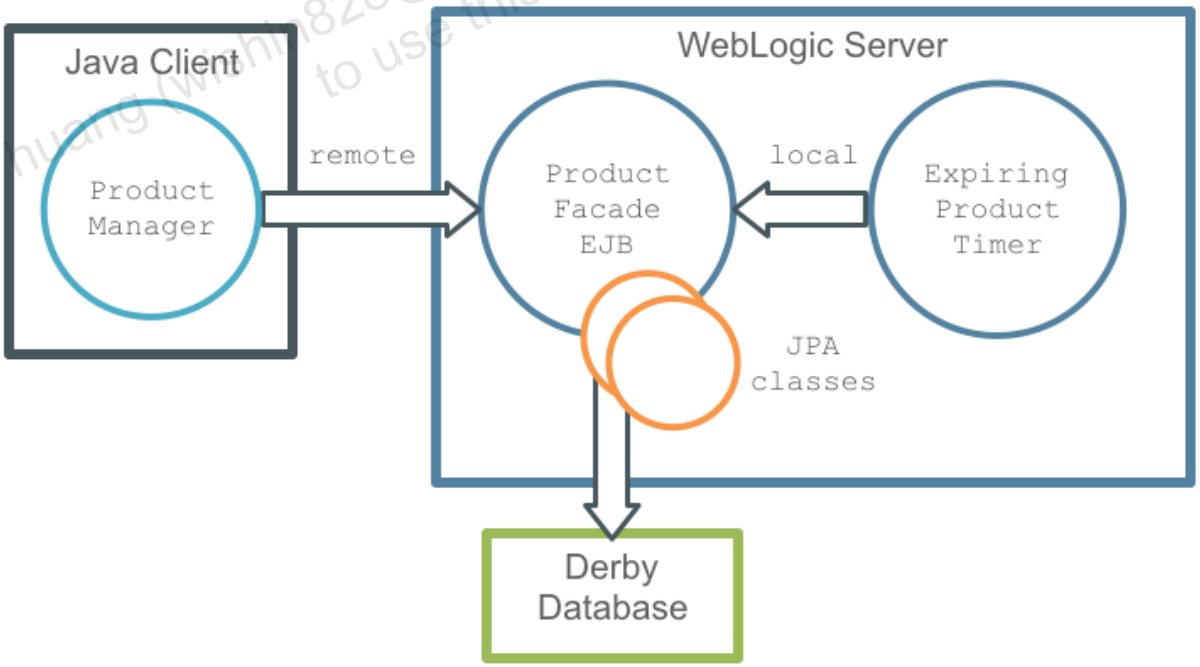
Practices for Lesson 4: Overview

Overview

In these practices, you implement Product Management application business logic with a new Java EE application containing an Enterprise Java Beans module.

In this practice, you will create a Java EE application deployed to the WebLogic Server. This application contains an EJB Module that performs operations on the same JPA objects as the client Java application did in the previous practice. However, this time your JPA entities will be handled by the ProductFacade EJB.

- ProductFacade is a Stateless Session Bean that controls persistence.
- You use exactly the same JPA classes (Product Entity and DateConverter) as in the previous practice, but this time they are deployed within the Java EE Server, rather than the client.
- The `ProductManager` class that was previously managing persistence within the client now plays a different role: it acts as a proxy to help the client access a remote ProductFacade EJB.
- You also create an ExpiringProduct Singleton Timer EJB. It interacts with ProductFacade using local calls.



Practice 4-1: Creating an EJB Module

Overview

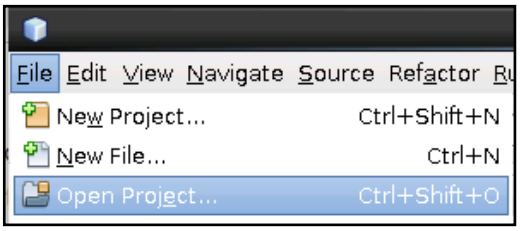
In this practice, you define a new Enterprise Java Bean that will contain product management application business logic, using same JPA code as you have created in “Practices for Lesson 3.”

Assumptions

You have successfully completed all previous practices.

Tasks

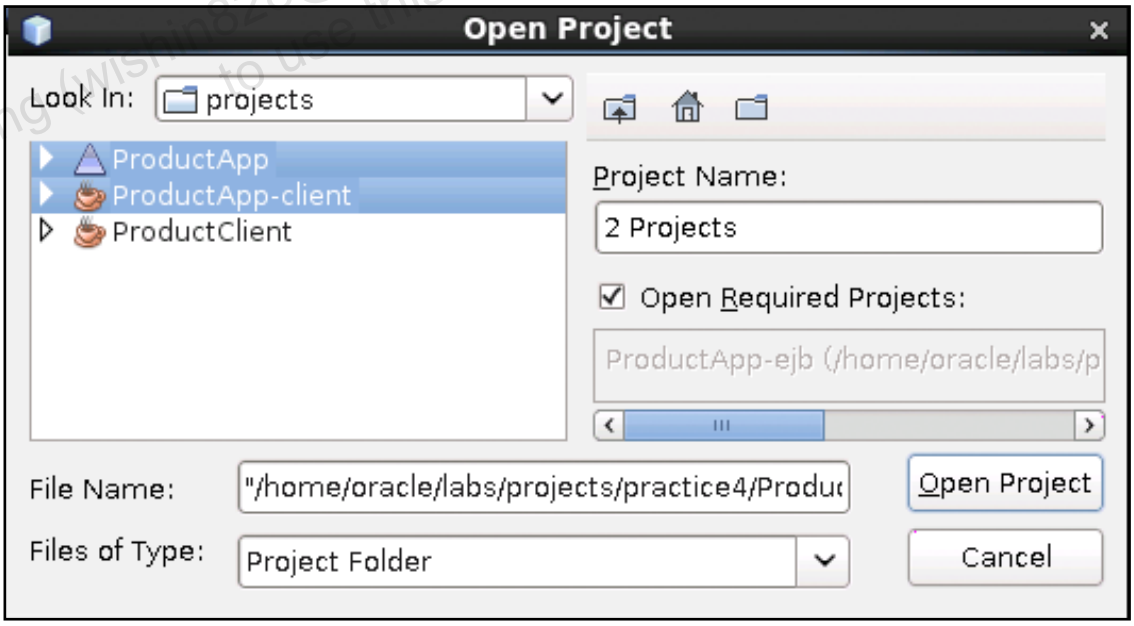
1. Open a Java Enterprise Application with an EJB Module and a ProductApp-client project:



   a. Select File > Open Project.

   b. Select the `ProductApp` and `ProductApp-client` projects from the `/home/oracle/labs/projects` folder.

   c. Select the **Open Required Projects** check box.

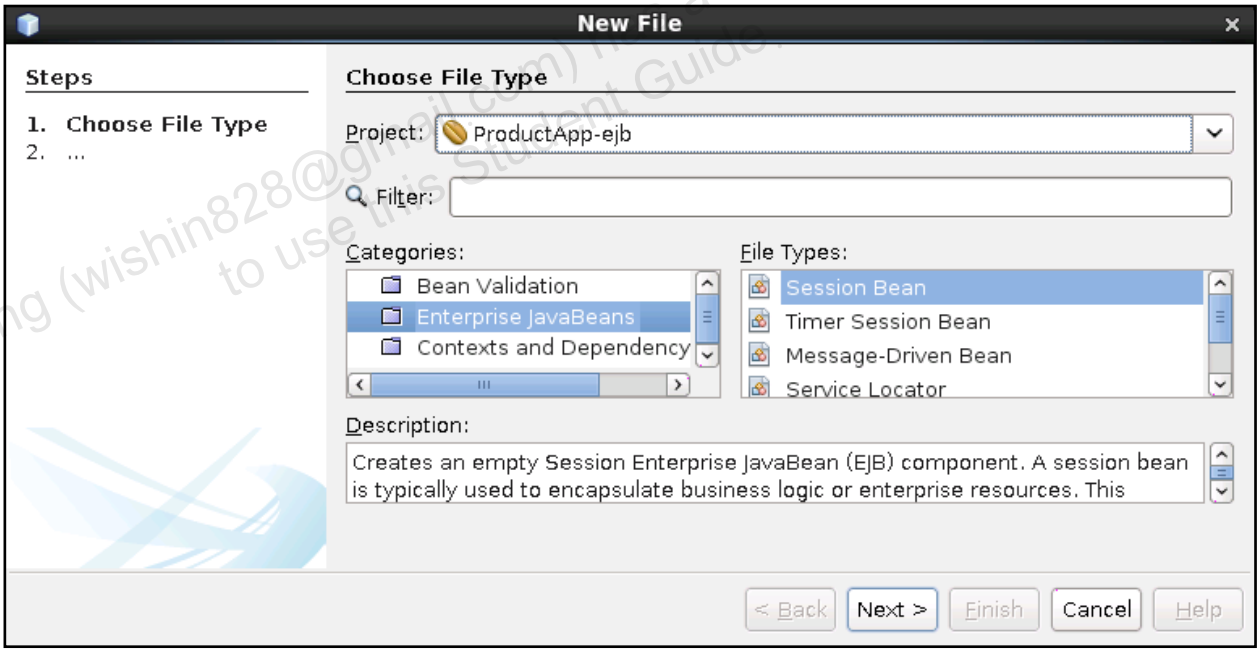


   d. Click **Open Project**.

Note: You have just opened the following three projects:

- Project ProductApp represents an Enterprise Application, deployed as an EAR file.
- Project ProductApp-ejb represents the EJB Module included in the ProductApp as an EJB-JAR file. This project contains a copy of the same JPA code that you created in "Practice for Lesson 3." You will find the `persistence.xml` file, the `ValidationMessages.properties` file, and `DateConverter` and `Products` classes. You will create an Enterprise Java Bean in this project that will manage the JPA logic of the application.
- Project ProductApp-client contains `ProductClient` class with method `main` and `ProductManager` class. All JPA handling code has been removed from the `ProductManager` class. Later in this practice you will add code to this class to invoke an Enterprise Java Bean from the ProductApp-ejb project instead.

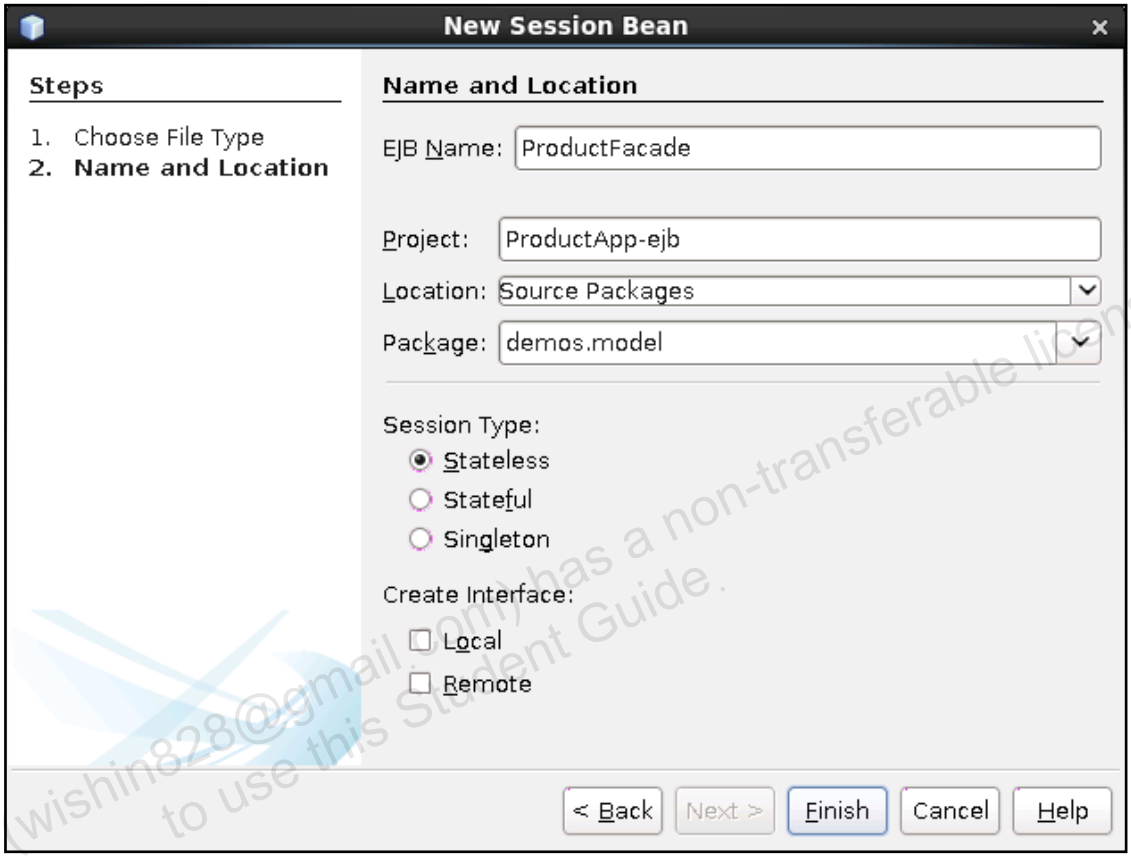
2. Create a new Enterprise Java Bean for handling product JPA objects:
   a. Select the **ProductApp-ejb** project.
   b. Select **File > New File**.
   c. Select **Enterprise JavaBeans** from Categories and **Session Bean** as File Type.



   d. Click **Next**.

e. In the New Session Bean dialog box, set the following properties:

- EJB Name: `ProductFacade`
- Package: `demos.model`
- Session type should be set to: `Stateless`



f. Click **Finish**.

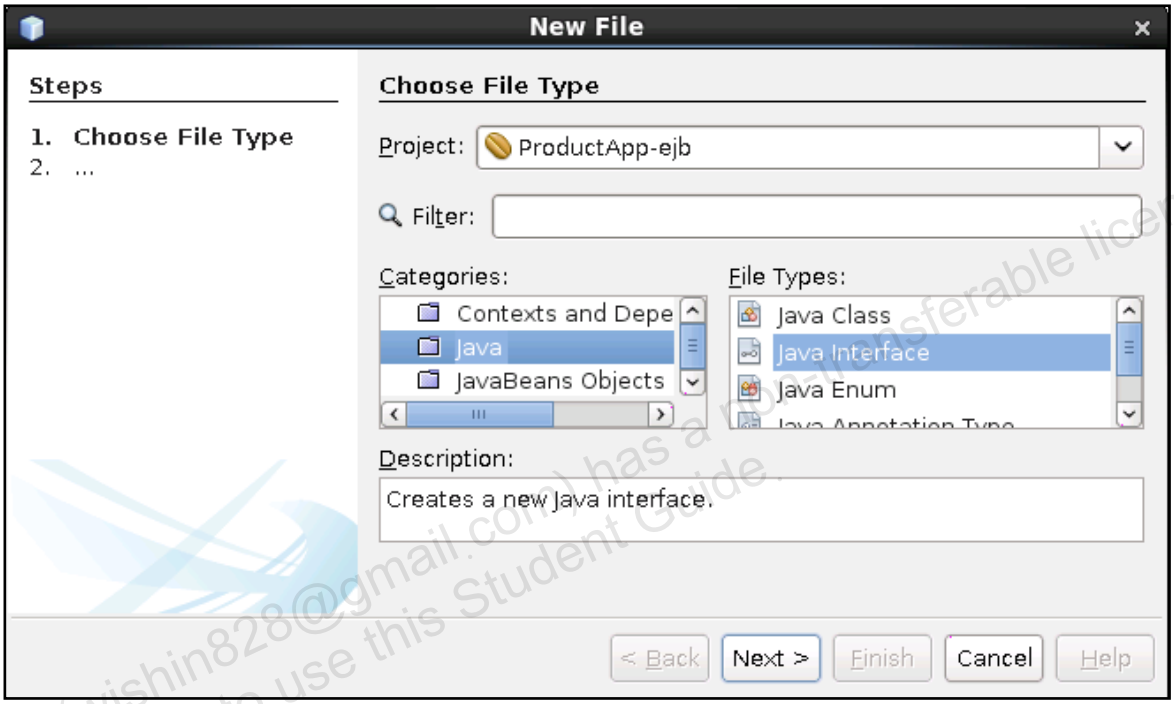
3. Create a Remote interface for the ProductFacade EJB:

   Note: Remote and Local interfaces could be added immediately to the EJB component as you are creating it with a NetBeans wizard. However, for training purposes you are asked to add an interface to the bean after it was already created.

   a. Select the **ProductApp-ejb** project.

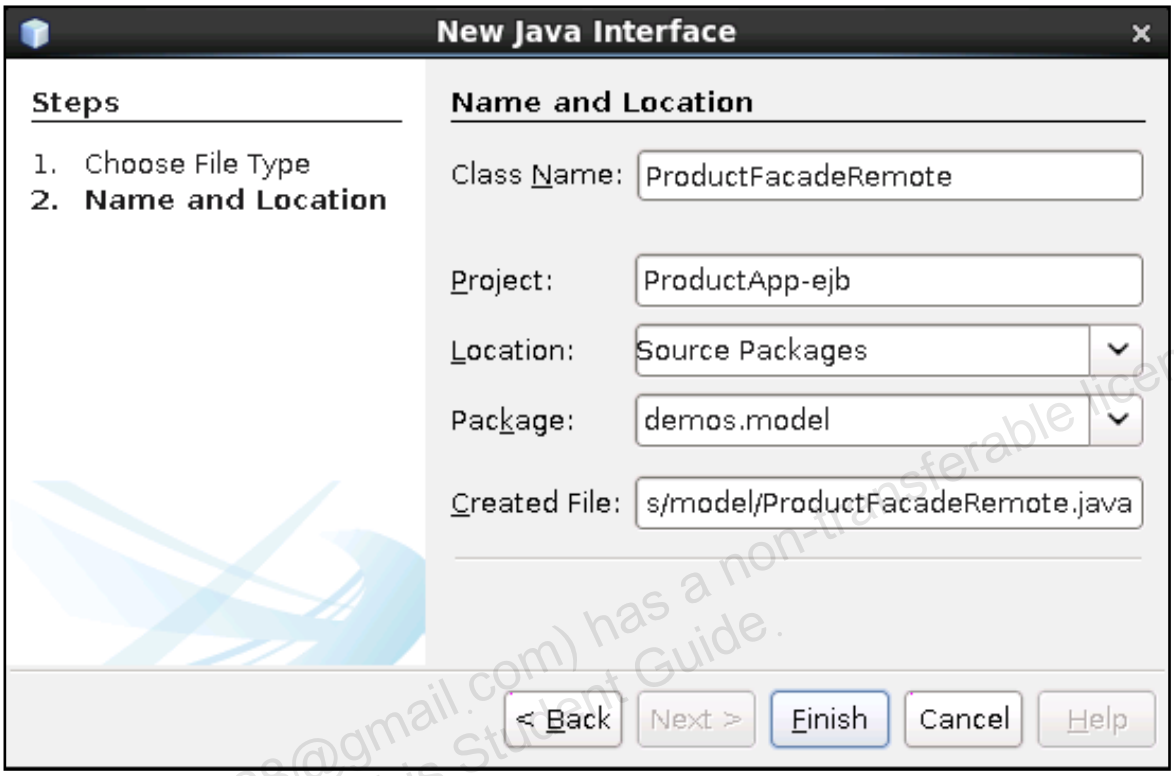
   b. Select **File > New File**.

   c. Select **Java** from the list of Categories and **Java Interface** from the list of File Types.



   d. Click **Next**.

e. In the New Java Interface dialog box, set the following properties:
   - Class Name: `ProductFacadeRemote`
   - Package: `demos.model`



f. Click **Finish**.

4. Add business method definitions to the Remote interface of the ProductFacade EJB:
   a. Open the `ProductFacadeRemote` interface.
   b. Just before the line of code containing the celebration of the ProductFacadeRemote interface, add the following `@Remote` annotation:

```
@Remote
public interface ProductFacadeRemote {
     // business method definitions will be added here
}
```

   c. Add an import: `javax.ejb.Remote`
   d. Add the following business method definitions into the body of ProductFacadeRemote:

```
void create(@Valid Product product);
void update(@Valid Product product);
void delete(Product product);
Product findProduct(Integer id);
List<Product> findProductByName(String name);
```

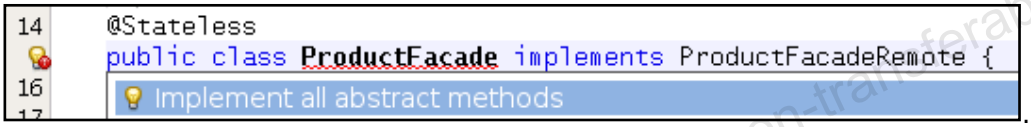
e. Add imports of following classes:

```
demos.db.Product
java.util.List
javax.validation.Valid
```

5. Add business method implementations to the ProductFacade EJB:
   a. Open the `ProductFacade` class.
   b. Add an interface implementation clause to the `ProductFacade` class definition:

```
public class ProductFacade implements ProductFacadeRemote {
    // business method implementations will be added here
}
```

   c. Click the light bulb icon to the left of the class definition line of code and select **implement all abstract methods**.



Note: A default dummy code has been added to all business method definitions. Your next test will be to replace it with actual business logic.

Note: Imports of `demos.db.Product` and `java.util`.List classes were automatically added to the `ProductFacade` class.

6. Add business method implementations to the ProductFacade EJB:
   a. On the first line of code inside the body of the ProductFacade class, add code that injects Persistence Context:

```
@PersistenceContext(unitName="ProductApp-ejbPU")
 private EntityManager em;
```

   b. Add the following imports:

```
javax.persistence.EntityManager
javax.persistence.PersistenceContext
```

   c. Replace the body of the `create` method by removing the existing line of code that throws an exception and adding instead code that persists a new entity object:

```
em.persist(product);
```

   d. Replace the body of the `update` method, just as in the previous step, with the code that merges an entity object:

```
em.merge(product);
```

e. Add validation annotation to the Product parameter in the create and update operations:

```
public void create(@Valid Product product) {
    em.persist(product);
}
public void update(@Valid Product product) {
    em.merge(product);
}
```

f. Add an import: `javax.validation.Valid`

g. Replace the body of the `delete` method, just as in the previous step, with the code that removes an entity object:

```
em.remove(product);
```

h. Replace the body of the `findProduct` method, just as in the previous step, with the code that finds and returns a Product object using its ID:

```
return em.find(Product.class, id);
```

i. Replace the body of the `findProductByName` method, just as in the previous step, with the code that initializes a `findByName` Named Query object, sets the `name` parameter, and returns query results as a list:

```
Query productNameQuery = em.createNamedQuery("Product.findByName");
productNameQuery.setParameter("name", name);
return productNameQuery.getResultList();
```

j. Add an import: `javax.persistence.Query` class.

7. Add local operations to the ProductFacade EJB:

Note: The `ProductFacade` class is already annotated with the `@LocalBean` annotation, so all of its operations can be invoked locally. However, they can also be invoked remotely, because they are described in the ProductFacadeRemote interface. You will add operations to the ProductFacade EJB that are not declared by its remote interface. Therefore, such operations will only be available to local callers.

a. Add a new business method called `findProductByDate` that should return a list of products with a specific best-before date.

```
public List<Product> findProductByDate(LocalDate date) {
    // add implementation code here
}
```

b. Add an import: `java.time.LocalDate`

c. Add implementation logic to this method that initializes a `findByBestBeforeDate` Named Query, sets the date parameter, and returns query results as a list:

```
Query productDateQuery =
em.createNamedQuery("Product.findByBestBefore");
productDateQuery.setParameter("bestBefore", date);
return productDateQuery.getResultList();
```

d. Add a new method called `findTotal` that should accept a list of product ID values and return an array of objects.

```
public Object[] findTotal(List<Integer> ids) {
    // add implementation code here
}
```

e. Add code to the `findTotal` method to initialize a Named Query object called `"Product.findTotal"` (this is a named query you added to Product entity in "Practices for Lesson 3").

```
Query productTotalQuery = em.createNamedQuery("Product.findTotal");
```

f. Query parameter called "ids":

```
productTotalQuery.setParameter("ids",ids);
```

g. Execute this query (it should return only one record) and return items contained within the query result as array of objects.

```
return (Object[])productTotalQuery.getSingleResult();
```

Note: Array object returned from this method would contain a product counter and a sum of product prices.

h. Add another business method to the ProductFacade class that just validates product object. This operation should be used with no transactional context.

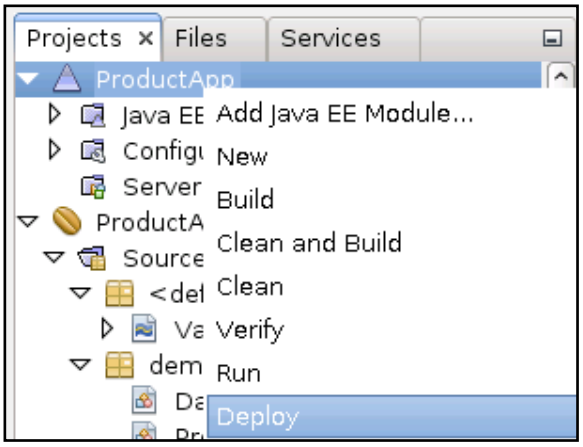
```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public void validateProduct(@Valid Product product) { }
```

Note: This operation is intended to be invoked to validate product, without actually attempting to make any changes to the database. Adding the @Valid annotation ensures that validation logic would be applied to the product; therefore, there is nothing else you would want to do in this operation.

i. Add imports: `javax.ejb.TransactionAttribute` and `javax.ejb.TransactionAttributeType`

j. Compile the **ProductApp-ejb** project using **Clean and Build**.

8. Deploy ProductApp:
   a. Right-click the **ProductApp** project and select **Deploy**.



   b. You should get a BUILD SUCCESSFUL message in the Output window, indicating that your project has been deployed successfully.

Practice 4-2: Creating an EJB Client

Overview

In this practice, you modify the `ProductManager` class to handle interactions with the ProductFacade EJB instead of using local JPA components.

Assumptions

- You have successfully completed all previous practices.
- The `ProductManager` class already contains business method definitions. However, you still need to add implementation code to this class to invoke the Product Facade EJB via its remote interface.
- The `ProductClient` class already contains a `main` method with the same `try` / `catch` block as the one you wrote in "Practices for Lesson 3." However, new implementation logic would have to be added to the `try` block to test remote EJB functionalities.

Tasks

1. Modify the ProductManager class to set up a Logger and declare a reference to represent the ProductFacade Remote EJB:
   a. Select the **ProductApp-client** project.
   b. Expand **Source Packages > demos.model** node.
   c. Open **ProductManager** class.
   d. Add a Logger to ProductManager class:
      ```
      private static final Logger logger =
      Logger.getLogger(ProductManager.class.getName());
      ```
   e. Add an import: `java.util.logging.Logger`
   f. Add an instance variable that will be used to hold a reference to the ProductFacadeRemote object:
      ```
      private ProductFacadeRemote productFacade;
      ```

2. Modify the constructor of the `ProductManager` class to perform remote EJB lookup:
   a. Add code to the constructor of the `ProductManager` class that creates a JDNI Context object and performs a lookup to initialize the ProductFacade ejb reference:

```
public ProductManager() {
  try {
    Context ctx = new InitialContext();
    productFacade = (ProductFacadeRemote)ctx.lookup(
                 "java:global/ProductApp/ProductApp-ejb/"+
          "ProductFacade!demos.model.ProductFacadeRemote");
   } catch (Exception ex) {
   logger.log(Level.SEVERE, "Error initialising EJB reference",
ex);
  }}
```

   Note: The JNDI name is concatenated out of two string objects because it did not fit on a single line in the manual. Instead of concatenating strings, you can simply write it as one line of text.

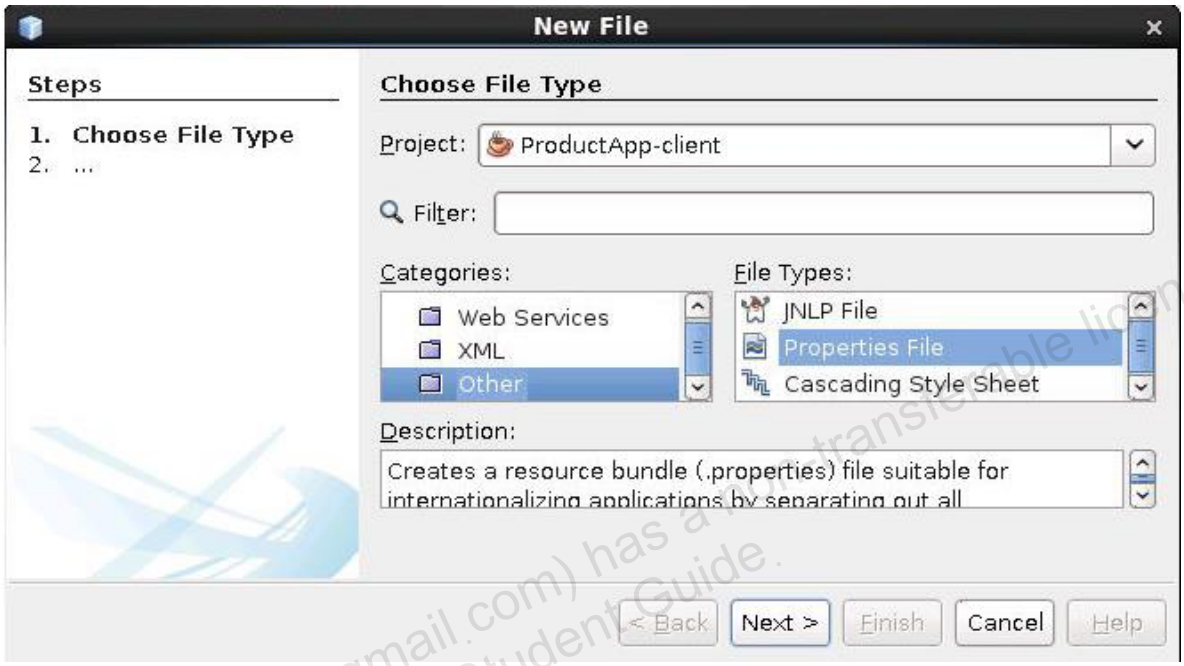
   b. Add imports of the following classes:

```
java.util.logging.Level
javax.naming.Context
javax.naming.InitialContext
```

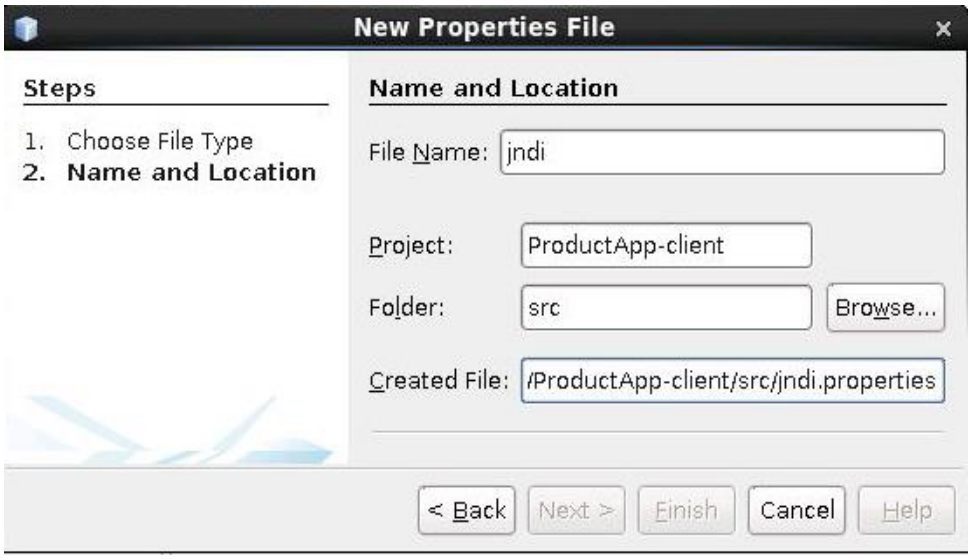
3. Modify business operations of the `ProductManager` class:
   a. Add code to the operations to propagate local calls to the remote EJB for all business methods of the `ProductManager`:

```
public void create(Product product) {
    productFacade.create(product);
}
public void update(Product product) {
    productFacade.update(product);
}
public void delete(Product product) {
    productFacade.delete(product);
}
 public Product findProduct(Integer id) {
    return productFacade.findProduct(id);
}
public List<Product> findProductByName(String name) {
    return productFacade.findProductByName(name);
}
```

4. Add the `jndi.properties` file to the ProductApp-client project to define connection properties, to allow successful JDNI server object lookup:
   a. Select **ProductApp-client** project.
   b. Select **File > New File**.
   c. Select **Other** from a list of Categories and **Properties File** from the list of File Types.



   d. Click **Next**
   e. In the New Properties File dialog box, set the following properties:
      - File Name: `jndi`
      - Folder: `src`



   f. Click **Finish**.

g. Add the following properties to the `jndi.properties` file:

Note: You can copy these values from the `jndi.properties` file located in the `/home/oracle/labs/resources` folder.

```
java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
java.naming.provider.url=t3://localhost:7001
java.naming.security.principal=weblogic
java.naming.security.credentials=welcome1
```

5. Modify the `ProductClient` class to test remote EJB functionalities:
   a. Expand the **demos.client** package folder in the **ProductApp-client** project.
   b. Open the **ProductClient** class.
   c. Inside the `try` block, initialize a reference to the ProductManager object:

```
ProductManager pm = new ProductManager();
```

Note: The `ProductManager` class operates on a stateless session bean, so no explicit closure code is required.

6. Compile the **ProductApp-client** project using **Clean and Build**.

Practice 4-3: Testing the EJB Client

Overview

In this practice, you modify the `ProductClient` class to test remote EJB functionalities.

Assumptions

You have successfully completed all previous practices.

Tasks

1. Add code to the `ProductClient` class to find and print the product with ID 1:
   a. After the line of code that initializes the ProductManager object in the `try` block within the `main` method of the `ProductClient` Class, add code that calls the `findProduct` operation to find the product with ID 1 and then print this product to the console.

      ```
      Product p = pm.findProduct(1);
      System.out.println(p);
      ```

      Note: You can simply uncomment relevant part of the code.

   b. Run the Project. You should see information about product Cookie printed onto the console.

2. Replace code that printed a single product with code that prints all products whose name starts with "Co":
   a. Comment out code used in the previous step.
   b. Add code that invokes the `findProductsByName` method of the `ProductManager` class and process the list of results.

      ```
      List<Product> products = pm.findProductByName("Co%");
      products.stream().forEach(p -> System.out.println(p));
      ```

      Note: You can simply uncomment the relevant part of the code.

   c. Run the Project. You should see information about products Coffee and Cookie printed onto the console.

3. Replace code that queried all products whose name starts with "Co", with the code that finds, changes, validates, and updates first product.
   a. Comment out code created in the previous step.
   b. Use the `findProduct` method to find product with ID 1. Set this product price to be 2.5 and its best-before-date to be tomorrow.

```
Product p = pm.findProduct(1);
p.setPrice(BigDecimal.valueOf(2.5));
p.setBestBefore(LocalDate.now().plusDays(1));
pm.update(p);
```

      Note: You can simply uncomment relevant part of the code.

      Note: You can open **SQL Console** to view data in the `PRODUCTS` table in order to observe how this test updates records. Instructions are available in Practice 3-3 step 8.

   c. Run this project again.

4. Change values so that the product update will trigger validation errors.
   a. Set this product price to be 0.1 and product name to be "x". Leave the rest of the code the same.

```
Product p = pm.findProduct(1);
p.setPrice(BigDecimal.valueOf(0.1));
p.setName("x");
p.setBestBefore(LocalDate.now().plusDays(1));
pm.update(p);
```

   b. Run this project again.
   c. Observe messages logged on to the Java Console:

```
INFO: Price must be greater than 1
INFO: Name cannot be shorter than 3 or longer than 40
```

   d. You may also rerun SQL query via the SQL console to see that the record has not been changed.

      Note: Optionally (if you have time) you may repeat the test that prevented record update due to the Optimistic Lock exception. Detailed instructions are available in Practice-3-3 step 11.

Practice 4-4: Creating an EJB Timer

Overview

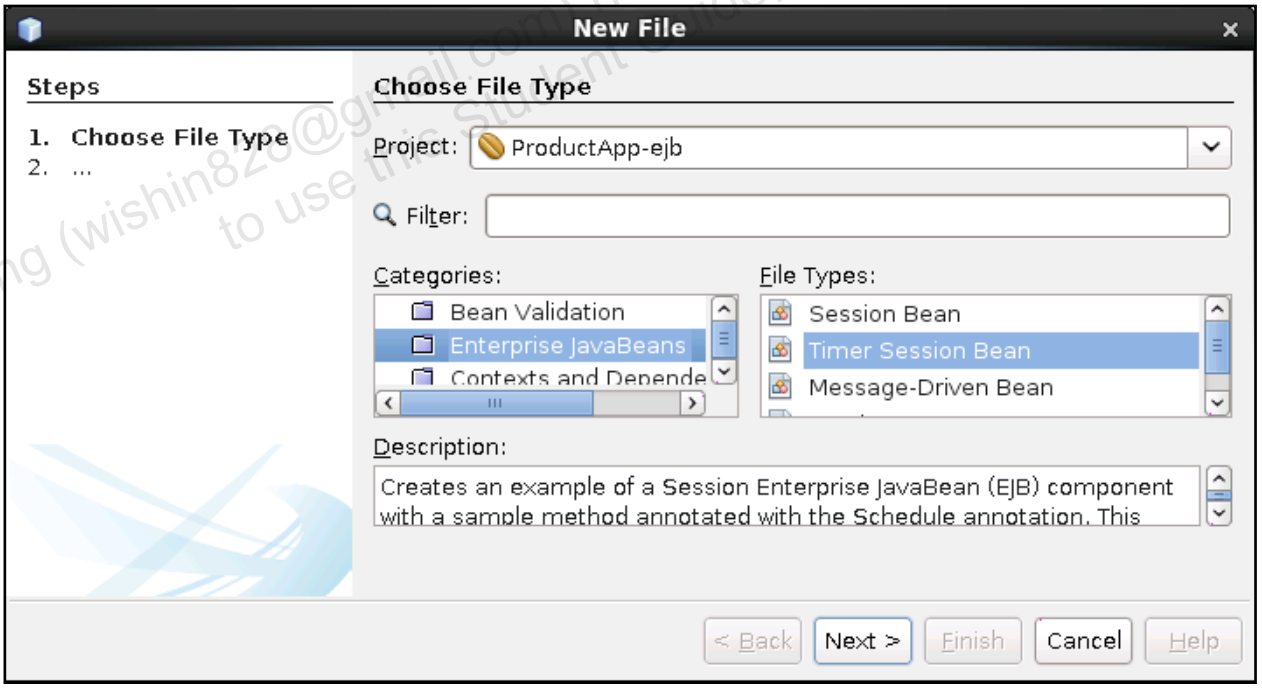
In this practice, you modify ProductFacade EJB to define operations that can only be invoked via local bean calls and not exposed to its remote interface. You also create a Singleton EJB component with automatic timer. Business logic of this timer component would periodically check if there are products that are soon to expire.

Assumptions

You have successfully completed all previous practices.

Tasks

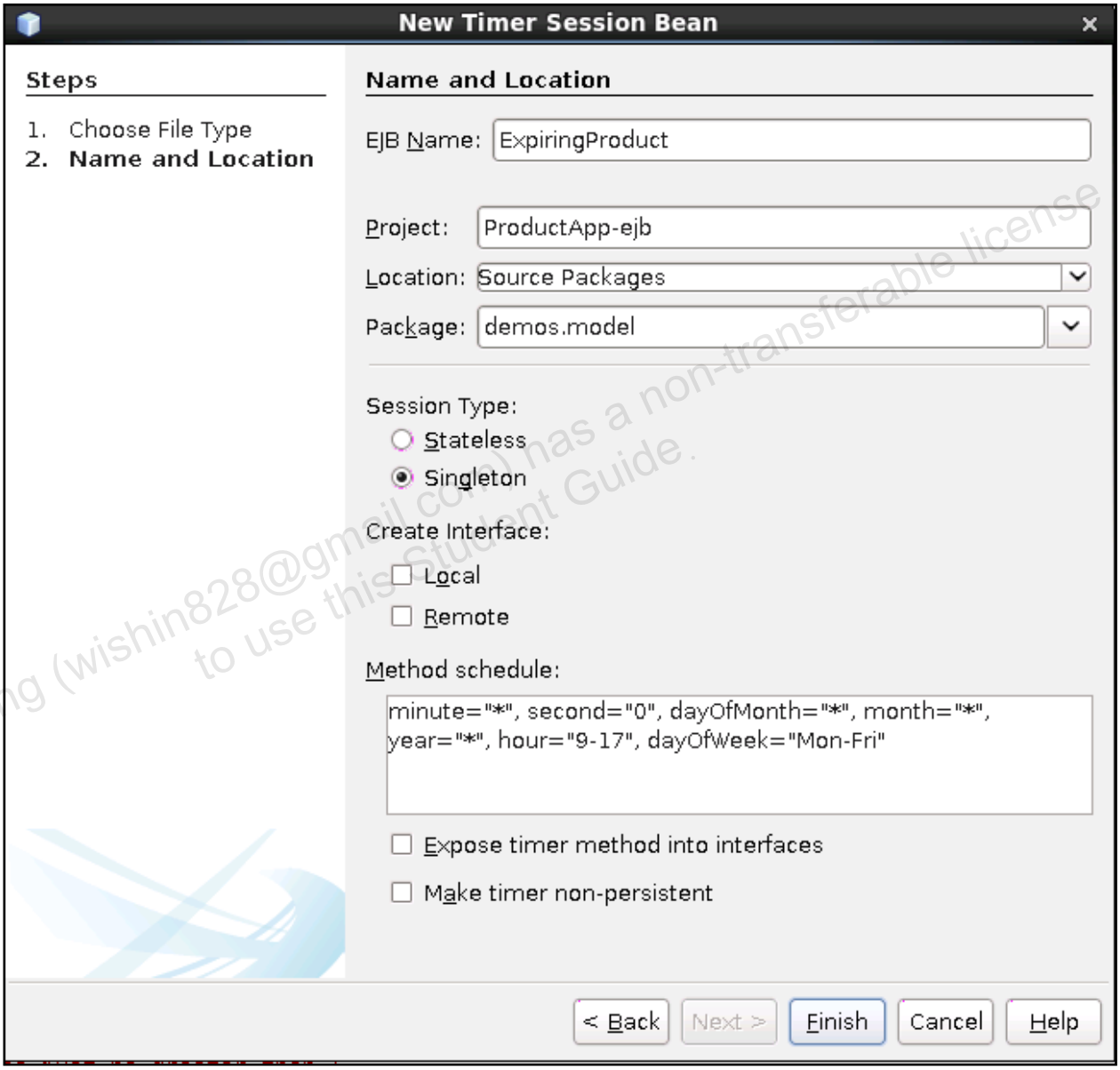
1. Create a new Singleton Timer Session EJB to periodically check products that are due to expire:
   a. Select the **ProductApp-ejb** project.
   b. Select **File > New File**.
   c. Select **Enterprise JavaBeans** from Categories and **Timer Session Bean** from File Types.



   d. Click **Next**

e. In the New Timer Session Bean dialog box, set the following properties:

- EJB Name: `ExpiringProduct`
- Package: `demos.model`
- Session Type: Select the **Singleton** option.
- Deselect the **Expose timer method into interfaces** check box.



f. Click **Finish**.

2. Modify the ExpiringProduct timer bean to handle products that are due to expire:
   a. Open the `ExpiringProduct` class.
   b. On a next line of code after the class declaration add initialization of a Logger object.
      ```
      private static final Logger logger =
      Logger.getLogger(ExpiringProduct.class.getName());
      ```
   c. Add an import: `java.util.logging.Logger`
   d. After the logger initialization, inject a ProductFacade EJB reference.
      ```
      @EJB
      private ProductFacade productFacade;
      ```
   e. Add an import: `javax.ejb.EJB`
   f. Rename `myTimer` method to: `handleExpiringProducts`
   g. Remove existing lines of code from the `handleExpiringProducts` operation.
   h. Instead add logic to the `handleExpiringProducts` operation to invoke the `findProductByDate` method of the ProductFacade bean to get a list of products whose best before date expires tomorrow.
      ```
      List<Product> products =
      productFacade.findProductByDate(LocalDate.now().plusDays(1));
      ```
   i. Process each fetched product by writing information about it to the log.
      ```
      products.stream().forEach(p -> logger.log(Level.INFO,
      p.toString()));
      ```
      Note: More realistic business logic will be added to this operation in a later practice.

3. Test the ExpiringProduct timer bean:
   a. Take a note of the current time on the computer on which you run your practices.
   b. Modify `Schedule` annotation against the `handleExpiringProducts` operation. For example, if your computer current time is 16:35 , set Schedule annotation properties to:

```
@Schedule(dayOfWeek = "*", month = "*", hour = "16", dayOfMonth = "*",
year = "*", minute = "37", second = "0")
```

      Note: This example schedule is going to trigger method invocation every day at 16:37. You may adjust values as required.

   c. Right-click the **ProductApp** project and select **Deploy**.
   d. Wait for deployment to complete successfully.
   e. Click the **Oracle WebLogic Server** tab in the **Output** window in NetBeans. Wait for the timer you have created to expire and observe messages logged by the ExpiringProduct bean. You should be able to see that a product called Cookie is due to expire. This was the product you have updated in the Practice 4-3 step 3.b to expire tomorrow. You can verify this by either querying this product using the ProductApp-client application or by using SQL Console.

Practices for Lesson 5: Using Java Message Service API

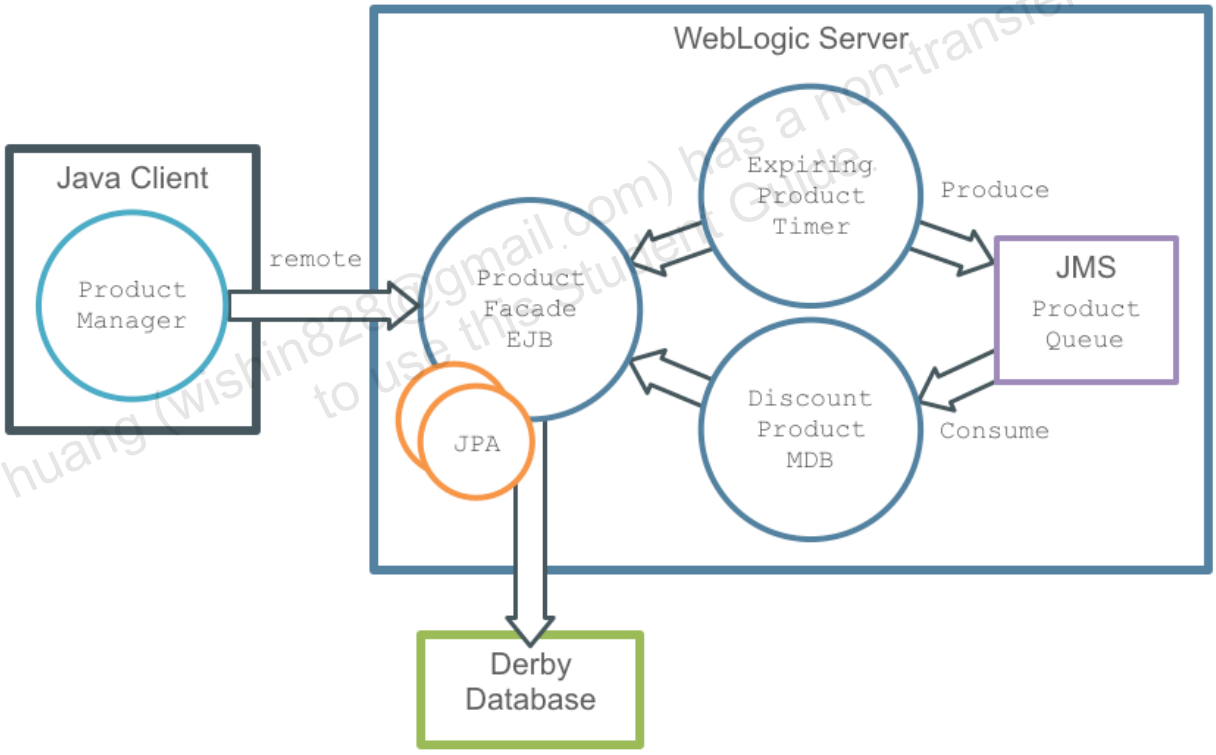
Practices for Lesson 5: Overview

Overview

In these practices, you create a WebLogic JMS Server configuration. You modify the ExpiringProduct Timer EJB to act as a message producer to publish messages into a queue. You create a new Message Driven Bean to act as a message consumer and handle messages arriving in the queue.

In these practices, you enable your Java EE application to send and receive JMS messages.

- You configure a WebLogic Server JMS server and JMS resources to be managed by it.
- You create an `ExpiringProduct Singleton` Timer EJB as a message producer.
- You create a `DiscountProduct` Message-Driven Bean as a message consumer.



Practice 5-1: Configuring WebLogic JMS Server

Overview

In this practice, you configure WebLogic JMS server and JMS resources to be managed by it.

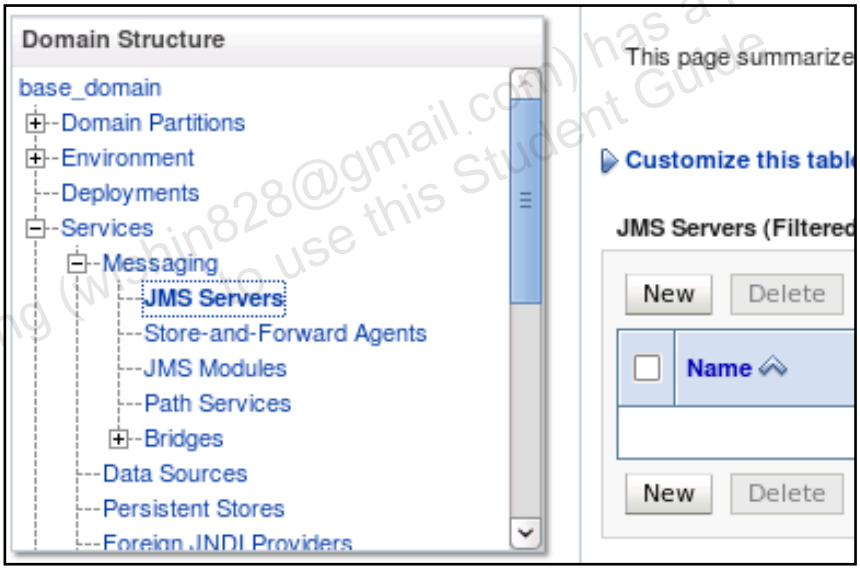
Assumptions

You have successfully completed all previous practices.

Tasks

1. Log in to WebLogic Server Console:
   a. Open a browser and navigate to: `http://localhost:7001/console`
   b. Log in using: username `weblogic` and password `welcome1`

2. Configure a JMS server:
   a. Expand **Services > Messaging** nodes in the Domain Structure panel.
   b. Click the **JMS Servers** node.

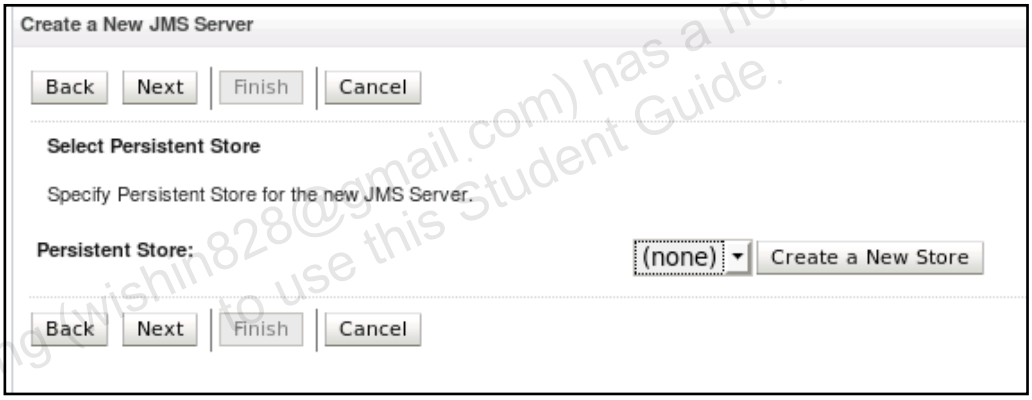


   c. Click the **New** button to create the new JMS server.

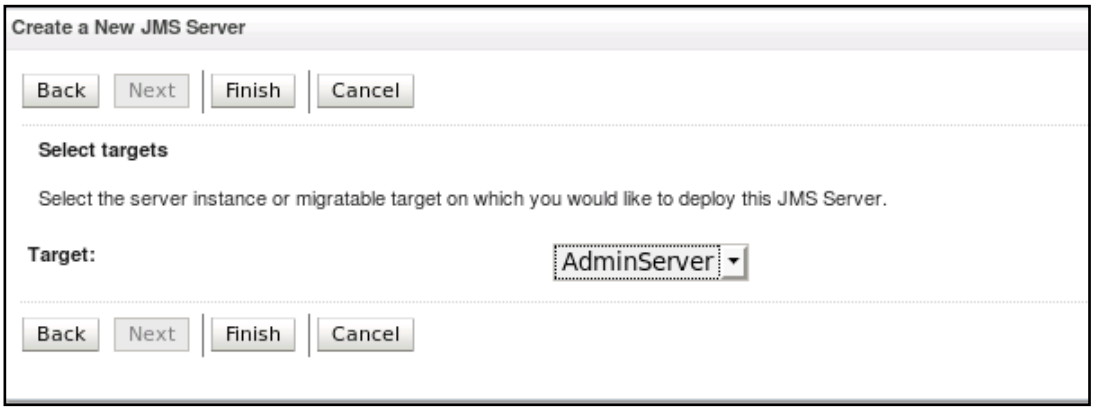
d. Set the Name property to: `JMSServer`

e. Click **Next**.

f. Do not create a persistent store for this JMS server. Just click **Next**.

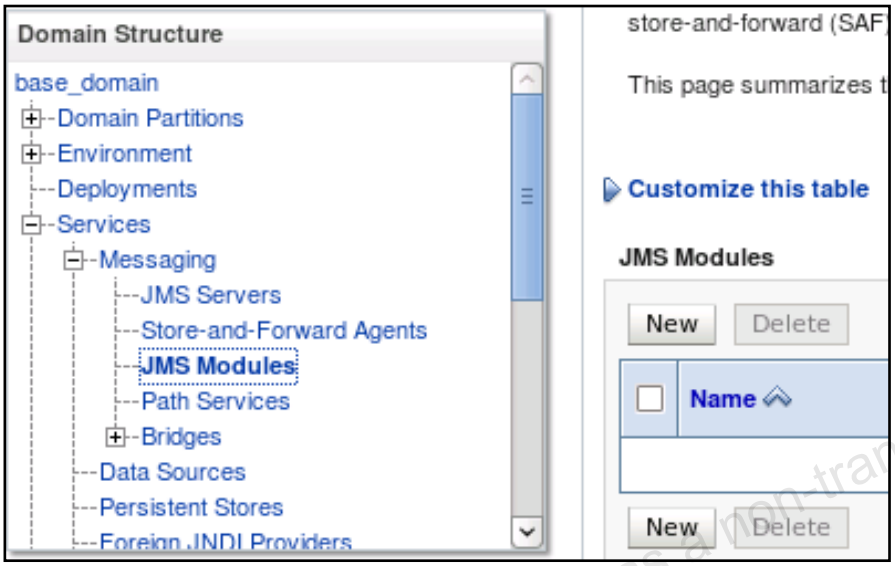


g. Select **AdminServer** as the target for the JMS server.

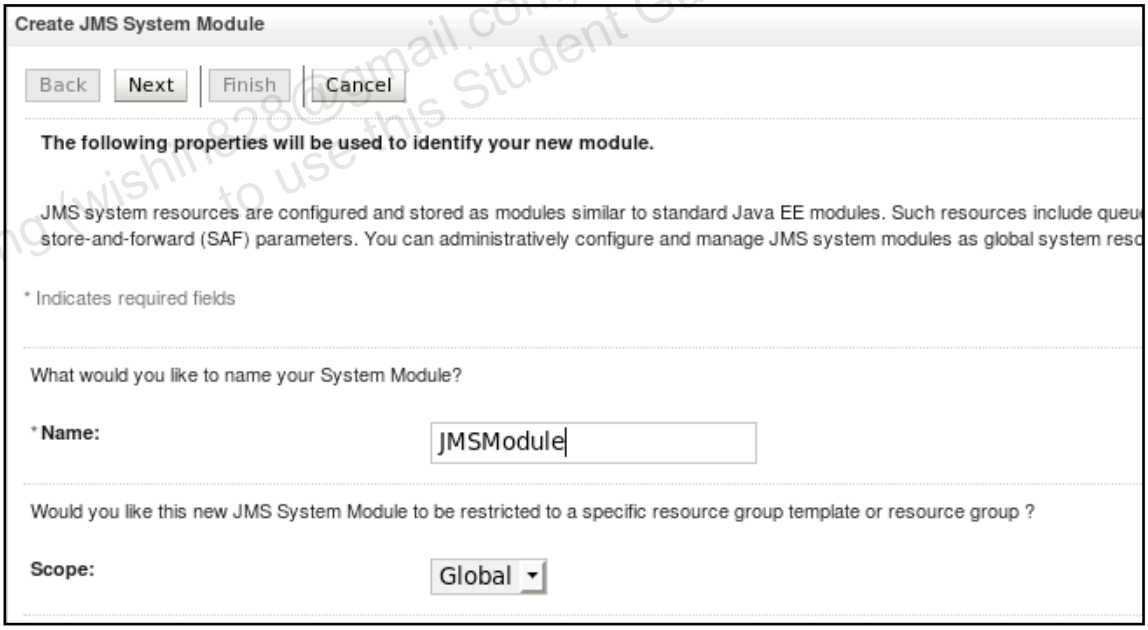


h. Click **Finish**.

3. Configure `JMSModule`:
   a. Expand **Services > Messaging** nodes in the Domain Structure panel.
   b. Click the **JMS Modules** node.
   c. Click the **New** button to create new a JMS module.

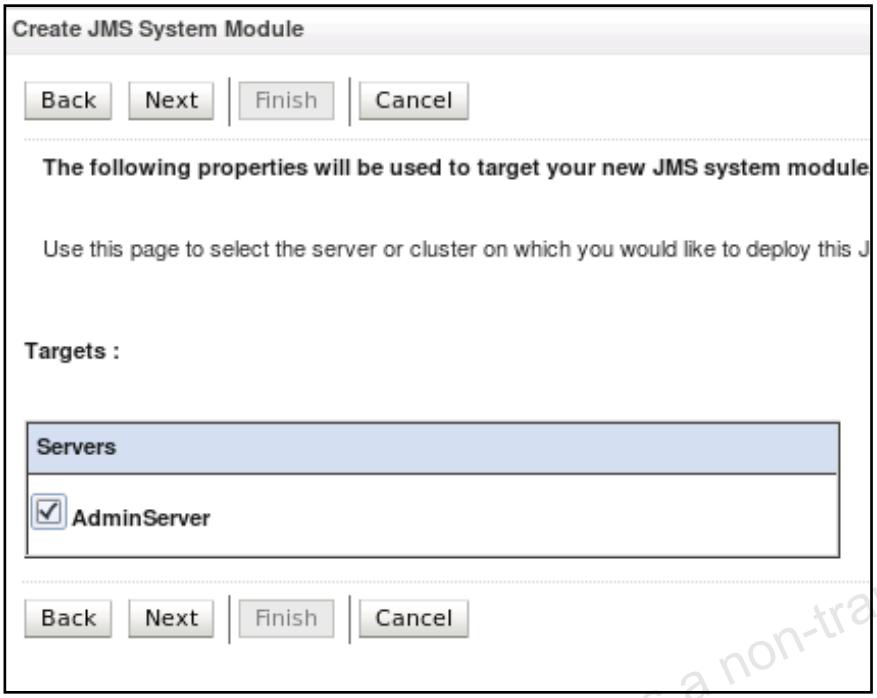


   d. Set the Name property to: `JMSModule`



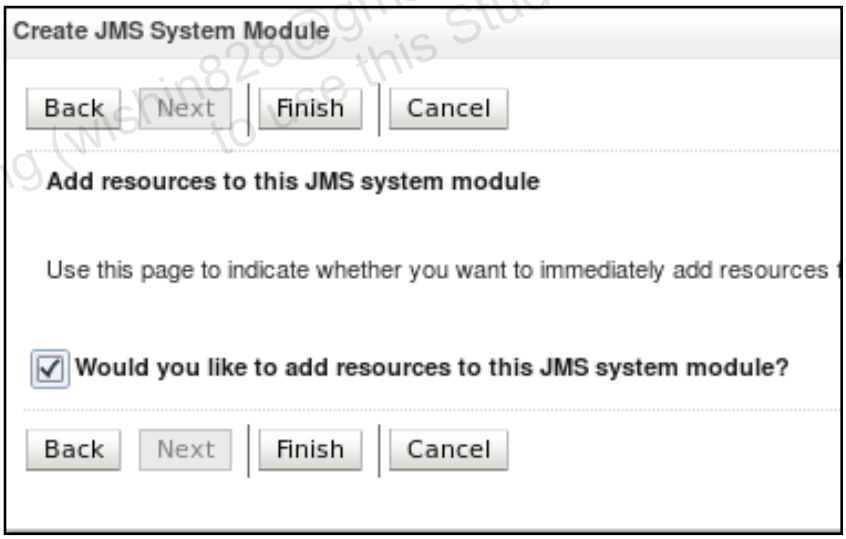
   e. Click **Next**.

f. Select **AdminServer** as the target for `JMSModule`.



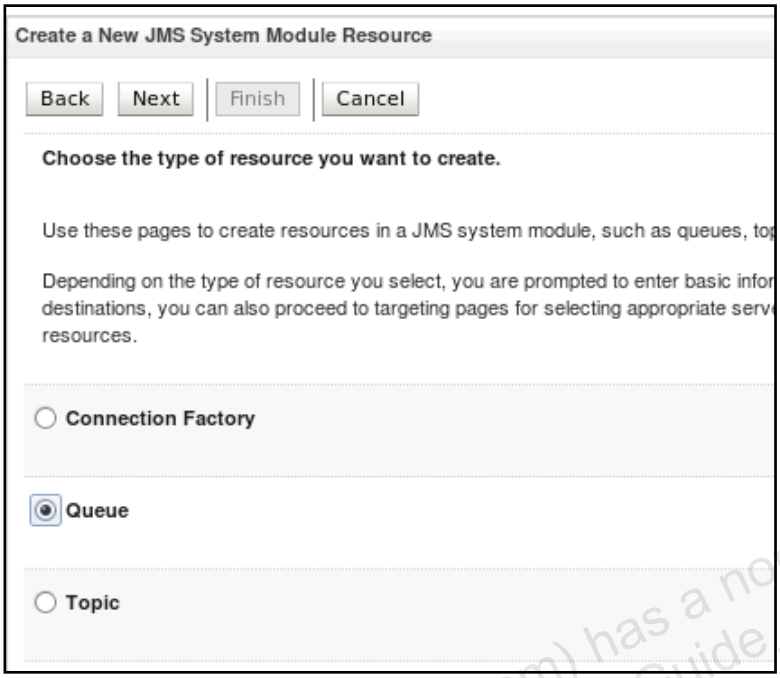
g. Click **Next**.

h. Select the **Would you like to add resources to this JMS system module?** check box.

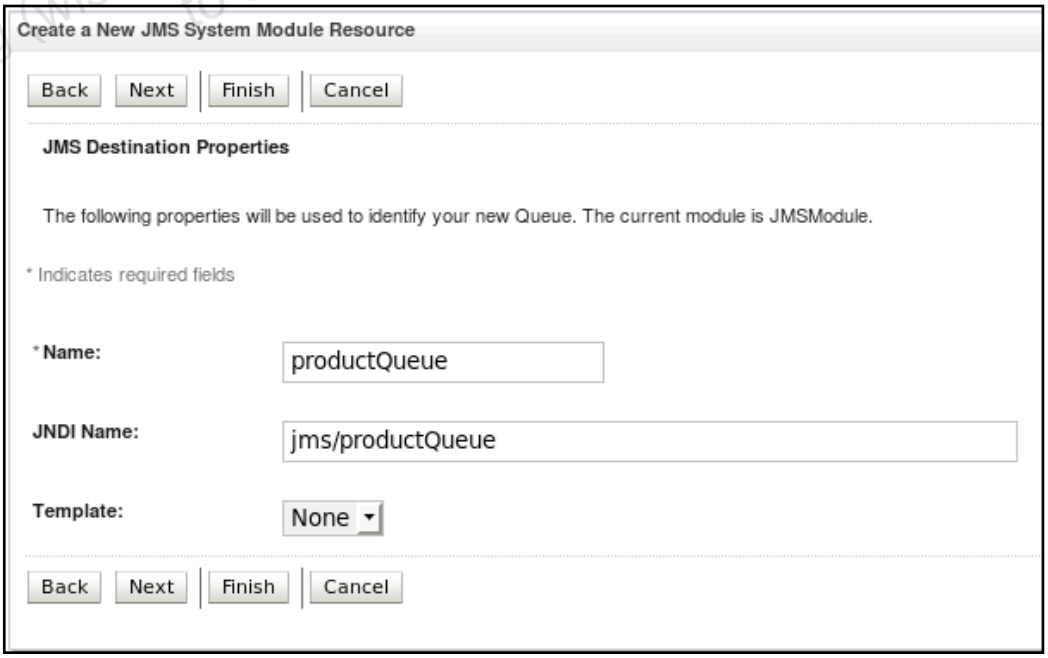


i. Click **Finish.**

4. Configure a JMS queue:
   a. Click **New** to add a new resource to the system module.
   b. Create a new Queue destination by selecting **Queue** from the list of option buttons.

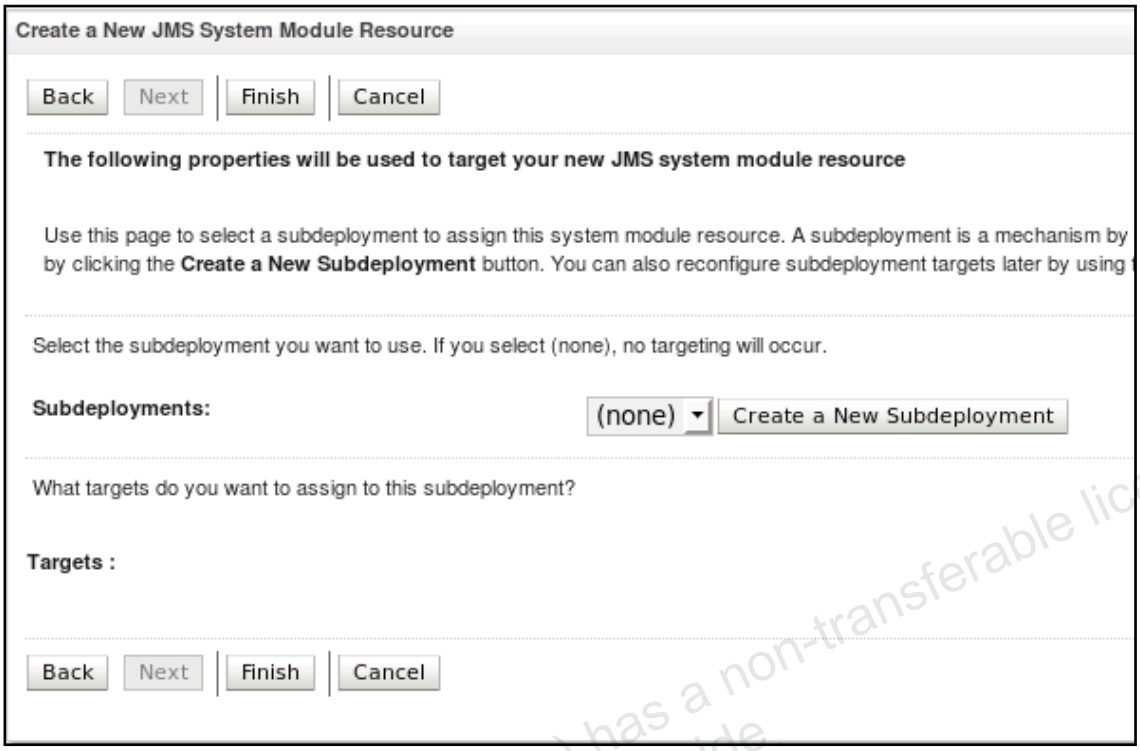


   c. Click **Next.**
   d. Set Queue properties.
      - Name: `productQueue`
      - JNDI Name: `jms/productQueue`

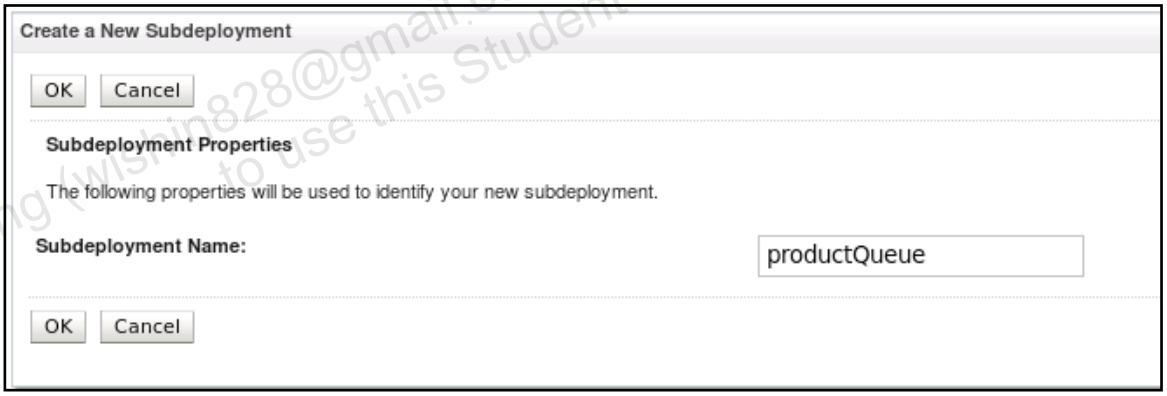


   e. Click **Next.**

f. Click the Create a New Subdeployment button.

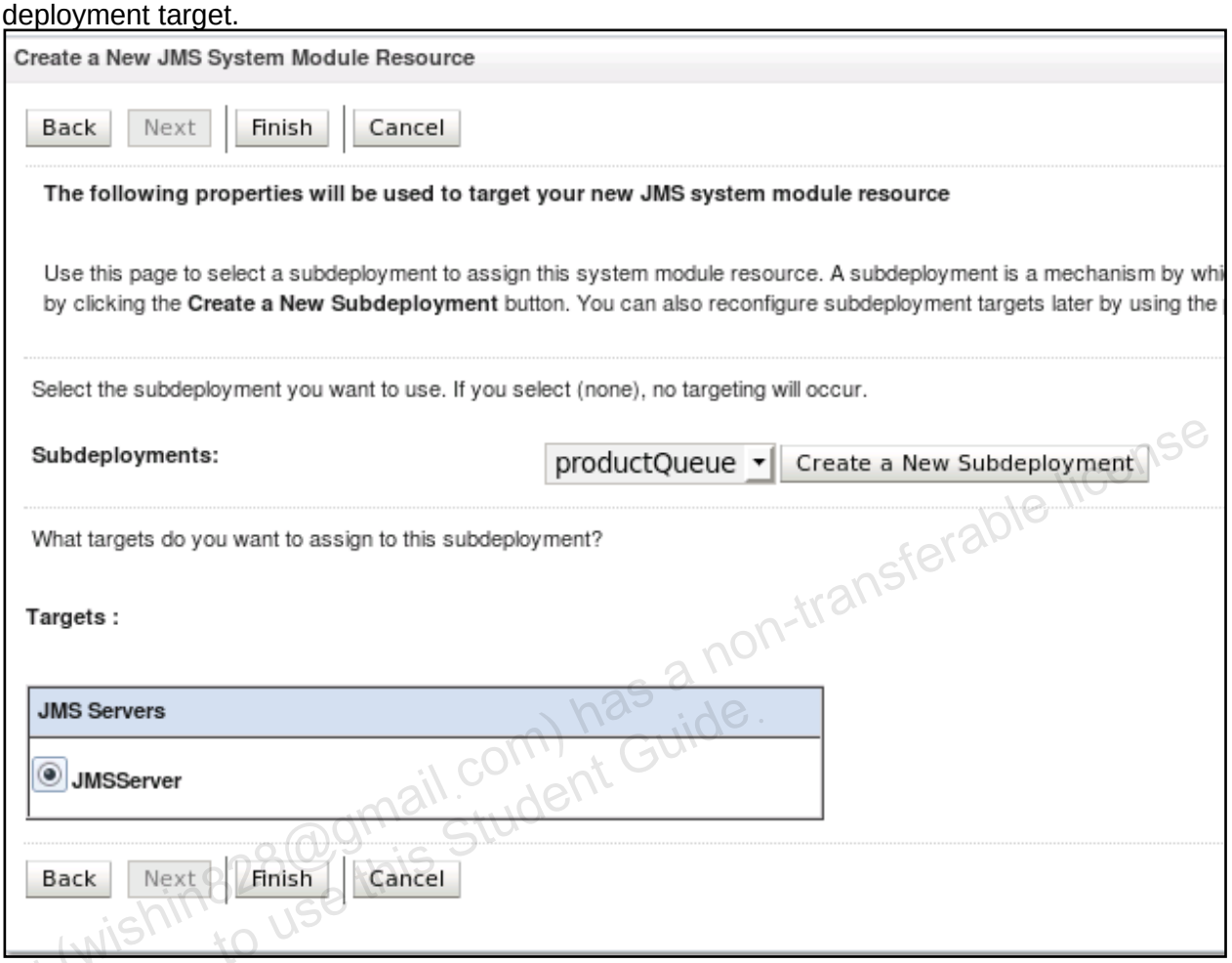


g. Set Subdeployment Name to: `productQueue`



h. Click **OK**.

i. With the **productQueue** subdeployment now selected, select the **JMSServer** as the deployment target.



j. Click **Finish**.

Note: You have just created a JMS server, a JMS module, a JMS Queue, and a subdeployment.

- A JMS Server is s a WebLogic-specific configuration that describes settings for the JMS services within the WebLogic environment.
- A JMS module is a WebLogic-specific configuration that describes a group of JMS resources, such as queues, topics, connection factories, and so on.
- A JMS queue defines a standard Java EE queue. This definition is stored as a part of the JMS module within the WebLogic configuration.
- A subdeployment is a WebLogic specific mechanism, by which JMS module resources (such as queues, topics, connection factories) are grouped and targeted to specific server resources (such as JMS servers or clusters).

Practice 5-2: Creating a JMS Producer and a JMS Consumer

Overview

In this practice, you modify code of the ExpiringProduct timer bean to place messages into a queue for every product that is soon to expire. Then you create a new Message Driven Bean that will be getting messages out of this queue and applying price discount for the products that are due to expire.

Assumptions

You have successfully completed all previous practices.

Tasks

1. Modify ExpiringProduct Timer EJB to place messages into a queue:
   a. Using NetBeans open the **ExpiringProduct** class located in the **ProductApp-ejb** project.
   b. After the line of code that declared and injected a ProductFacade variable, add an injection of JMSContext and jms/productQueue objects.

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

   c. Add imports of the following classes:

```
javax.jms.JMSContext
javax.jms.Queue
javax.inject.Inject
javax.annotation.Resource
```

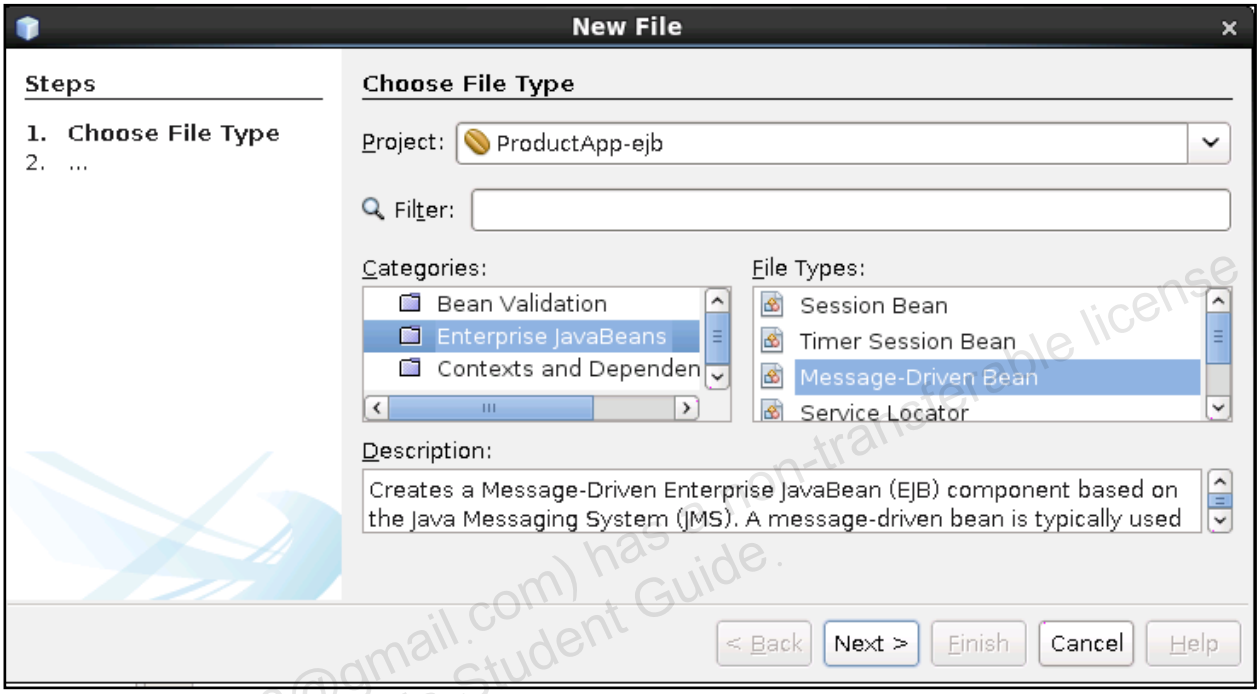
   d. Add a new line of code at the beginning of the `handleExpiringProducts` method that creates a new JMSProducer object using JMSContext object that you have injected in the previous step of this practice.

```
JMSProducer producer = context.createProducer();
```

   e. Add an import: `javax.jms.JMSProducer`
   f. Modify last line of code in the `handleExpiringProducts` method. Instead of just writing product objects to log, you need to create a new object message for each product object and send it to the queue.

```
products.stream().forEach(p -> producer.send(productQueue,
                                 context.createObjectMessage(p)));
```

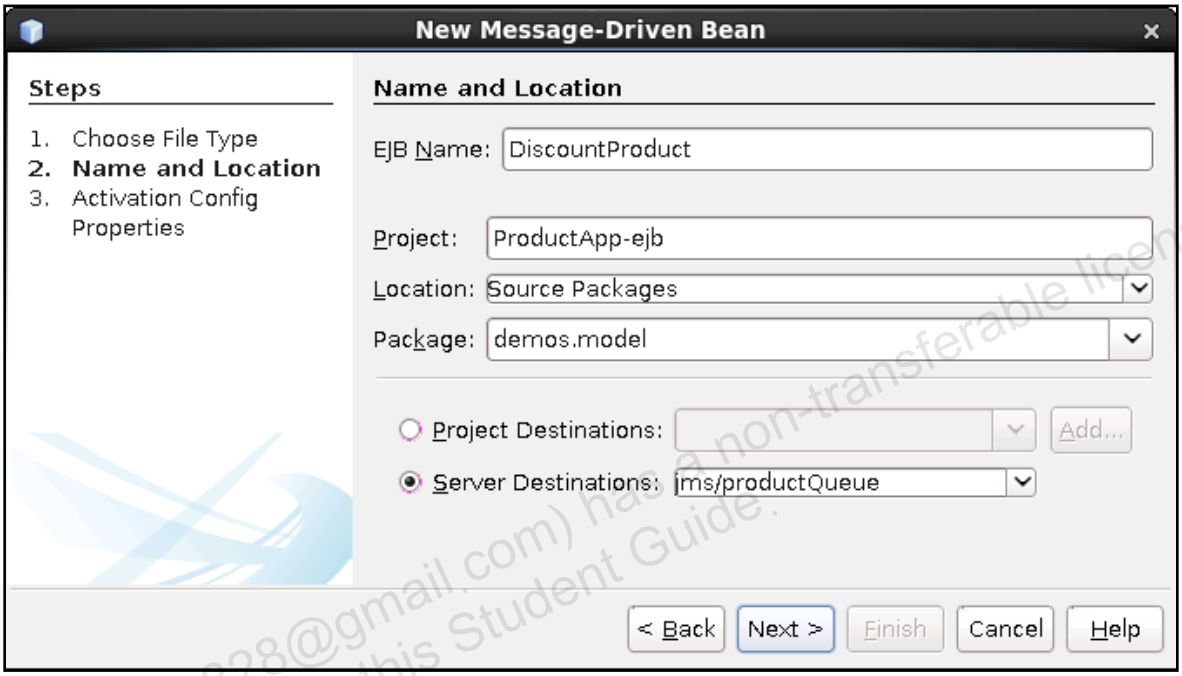
2. Create a new Message-Driven Bean that will fetch messages from the queue:
   a. Select the **ProductApp-ejb** project.
   b. Select **File > New File**.
   c. Select **Enterprise JavaBeans** from the list of Categories and **Message-Driven Bean** from the list of File Types.



   d. Click **Next.**

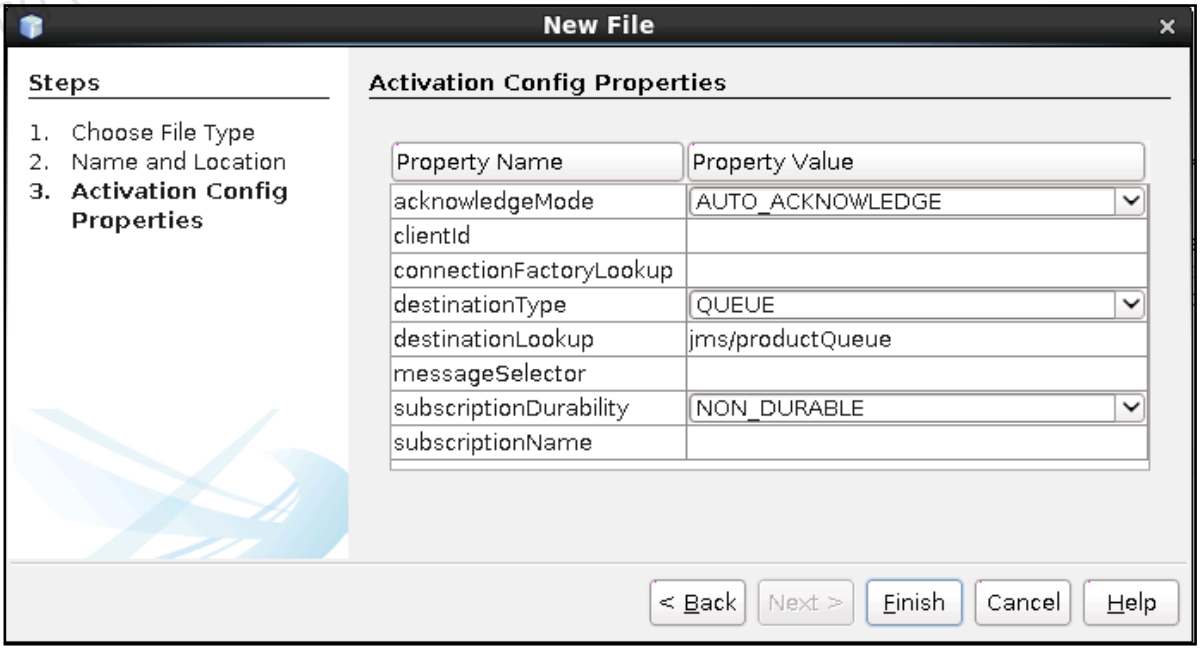
e. Set Message-Driven Bean properties:

- EJB Name: `DiscountProduct`
- Package Name: `demos.model`
- Click the Server Destinations option button.
- Select `jms/productQueue` from the drop-down list.



f. Click **Next**.

g. Accept all default values for the Activation Config properties.



h. Click **Finish**.

3. Modify Message-Driven bean activation configuration to limit the amount of retry attempts when handling errors messages:
   a. Open the **DiscountProduct** class.
   b. Find the **MessageDriven** annotation that contains two **ActivationConfigProperty** annotations in inside.
   c. Add a third **ActivationConfigProperty** annotation to set maximum delivery retry count of 3 as a message selector property.

```
@ActivationConfigProperty(propertyName="messageSelector",
propertyValue="JMSXDeliveryCount < 3")
```

4. Modify declarations and initializations of static and instance fields in the DiscountProduct class:
   a. Open DiscountProduct class and add Logger to it.

```
private static final Logger logger =
Logger.getLogger(DiscountProduct.class.getName());
```

   b. Add an import: `java.util.logging.Logger`
   c. Inject a reference to the `ProductFacade` bean

```
@EJB
private ProductFacade productFacade;
```

   d. Add an import: `javax.ejb.EJB`

5. Add exception handling code to the `onMessage` method of the `DiscountProduct` Message-Driven bean:
   a. You need to create a handler for the JMSException and also handlers for exceptions that relate to ProductFacade JPA functionalities. The second `catch` block that handles the `Exception` object can be copied from the ProductClient class in the ProductApp-client project.

```
try {
   // more code will be added here
} catch (JMSException ex) {
   logger.log(Level.SEVERE, "Error Acquiring Message", ex);
} catch (Exception ex) {
   Throwable cause = ex.getCause();
   if (cause instanceof ConstraintViolationException) {
     ConstraintViolationException e = (ConstraintViolationException)cause;
     e.getConstraintViolations().stream()
         .forEach(v -> logger.log(Level.INFO,v.getMessage()));
   } else if (cause instanceof OptimisticLockException) {
     OptimisticLockException e =(OptimisticLockException)cause;
     logger.log(Level.INFO, e.getMessage());
    } else {
         logger.log(Level.SEVERE, "Product Manager Error",ex);
    }
}
```

   Note: The comment line in the example indicates the place where the rest of the logic of the `onMessage` operation should be placed in this practice.

   b. Add imports of following classes:

```
java.util.logging.Level
javax.persistence.OptimisticLockException
javax.validation.ConstraintViolationException
javax.jms.JMSException
```

6. Add logic to handle received messages containing Product objects, calculate and apply a discount, and validate and update these products:
   a. Inside the `try` block add a new line of code that retrieves Product object from the Message that the method had received as parameter.

```
Product product = message.getBody(Product.class);
```

   b. Add an import: `demos.db.Product` class
   c. Calculate new discount value by applying a 10% discount to the price of the product.

```
BigDecimal discount =
product.getPrice().multiply(BigDecimal.valueOf(0.1));
```

   d. Apply discounted price to the product.

```
product.setPrice(product.getPrice().subtract(discount));
```

e. Validate product.

```
productFacade.validateProduct(product);
```

f. Update product.

```
productFacade.update(product);
```

g. Write log message indicating that discount was applied successfully.

```
logger.log(Level.INFO, "Product "+product+" discounted by "+discount);
```

Note: In case product validation fails, `try` block would be interrupted and your code will not attempt to update the product. Exception-handling logic will be triggered and you will be able to see the constraint violation error messages that it would write to the log.

7. Compile the **ProductApp-ejb** project using **Clean and Rebuild**.

Practice 5-3: Testing the JMS Producer and the JMS Consumer

Overview

In this practice, you test the JMS producer and the JMS consumer by modifying two product records to set the expiry date to be tomorrow. Set the price for one of the records to be more then 1 and for the other to be exactly 1, so that both discount calculation outcomes can be tested.

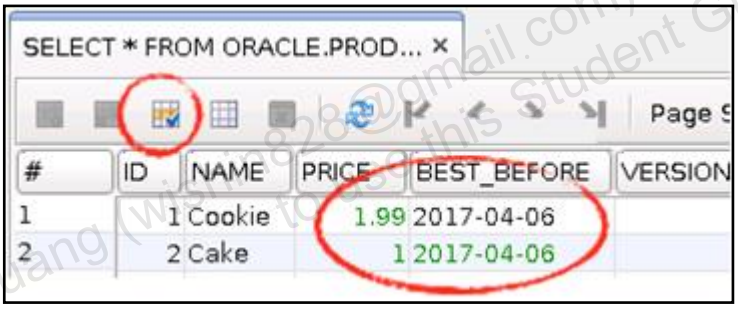
Assumptions

You have successfully completed all previous practices.

Tasks

1. Open SQL Console and update the first two records in the `PRODUCTS` table so that their `best_before` date will be set to tomorrow and set one of these records to have a price value of 1.99 and another to be 1. Commit your changes.

   Note: Check the date on the computer on which you perform your practices and add one day to it. Date format presented by the SQL Console is yyyy-MM-dd, so in the example below the date on the practice computer is 5th of April 2017 and you set the next day to be 2017-04-06.



   Note: Alternatively use the ProductApp-client application to find and update records with IDs 1 and 2 with same values.

2. Modify the `ExpiringProduct` Timer EJB to adjust the schedule for testing:
   a. Take a note of the current time on the computer on which you run your practices.
   b. Modify the `Schedule` annotation against the `handleExpiringProducts` operation. For example, if your computer current time is 16:35 , set the `Schedule` annotation properties to:

```
@Schedule(dayOfWeek = "*", month = "*", hour = "16", dayOfMonth = "*",
year = "*", minute = "37", second = "0")
```

      Note: This example schedule is going to trigger method invocation every day at 16:37. You should adjust values as required.

   c. Compile the ProductApp-ejb project using **Clean and Rebuild**.
   d. Right-click the **ProductApp** project and select **Deploy**.

e. After successful deployment of the enterprise application click the **Oracle WebLogic Server** tab in the **Output** window in NetBeans and observe messages that would be logged by the `DiscountProduct` bean. You can also observe changes to the discount of the first product value via SQL console.

f. Right-click the **ProductApp** project and invoke **Deploy** menu

g. Wait for deployment to complete successfully.

h. Click the **Oracle WebLogic Server** tab in the **Output** window in NetBeans.

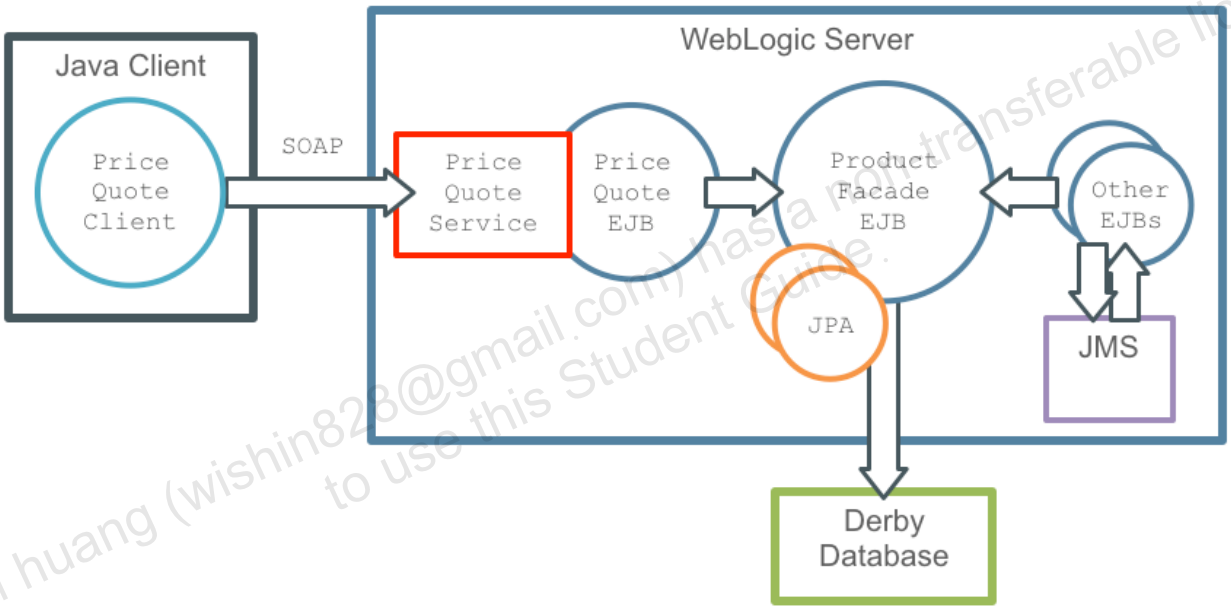
- Wait for the timer you have created to expire.
- After the timer expires, `ExpiringProductBean` will post messages to the queue so that `DiscountProduct` bean can pick them up and attempt to update products by applying the discount.
- Observe messages logged by the `DiscountProduct` bean. You should be able to see that a product called Cookie price was reduced, but Cake was not. This is because of the Validation constraint set on the Product entity that does not allow the price to be smaller than 1.
- You can verify this by either querying this product using the ProductApp-client application or by using SQL Console.

Practices for Lesson 6: Implementing SOAP Services by Using JAX-WS

Practices for Lesson 6: Overview

Overview

In these practices, you create a `PriceQuote` SOAP web service, implemented as a Session EJB, using JAX-WS API. You test this service using a WebLogic web service testing tool. You also produce a Java Client Application that invokes this `PriceQuote` service using JAX-WS API.



Practice 6-1: Exposing an Enterprise Java Bean as a JAX-WS Service

Overview

In this practice, you create a new Singleton Enterprise Java Bean and expose it as JAX-WS web service.

Assumptions

You have successfully completed all previous practices.

Tasks

1. In this practice, you are going to create a JAX-WS service called `PriceQuoteService` based on an existing WSDL file. You do not need to create these files. The `PriceQuote.xsd` and `PriceQuoteService.wsdl` files are located in the `/home/oracle/labs/resources` folder. Before you use WSDL and XSD file to create a service, examine these files to better understand exactly what to expect from the `PriceQuote` service.
   a. This is the source code of the `PriceQuote.xsd` file:

```
<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://demos/PriceQuote"
           elementFormDefault="qualified">
<xs:element name="Products">
<xs:complexType><xs:sequence>
   <xs:element name="productId" type="xs:int" maxOccurs="unbounded"/>
</xs:sequence></xs:complexType>
</xs:element>
<xs:element name="Quote">
<xs:complexType><xs:sequence>
 <xs:element name="quotedPrice" type="xs:decimal"/>
</xs:sequence></xs:complexType>
</xs:element>
<xs:element name="QuoteError">
<xs:complexType><xs:sequence>
 <xs:element name="message" type="xs:string"/>
</xs:sequence></xs:complexType>
</xs:element>
</xs:schema>
```

   Note: This schema defines an element, `Products`, which represents a collection of product ID values, an element, `Quote`, which contains a single numeric value of the `quotedPrice`, and a `QuoteError` element, which contains an error message.

b. The source code of the `PriceQuoteService.wsdl` file is presented here in several parts, with notes at the end of each part.

- **Part 1 WSDL declaration and namespace definitions:**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PriceQuote"
                 targetNamespace="http://demos/PriceQuote/Service"
             xmlns:tns="http://demos/PriceQuote/Service"
             xmlns:ns1="http://demos/PriceQuote"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Note: You can see all namespace definitions described in the root element of the WSDL document. The `ns1` namespace defines a prefix that points to a namespace used by the `PriceQuote.xsd` schema document. The `tns` namespace defines a prefix that points to a namespace used by this WSDL file.

- **Part 2 Abstract WSDL:**

```
<types><xsd:schema>
     <xsd:import namespace="http://demos/PriceQuote"
                 schemaLocation="PriceQuote.xsd"/>
</xsd:schema></types>
<message name="QuoteRequest">
     <part name="request" element="ns1:Products"/>
</message>
<message name="QuoteResponse">
     <part name="response" element="ns1:Quote"/>
</message>
<message name="QuoteFault">
     <part name="response" element="ns1:QuoteError"/>
</message>
<portType name="PriceQuote">
     <operation name="getQuote">
          <input message="tns:QuoteRequest"/>
     <output message="tns:QuoteResponse"/>
     <fault name="QuoteFault" message="tns:QuoteFault"/>
   </operation>
</portType>
```

Note: In this part of the WSDL file you see `PriceQuote.xsd import`, a declaration of `QuoteRequest`, which contains a Products element, `QuoteResponse` containing a Quote, and `QuoteFault` with a `QuoteError` element inside. These messages are then used as

`input`, `output`, and `fault` parts of the `getQuote` operation defined within the `PriceQuote` port type.

- **Part 3 Concrete WSDL:**

```
<binding name="PriceQuoteSOAP" type="tns:PriceQuote">
      <soap:binding transport="http://schemas.xml.org/soap/http"
                        style="document"/>
      <operation name="getQuote">
      <input><soap:body use="literal" parts="request"/></input>
   <output><soap:body use="literal" parts="response"/></output>
   <fault name="QuoteFault">
           <soap:fault use="literal" name="QuoteFault"/>
   </fault>
   </operation>
</binding>
<service name="PriceQuoteService">
      <port name="PriceQuote" binding="tns:PriceQuoteSOAP">
      <soap:address
           location="http://localhost:7001/demos/PriceQuoteService"/>
   </port>
</service>
```

Note: In this part of the WSDL file, you see a description of the SOAP binding for the `getQuote` operation. You also see description of the SOAP service and its endpoint address.

- **Part 4 WSDL root element is now closed:**

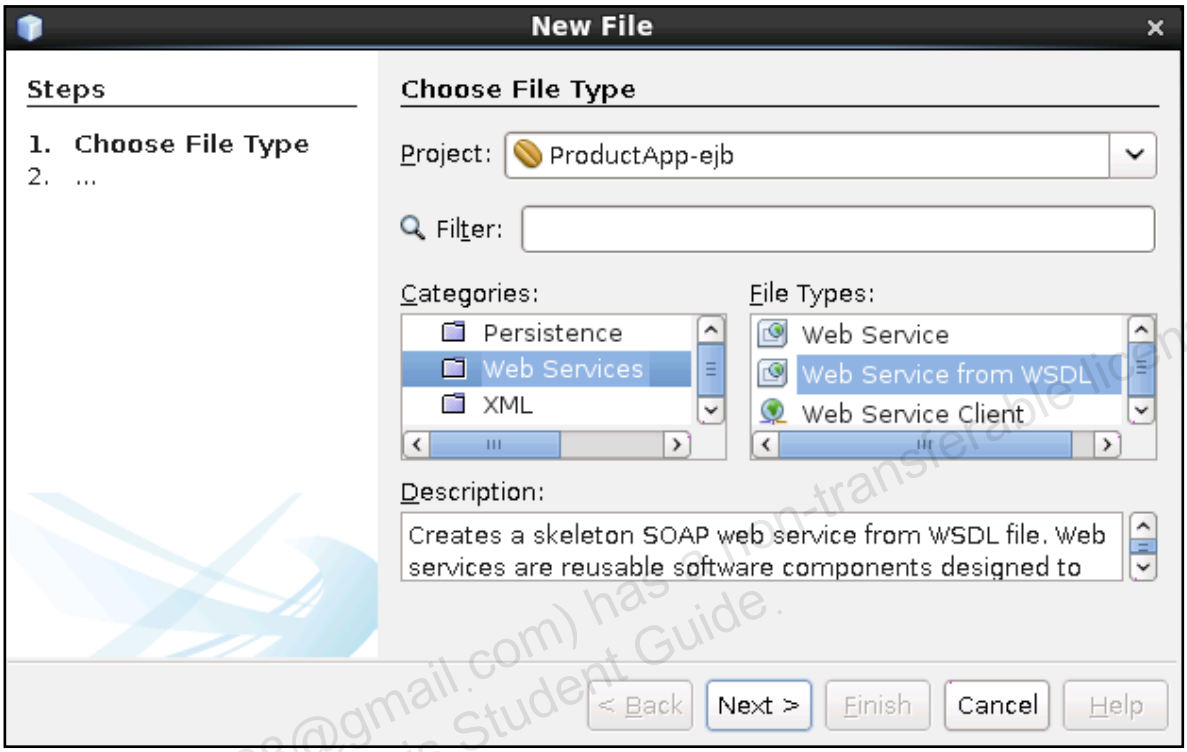
```
</definitions>
```

Note: This is the end of the WSDL file.

c. Select the **ProductApp-ejb** project.

d. Select **File > New File**.

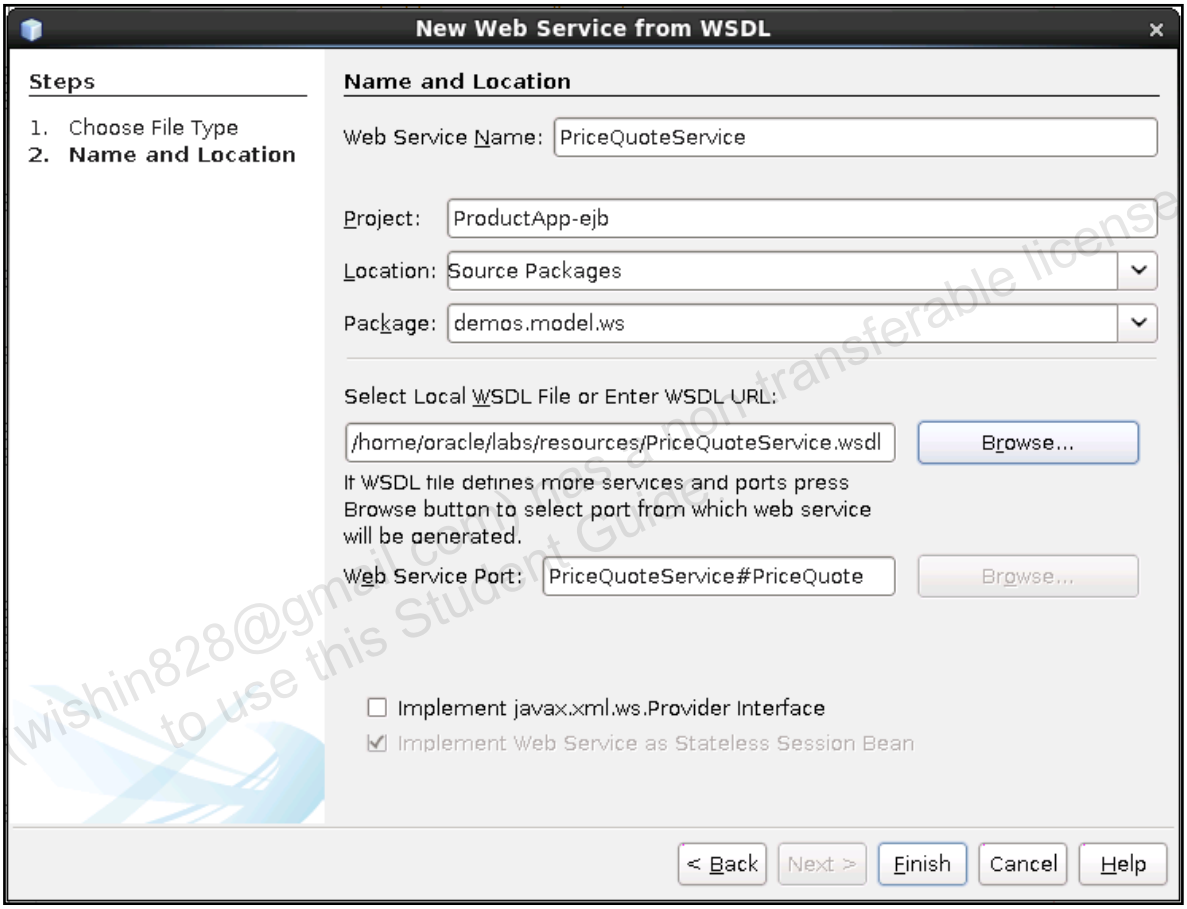
e. Select **Web Services** from the list of Categories and **Web Service from WSDL** from the list of File Types.



f. Click **Next**

g. In the New Web Service from WSDL dialog box, set the following properties:

- Web Service Name: `PriceQuoteService`
- Package: `demos.model.ws`
- Click the **Browse** button and select the `PriceQuoteService.wsdl` file located in the `/home/oracle/labs/resources` folder.
- The Web Service Port property would then be set for you automatically.



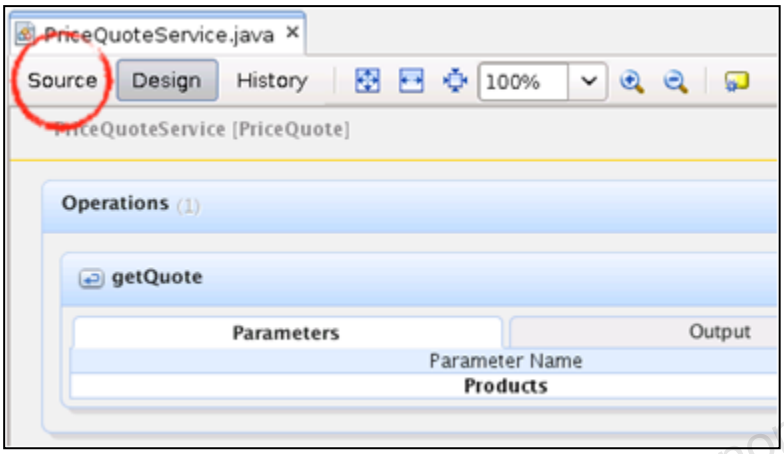
h. Click **Finish.**

Note: A new web service from WSDL file wizard has just produced a new Stateless Session Enterprise Java Bean mapped to a WSDL file with JAX-WS annotations. You are going to use this bean to write Price Quote Service implementation logic.

Note: The same wizard had also generated JAXB artifacts (classes such as `Products`, `Quote`, and `QuoteError`) that map XML schema objects to Java classes that represent input, output, and fault messages of the web service operation. You may (optionally) open the `PriceQuoteService.WSDL` file and `PriceQuote.XSD` files located in the `/home/oracle/labs/resources` folder and observe how elements that they define were mapped to Java classes located in `ProductApp-ejb` project under the "Generated Sources (jax-ws)" node and on to the `PriceQuoteService` Session bean class.

2. Prepare the `PriceQuoteService` Stateless Session EJB to host your web service logic implementation:
   a. Open the source code editor for the **`PriceQuoteService`** class.

      Note: When you double-click the `PriceQuoteService` Java class in the Projects Navigator panel, NetBeans can open it in Design mode rather than Source code editor mode. To switch to source code, click the **Source** button.



   b. Add a static Logger reference to the `PriceQuoteService` class.

```
private static final Logger logger =
Logger.getLogger(PriceQuoteService.class.getName());
```

   c. Add an import: `java.util.logging.Logger`
   d. Inject a reference to the `ProductFacade` EJB object.

```
@EJB
private ProductFacade productFacade;
```

   e. Add imports: `demos.model.ProductFacade` and `java.ejb.EJB`
   f. Remove package prefixes in front of the **`Quote`**, **`Product`**, and **`QuoteFault`** declarations. This makes it more convenient to use these classes to write the implementation of the `getQuote` operation.
   g. Add the following imports:

```
demos.pricequote.Products
demos.pricequote.Quote
demos.pricequote.service.QuoteFault
```

3. Implement Price Quote Service functionality within the `getQuote` method:
   a. Remove all placeholder code from the body of the `getQuote` method (this is the line of code that throws an `UnsupportedOperationException`).
   b. Inside the `getQuote` method, add a `try` block followed by two `catch` blocks. First one of these `catch` blocks should catch the `QuoteFault` exception, write information about it to the log, and throw the same exception again. The second `catch` should

handle all other exceptions, write information about these exceptions to the log, and then wrap them into a `QuoteFault` and throw it out of the `getQuote` method.

```
try{
  // you will add more code here in the next step of this practice
}catch(QuoteFault ex){
  logger.log(Level.INFO,"Error producing price quote",ex);
  throw ex;
}catch(Exception ex){
  logger.log(Level.INFO,"Error producing price quote",ex);
  QuoteError error = new QuoteError();
  error.setMessage(ex.getMessage());
  throw new QuoteFault("Error producing price quote",error);
}
```

c. Add two imports: `java.util.logging.Level` and `demos.pricequote.QuoteError`

Note: The `getQuote` method is declared to throw a `QuoteFault` exception, which can be marshaled by a JAXB API as a SOAP fault message and transported to the web service caller. However, it is considered to be a good practice not only to notify a caller about the problem, but also log this error on a server. This is why you were instructed to catch exceptions, log them, and then throw them again.

d. Inside the `try` block, add code that extracts a list of product ID values from the `Products` parameter of the `getQuote` method:

```
List<Integer> productIds = request.getProductId();
```

e. Add an import: `java.util.List`

f. Verify that this list is not empty:

```
if(productIds.isEmpty()){
     // you will add more code here in the next step
}
```

g. Inside this `if` block, add code that prepares a new `QuoteError` object that should contain details of the web service fault and place this object into the new `QuoteFault` object that represents the exception you need to throw.

```
QuoteError error = new QuoteError();
error.setMessage("Provide at least one product id to get a quote");
throw new QuoteFault("Error producing price quote",error);
```

h. After the end of the `if` block add an invocation of the `findTotal` method of the ProductFacade EJB. This method accepts a list of product id values as an argument and returns an object array of two values: number of products and total price.

```
Object[] values = productFacade.findTotal(productIds);
```

i. Extract the first value from the object array of results returned by the `getTotal` method. This value is a long number that represents the number of products found by the query. Compare this number to the number of product ID values that the `getQuote` method received as an argument. If the number of `productIds` list is greater than the number of products found by the `getTotal` operation, then it means that not all product IDs that you have received are valid. Therefore, you should prepare and throw a `QuoteFault` to indicate that.

```
Long count = (Long)values[0];
if(count < productIds.size()){
   QuoteError error = new QuoteError();
   error.setMessage("Unable to locate some of these products: "+
   productIds);
   throw new QuoteFault("Error producing price quote",error);
}
```

j. After the end of this `if` block, add code that extracts a second value out of the array returned by the `getTotal` operation. This value represents a BigDecimal number that is a total price of the quoted products. Create a new Quote object, place the total quoted price value into it and return this quote object.

```
BigDecimal quotedPrice = (BigDecimal)values[1];
Quote quote = new Quote();
quote.setQuotedPrice(quotedPrice);
return quote;
```

k. Add an import: `java.math.BigDecimal`

4. Compile the **ProductApp-ejb** project using **Clean and Build**.
5. Right-click the **ProductApp** project and deploy it.

Practice 6-2: Testing the JAX-WS Service

Overview

In this practice, you test PriceQuoteService by using an online web services testing tool supplied with the WebLogic server.

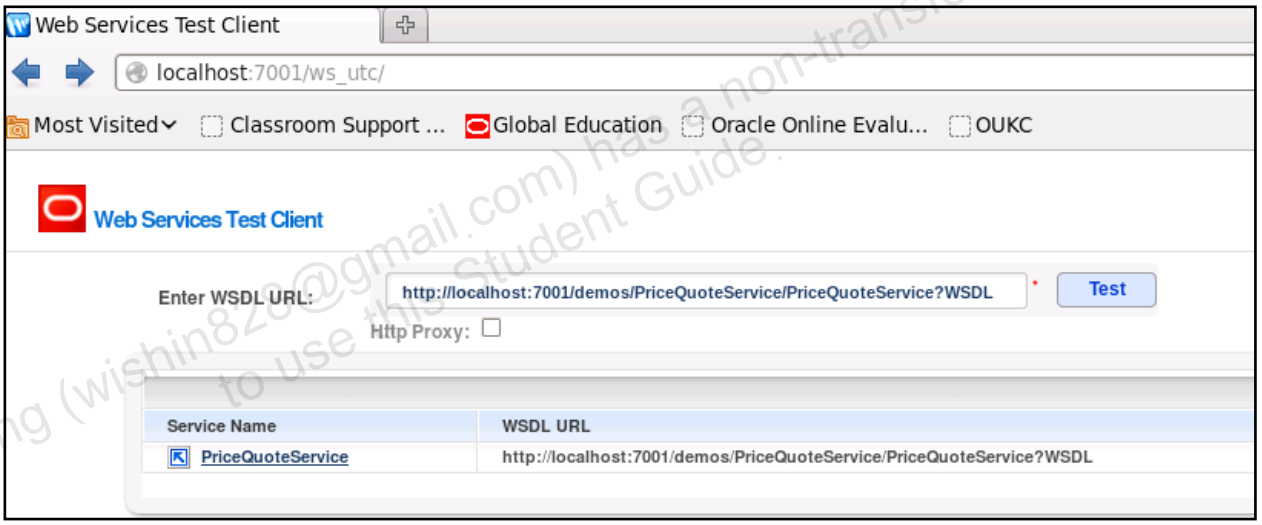
Assumptions

You have successfully completed all previous practices.

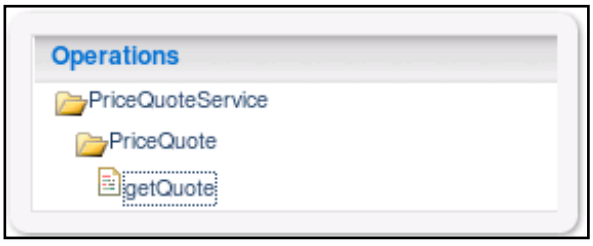
Tasks

1. Access the online web service testing tool:
   a. Open browser and navigate to the following url: `http://localhost:7001/ws_utc`
   b. Enter your PriceQuoteService WSDL URL into the **WSDL URL** field:

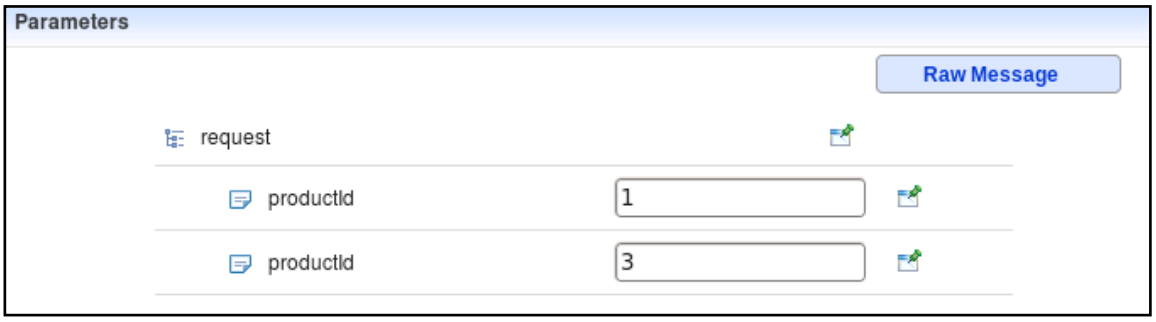
`http://localhost:7001/demos/PriceQuoteService/PriceQuoteService?WSDL`



   c. Click **Test**.
   d. One the left side of the screen, expand the **Operations** panel, and click the **getQuote** operation.

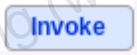


2. Test normal invocation of the PriceQuoteService:
   a. Set two product ID values **1** and **3** as `getQuote` operation parameters.



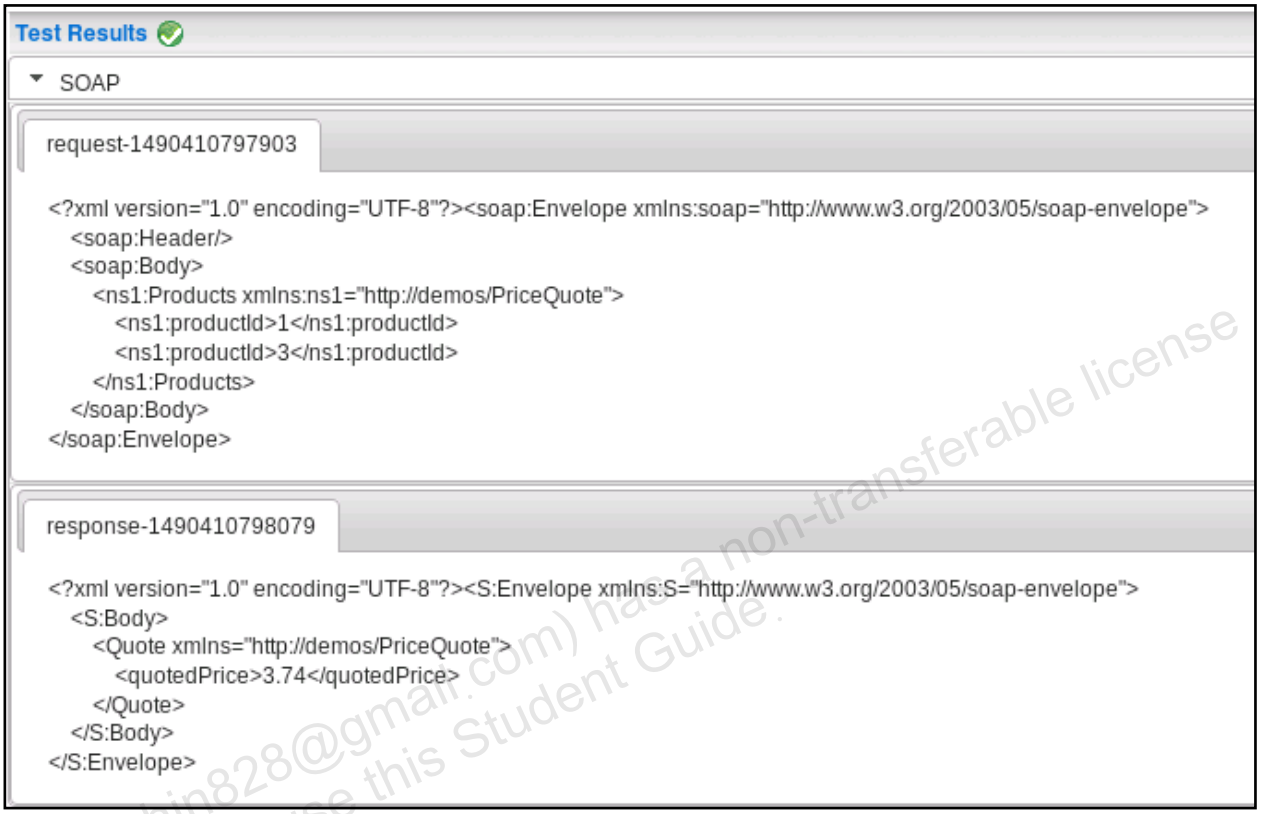
   b. Optionally, you may click a **Raw Message** button to view or edit the actual soap massage that you will be sending to the service. A Form Entry button will switch this screen back.

   c. After you set the operation parameters and have formed a SOAP message, invoke the `getQuote` operation by clicking the **Invoke** button at the bottom-right corner of the screen.

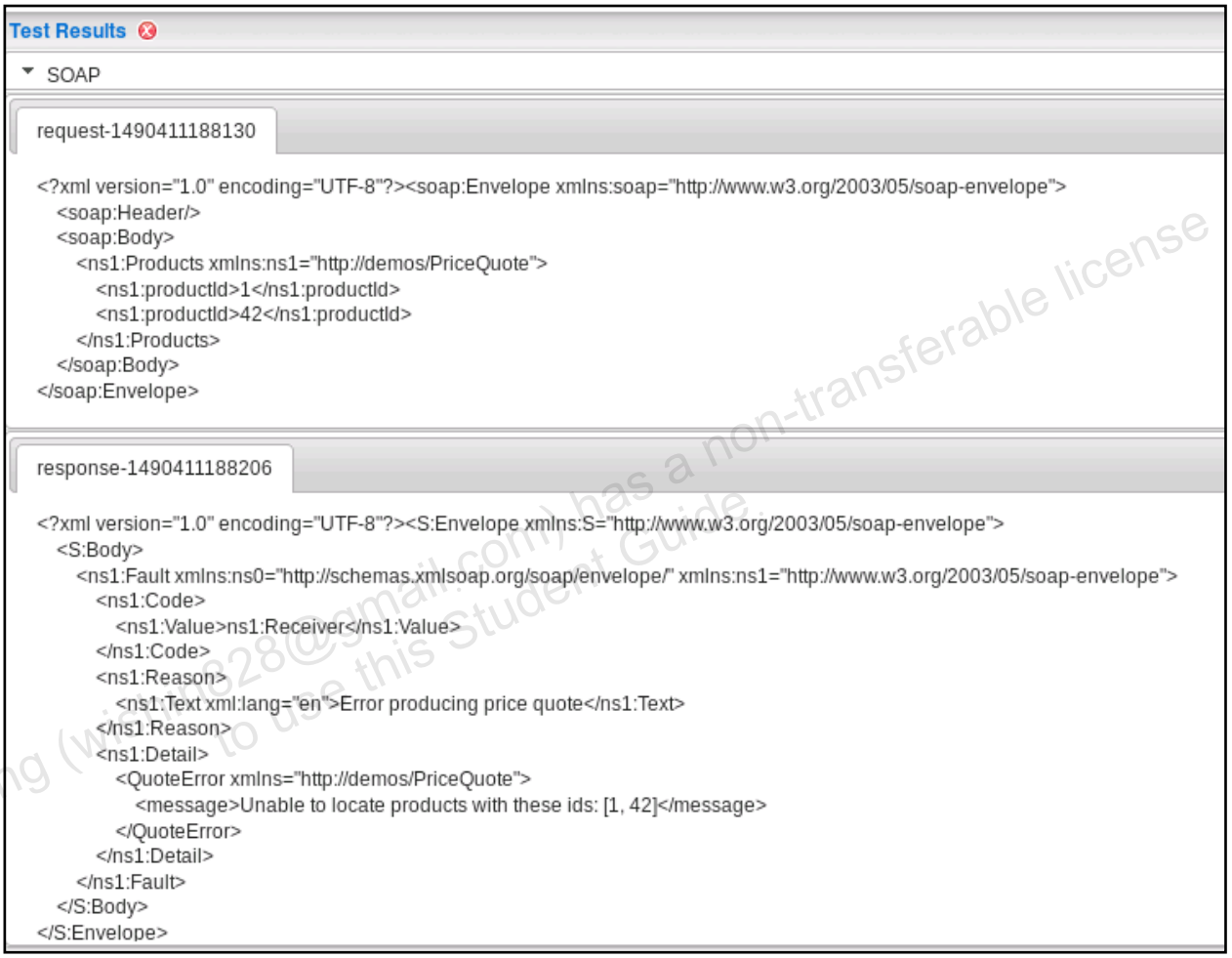


d. Scroll down to the Test Results section and observe the SOAP request and response objects that you just exchanged with the service.

Note: The actual quoted value that you will receive from the server may differ from this example, because your database may contain different prices for products.



3. Test an incorrect invocation of the PriceQuoteService:
   a. Scroll up to the Parameters section and change one of the product ID values to be nonexistent ID. For example, request a quote for products **1** and **42**.
   b. Click the **Invoke** button.
   c. Go down to the **Test Results** section and expand the **SOAP** node to observe the SOAP Fault message returned from the service.



   .
   d. Scroll up to the **Parameters** section.
   e. Switch to **Raw Message** mode and remove all product IDs from the list, so that you can test the case when an empty list is passed to a service. This is what your message should look like:

```
<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <ns1:Products xmlns:ns1="http://demos/PriceQuote">
    </ns1:Products>
  </soap:Body>
</soap:Envelope>
```

f. Click the **Invoke** button again.

g. Observe a different fault message in the SOAP section of the **Test Results** panel.

Practice 6-3: Creating a JAX-WS Client

Overview

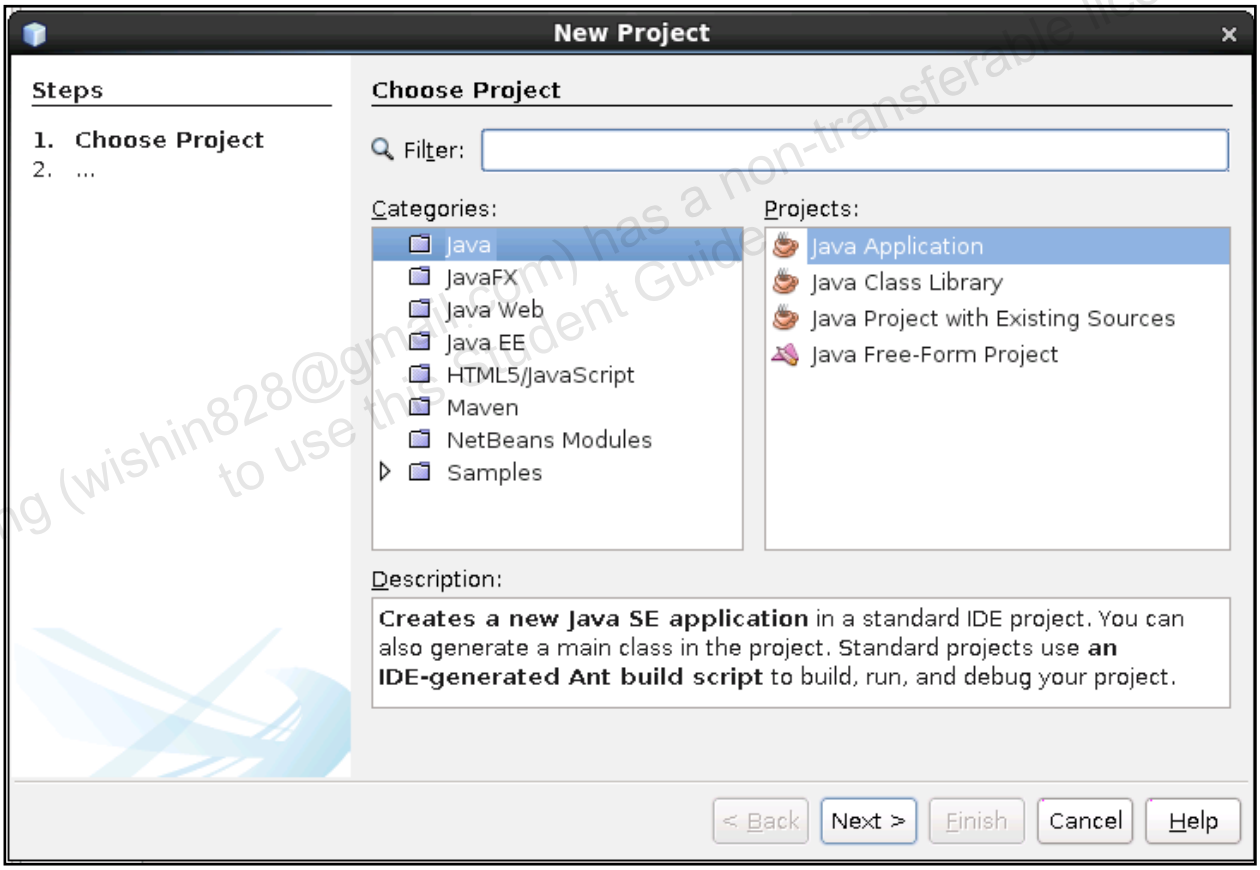
In this practice, you create a Java client that can interact with the PriceQuoteService.

Assumptions

You have successfully completed all previous practices.

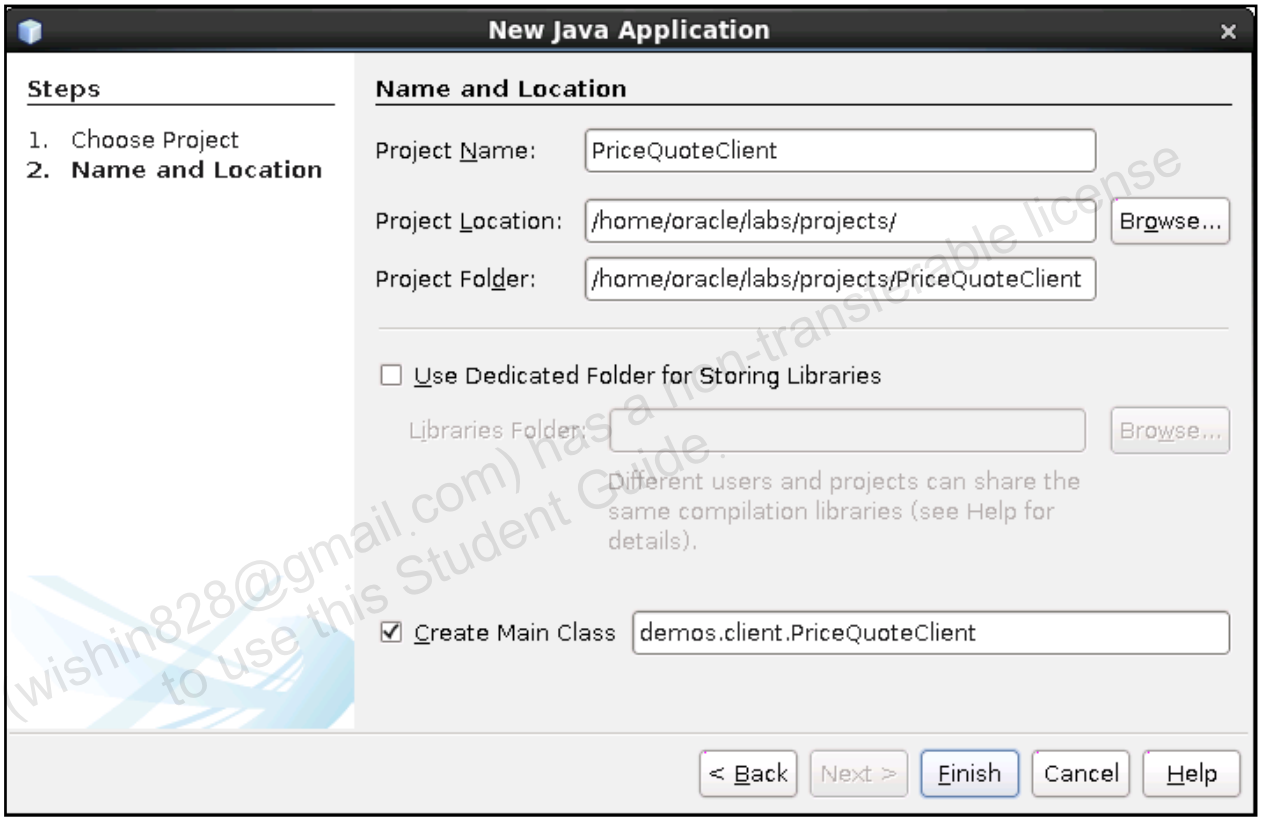
Tasks

1. Create a new Java application project:
   a. Select **File > New Project**.
   b. Select **Java** from the list of Categories and **Java Application** from the list of Projects.



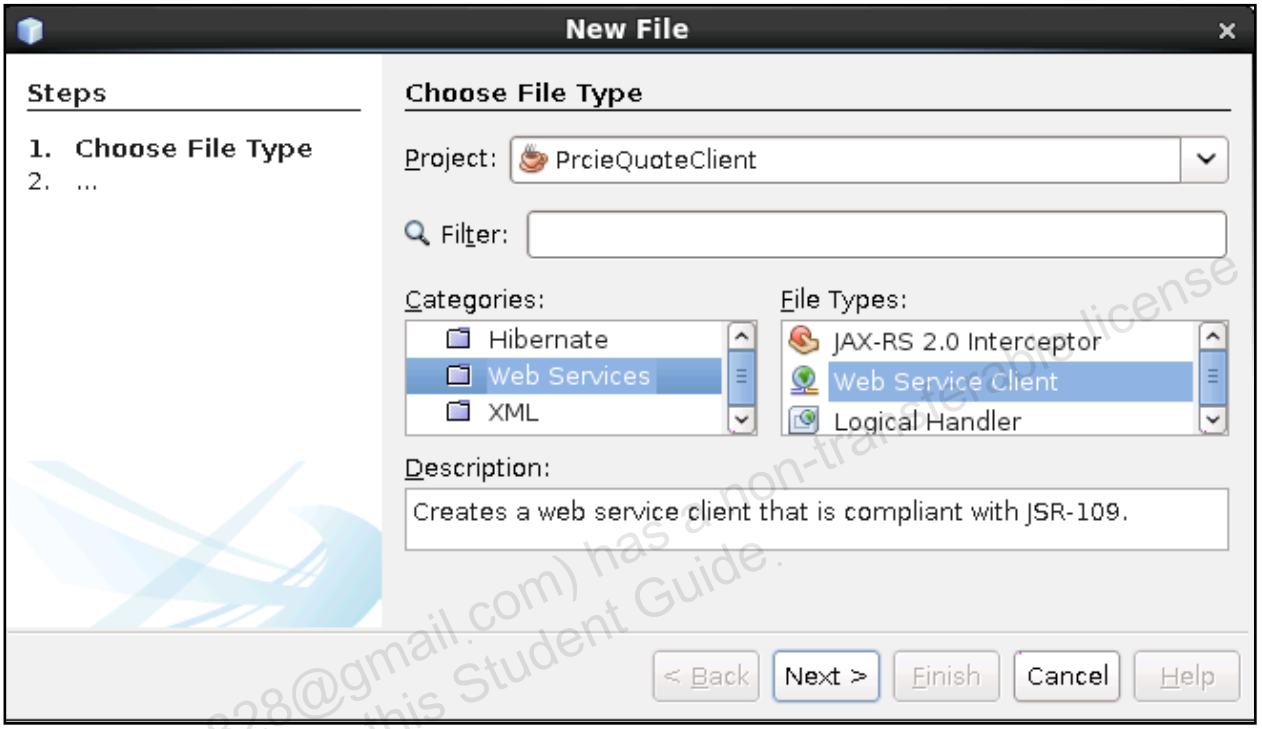
   c. Click **Next.**

d. In the New Java Application dialog box, set following properties:
   - Project Name: `PriceQuoteClient`
   - Project Location: `/home/oracle/labs/projects`
   - Project Folder: `/home/oracle/labs/projects/PriceQuoteClient`
   - Select the **Create Main Class** check box and set the class name to: `demos.client.PriceQuoteClient`



e. Click **Finish**.

2. Create a new Java web service client:
   a. Select the PriceQuoteClient project.
   b. Select **File > New File**.
   c. Select **Web Services** from the Categories list and **Web Service Client** from the File Types list.



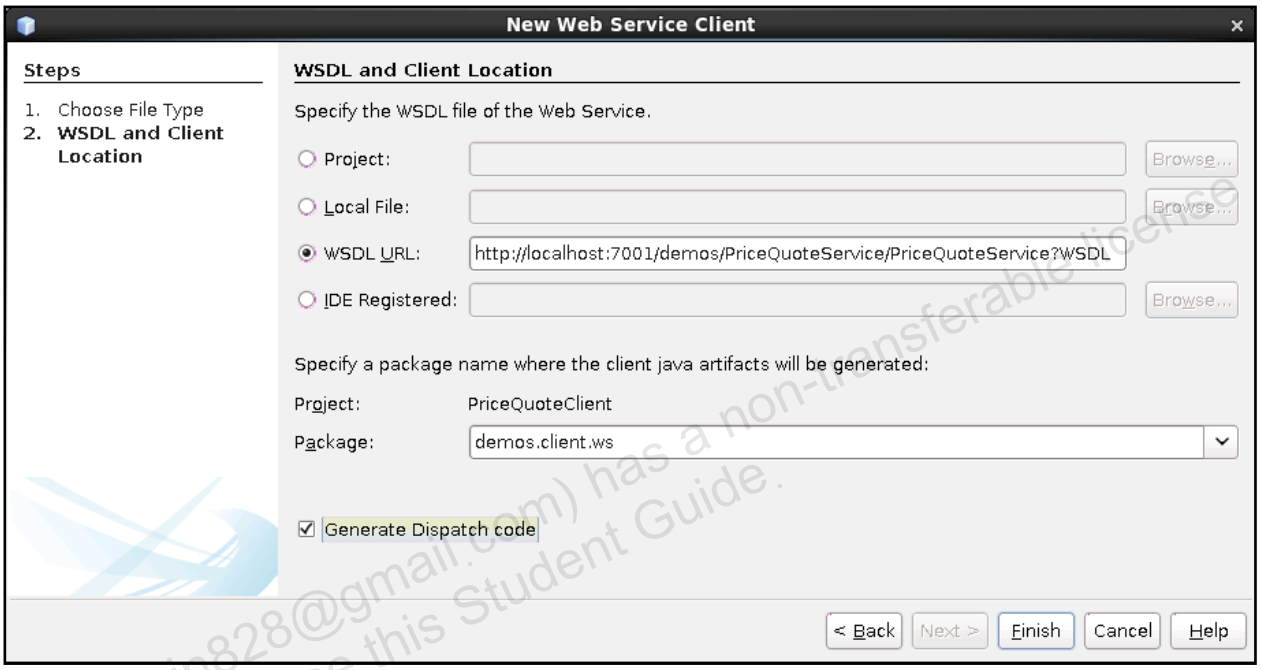
   d. Click **Next**.

e. In the New Web Service Client dialog box, set the following properties:

- Select **WSDL URL** from the list of option buttons.
- Set **WSDL URL** property as:

`http://localhost:7001/demos/PriceQuoteService/PriceQuoteService?WSDL`

- Set Package: `demos.client.ws`
- Select the “Generate Dispatch code” check box.



f. Click **Finish.**

Note: This wizard has generated JAXB classes (such as `Products`, `Quote`, and `QuoteError`) that represent input, output, and fault messages of the `PriceQuoteService` and JAX-WS classes (such as `PriceQuoteService`, `PriceQuote`, and `QuoteFault`) that represent the corresponding Service, its `PortType` containing operation `getQuote`, and a SOAP fault.

3. Add code to the PriceQuoteClient class to set up Logger:
   a. Open class **`PriceQuoteClient`** from the **`demos.client`** package in the **`PriceQuoteClient`** project.
   b. Add and initialize a static Logger object reference to be used by this class:

```
private static final Logger logger =
Logger.getLogger(PriceQuoteClient.class.getName());
```

   c. Add an import: `java.util.logging.Logger`

4. Add exception handling logic to the main method of the `PriceQuoteClient` class:

   You need to catch `QuoteFault` exceptions, extract fault information from it, and write it to the log. You also need to catch other exceptions and write information about them to the log.

   a. Inside the main method of the `PriceQuoteClient` class add the following code:

```
try {
    // more code will be added here to invoke PriceQuoteService
} catch (QuoteFault ex) {
    logger.log(Level.INFO, ex.getFaultInfo().getMessage());
} catch (Exception ex) {
    logger.log(Level.SEVERE, ex.getMessage(),ex);
}
```

   b. Add the following imports:

```
java.util.logging.Level
demos.client.ws.QuoteFault
```

5. Add code that invokes `PriceQuoteService`:

   a. Inside the `try` block in the main method of the `PriceQuoteClient` class, add code that instantiates the `PriceQuoteService` object.

```
PriceQuoteService service = new PriceQuoteService();
```

   b. Add an import: `demos.client.ws.PriceQuoteService` class

   c. Using the `PriceQuoteService` object, initialize a reference to the PriceQuote object.

   Note: This object represents a port type element of the WSDL file and is used to invoke operations described by this port.

```
PriceQuote port = service.getPriceQuote();
```

   d. Add an import: `demos.client.ws.PriceQuote` class

   e. Form a new service request object. Service request is represented by the Products class. Create a new Products object, then get a list of product ID integer values from it and add two product IDs to this list. You can use same product ID values (**1** and **3**) that you used when you tested this service with the online testing tool.

```
Products request = new Products();
List<Integer> ids = request.getProductId();
ids.add(1);
ids.add(3);
```

   f. Add the following imports:

```
java.util.List
demos.client.ws.Products
```

g. Invoke the `getQuote` operation upon the `PriceQuote` port object. Pass the `Products` request object as an argument. Assign the response returned by the `getQuote` operation to a `Quote` object.

```
Quote quote = port.getQuote(request);
```

h. Add an import: `demos.client.ws.Quote` class

i. Print quoted price to the console.

```
System.out.println(quote.getQuotedPrice());
```

6. Compile the **PriceQuoteClient** project using **Clean and Build**.
7. Run the **PriceQuoteClient**. You should be able to observe quoted price values printed to the console.
8. Optionally (if you have time) change one of the product ID values to a nonexistent product ID and run **PriceQuoteClien**t again to observe a price quote fault. You may also try to comment out both lines of code that add product IDs to the list and run the project again to see a different error message produced by the quote fault.

Practices for Lesson 7: Creating Java Web Applications by Using Servlets

Practices for Lesson 7: Overview

Overview

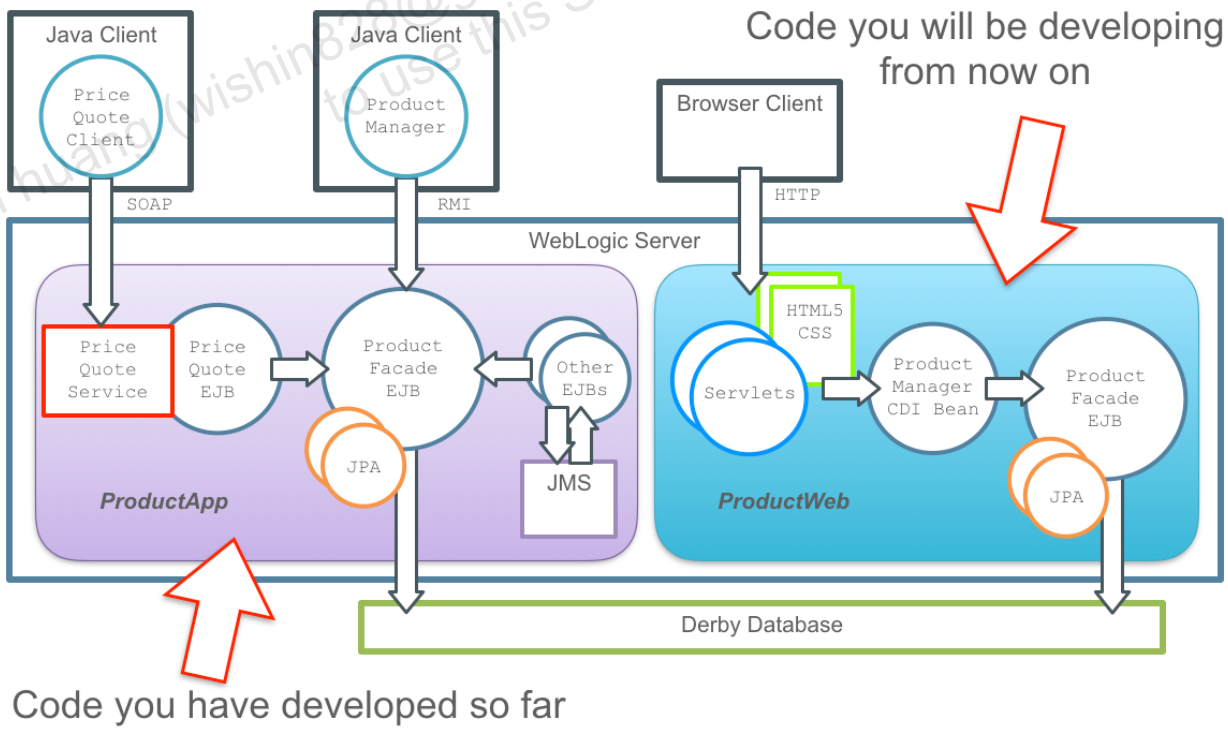
In these practices, you create a Java web module. You will:

- Use new Java EE7 feature that allows placement of the EJB and JPA code directly inside the web module
- Use HTML 5 and CSS to construct a user interface
- Create an HTML form to initiate a product search
- Create a Java servlet to display a product list
- Add exception handling logic to the web module

You have already developed a Java EE EJB application by using JPA Entities, remotely invoked EJBs, SOAP Services, JMS, and Timer components.

From now on you will be working on a new Java EE web application module. This new module, called `ProductWeb`, contains a copy of the `ProductFacade` EJB and exactly the same JPA code as you created for the `ProductClient` and `ProductApp` modules. It uses a refactored copy of the `ProductManagement` class. However, this time it is formed as a CDI bean that presents `ProductFacade` EJB functionalities to the web components of the application.

In these practices, you will develop a web application that uses HTML5, CSS, servlets, and CDI Beans to present `Product Management` functionalities to browser clients.



Practice 7-1: Creating a Java Web Application

Overview

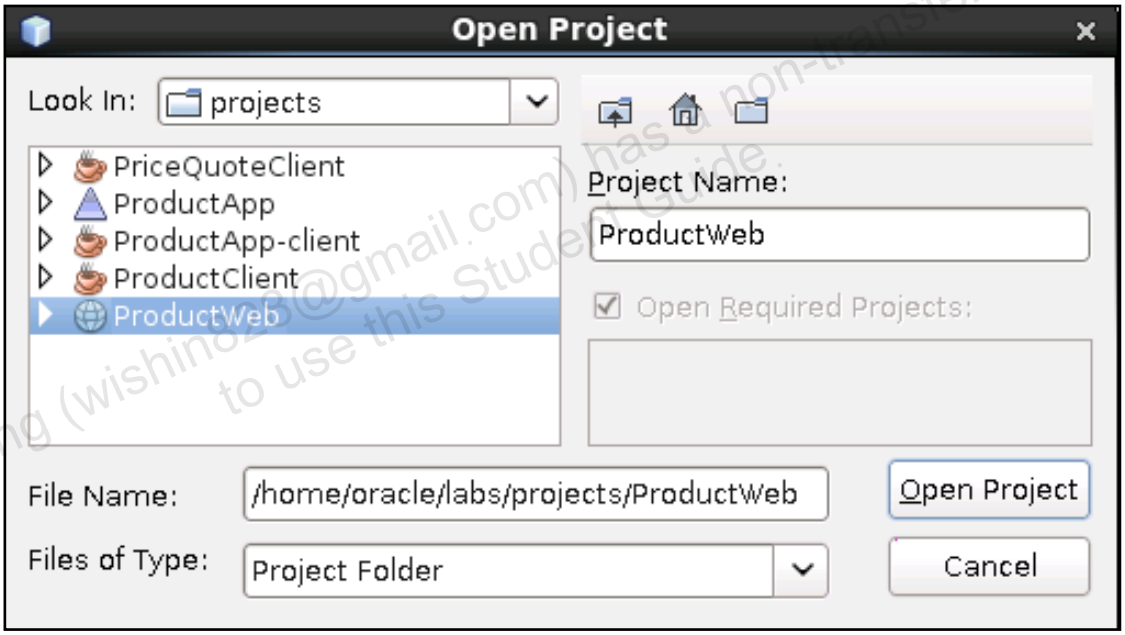
In this practice, you modify a Java web application module project: You add a cascading stylesheet file to it, design an index page and build an HTML page to contain a product query form.

Assumptions

You have successfully completed all previous practices.

Tasks

1. Open the `ProductWeb` Java web application project:
   a. Select **File > Open Project**.
   b. Select the `ProductWeb` project from the `/home/oracle/labs/projects` folder.



   c. Click **Open Project**.

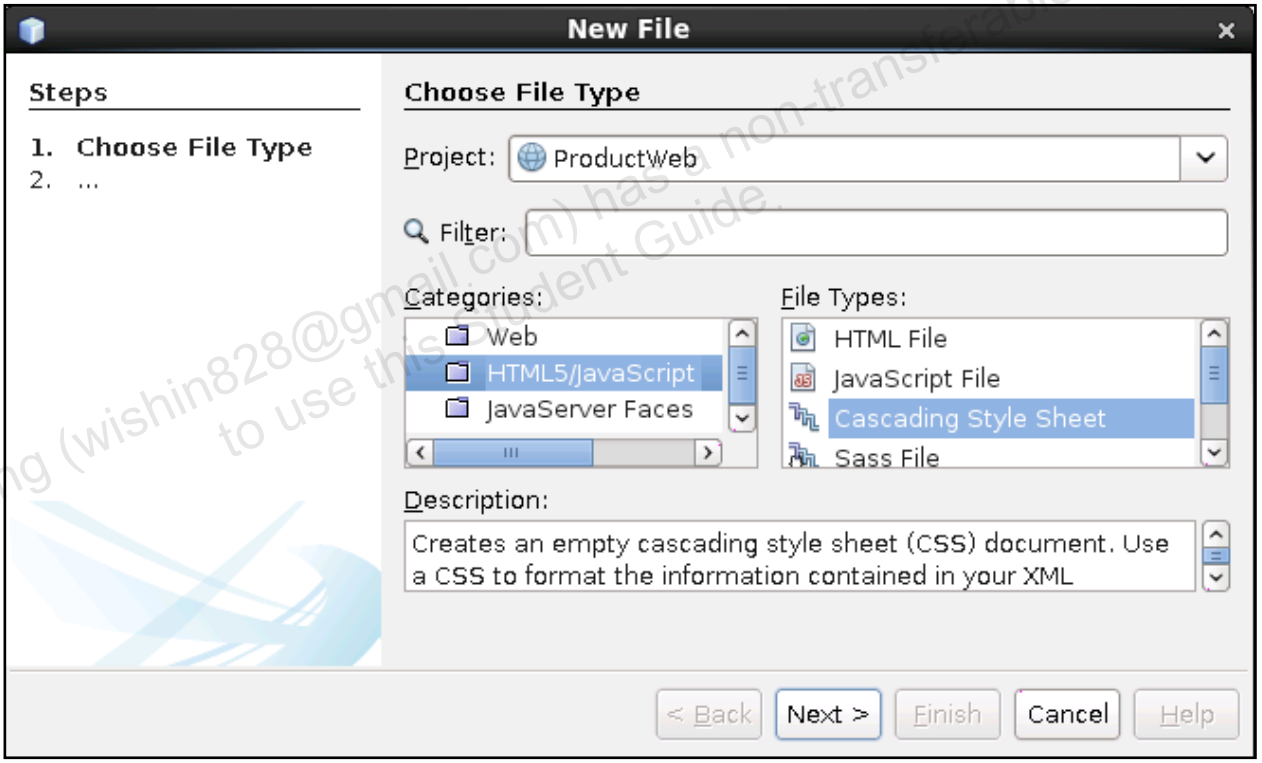
Note: Packaging an EJB module directly into a web module is a new Java EE7 feature.

- To demonstrate this capability, you open the `ProductWeb` project, which contains a copy of `ProductFacade` EJB and exactly the same JPA code as the `ProductApp-ejb` project you worked on before.
- Of course an EJB module packed inside a web module would not have all the capabilities of a full EJB container–deployed module. No Remote Interface or Message-Driven beans are available in the `ProductWeb` module.
- Another interesting thing to note is that in the `ProductWeb` project a `ProductManager` class is now annotated as a RequestScoped CDI bean. This

annotation would allow injection of the `ProductManager` object into servlets, Java Server Pages, Java Server Faces, and other web components. `ProductManager` still has the same role within the application, acting as a proxy between whatever invoker and the `ProductFacade` EJB.

- (Optionally) You may observe these differences by opening `ProductFacade` and `ProductManager` classes located under the Source Packages node in the `ProductWeb` project.

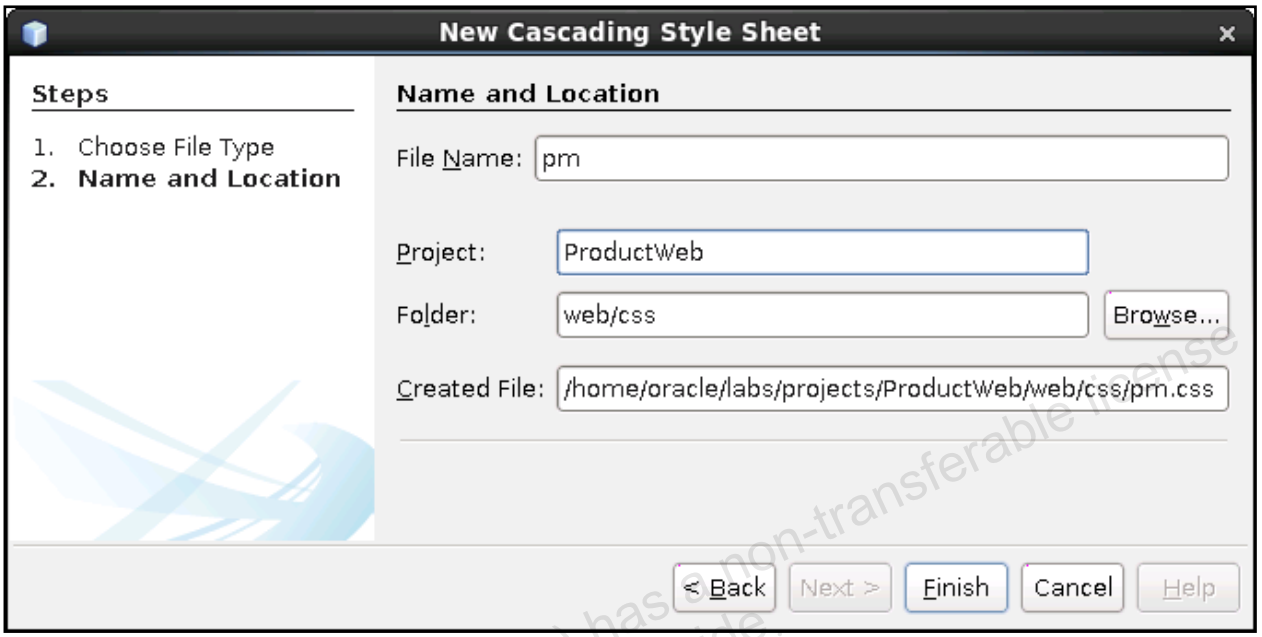
2. Add a new cascading stylesheet file to the `ProductWeb` project:
   a. Select the **`ProductWeb`** project.
   b. Select **File > New File**.
   c. Select **HTML5/JavaScript** from the list of Categories and **Cascading Style Sheet** from the list of File Types.



   d. Click **Next**.

e. In the New Cascading Style Sheet dialog box, set the following properties:

- File Name: `pm`
- Folder: `web/css`



f. Click **Finish**

g. Add style classes to the `pm.css` file

Note: You can simply replace the entire contents of the `pm.css` file with text from the `pm.css` file located in the `/home/oracle/labs/resources` folder.

This is the resulting source code:

```
.header {background:#7777EE;color:white;font-size:120%;
         font-weight:bold;padding:15px}
.footer {background:#7777EE;color:white;padding:15px;font-size:80%;}
.nav {background:#DEDEFF;color:white;padding:15px;
      border-bottom-color:white;border-top-color:white;
      border-bottom-style:solid;border-top-style:solid}
.content {background:#DEDEFF;padding:15px}
.error {border-color:red;border-style:double;background:#DEDEFF;
        padding:3px;margin:3px}
.info {border-color:green;border-style:double;background:#DEDEFF;
        padding:3px;margin:3px}
.warn {border-color:yellow;border-style:double;background:#DEDEFF;
        padding:3px;margin:3px}
.data {padding:3px}
.field label {display:inline-block;width:140px;margin:2px}
.field input {display:inline-block;margin:2px}
```

Note: If you are interested in learning more about what this CSS code does, Oracle University offers a course called "JavaScript and HTML5: Develop Web Applications."

3. Modify the `index.html` page:
   a. Open the `index.html` file located under the Web Pages node in the `ProductWeb` project.
   b. Modify `index.html` code to define the page structure:

      Note: You can simply replace the entire contents of the `index.html` file with text from the `template.html` file located in the `/home/oracle/labs/resources` folder.

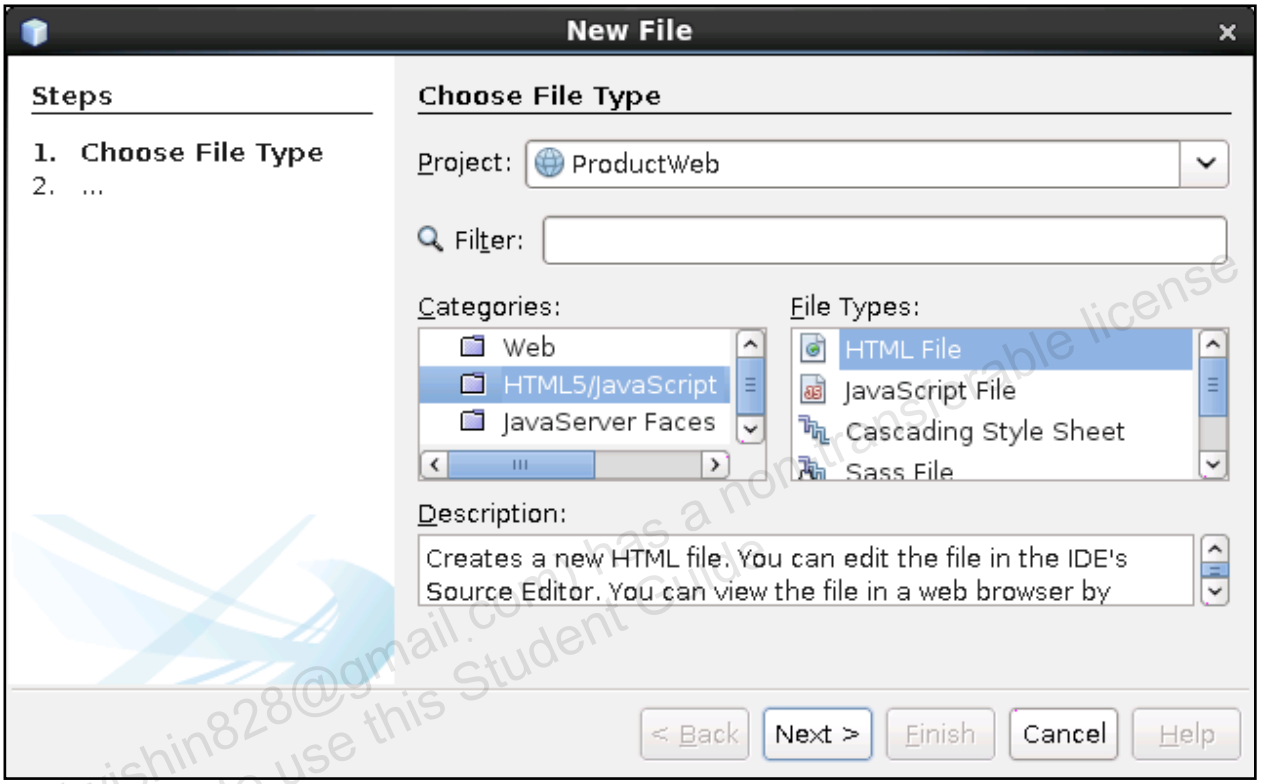
      This is the resulting source code:

```
<!DOCTYPE html>
<html>
      <head>
<meta charset='UTF-8'>
<meta name='viewport' content='width=device-width, initial-scale=1.0'>
<link rel='stylesheet' type='text/css' href='/pm/css/pm.css'>
      <title>PAGE TITLE</title>
      </head>
   <body>
            <header class='header'>PAGE HEADER</header>
            <nav class='nav'>PAGE NAVIGATION</nav>
            <section class='content'>
                  PAGE CONTENT
            </section>
            <footer class='footer'>PAGE FOOTER</footer>
    </body>
</html>
```

Note: This is a generic template that you are going to use for all HTML files and Java Server Pages produced within this set of practices.

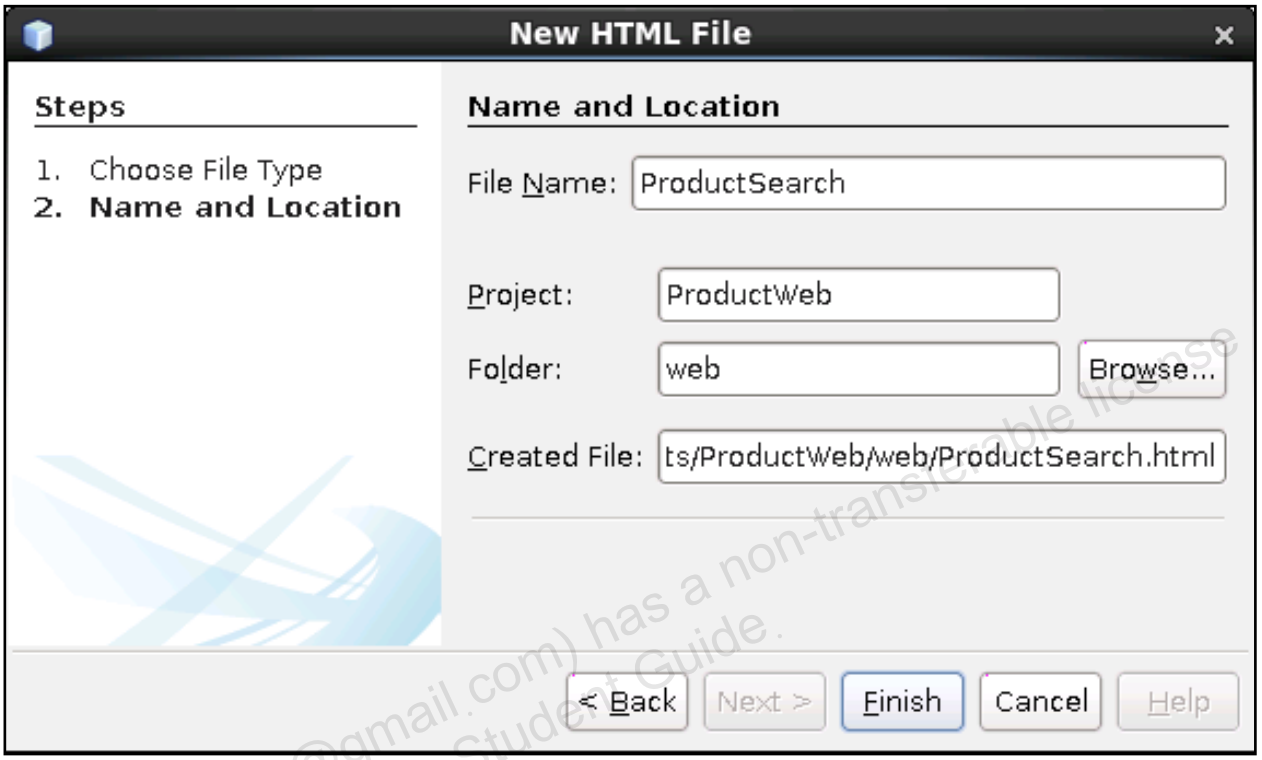
   c. Change the index page title, header, navigation, and footer elements.
      - Replace the PAGE TITLE text with: `Product Management`
      - Replace the text PAGE HEADER with: `Product Management`
      - Replace the text PAGE FOOTER with: `Product Management Application`
   d. Replace the text PAGE NAVIGATION with the link back to product search page:
      `<a href="/pm/ProductSearch.html">Product Search</a>`
   e. Replace PAGE CONTENT with a divider containing the following text:
      `<div>Welcome!</div>`

4. Create a Product Search HTML page:
   a. Select the **`ProductWeb`** project.
   b. Select **File > New File**.
   c. Select **HTML5/JavaScript** from the list of Categories and **HTML File** from the list of File Types.



   d. Click **Next**.

e. In the New HTML File dialog box, set the following properties:
   - File Name: ProductSearch
   - Folder: `web`



f. Click **Finish**.

5. Design the Product Search page:
   a. Modify `ProductSearch.html` code to define the page structure. Replace the entire contents of the `ProductSearch.html` file with text from the `template.html` file located inside the `/home/oracle/labs/resources` folder.
   b. Change the `ProductSearch` page title, header, navigation, and footer elements.
      - Replace the text PAGE TITLE with: `Find Products`
      - Replace the text PAGE HEADER with: `Find Products`
      - Replace the text PAGE FOOTER with: `Enter product name. For partial name search use % as wildcard.`
   c. Replace the text PAGE NAVIGATION with the link back to the home page (`index.html`):

```
<a href="/pm">Home</a>
```

Note: A Java EE web module implicitly assumes that files with names `index.html` and `index.jsp` are included in the welcome file list configuration. Therefore, if you just access your web archive context URL (in this case `/pm`), you will be served `index.html` file by default. Later in this course you will replace this default page. That

is why in this practice you are asked not to hardcode the name of the welcome page in your application.

d. Replace the content area of the ProductSearch page with an HTML from. This form should submit itself to a servlet that you will create later in this practice. This servlet would be mapped to URL `list`. You should use HTTP method `POST` to submit this form.

   Replace the text PAGE CONTENT with:

```
<form action="list" method="POST">
</form>
```

e. Inside this form element, add an HTML divider containing a text input item with a label to represent a field for a product name to be used in a product search.

```
<div class="field">
     <label for="name">Product Name:</label>
    <input type="text" id="name" name="p_name" required>
</div>
```

   Note: An HTML div element is used to apply style adjustments to the field and label, defined by the CSS file.

f. Inside this form, after the divider element that you just added, add another divider containing a submit button for this form.

```
<div class="field">
 <input type="submit" value="Find">
</div>
```

Practice 7-2: Creating a Product Search Servlet

Overview

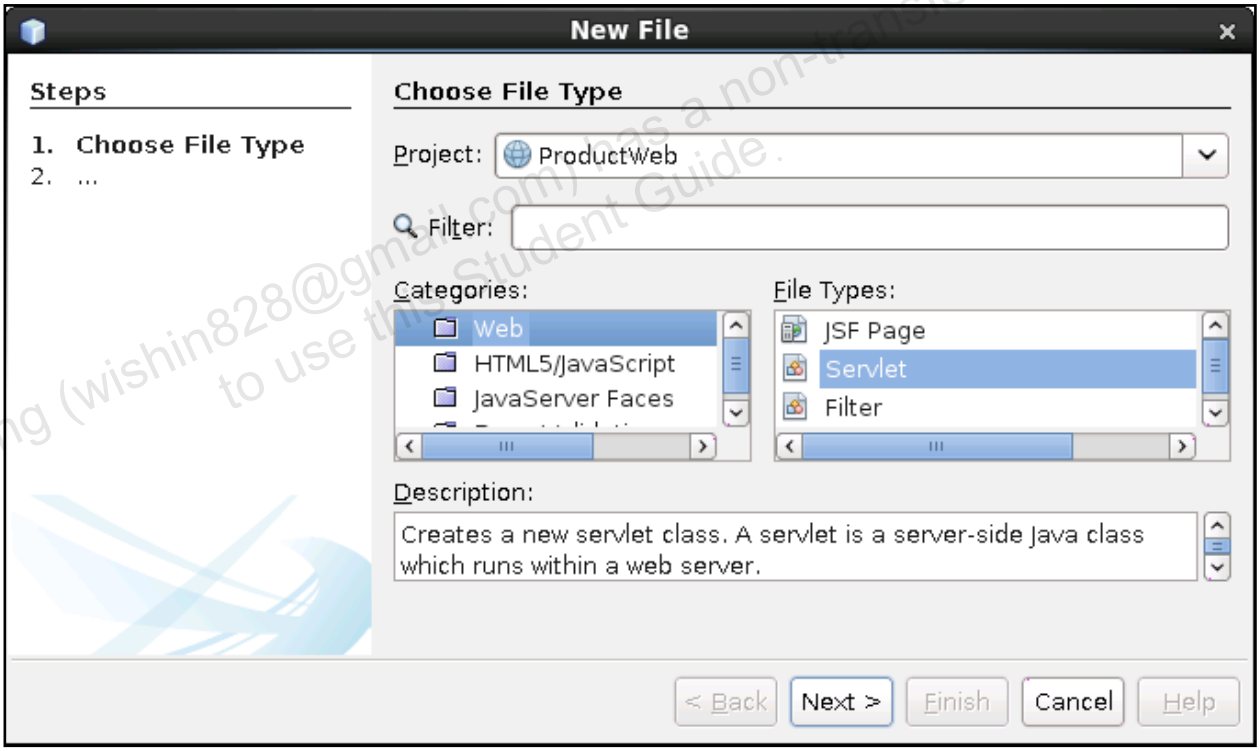
In this practice, you add a servlet to execute a product search and display the results of the search as a list of products.

Assumptions

You have successfully completed all previous practices.

Tasks

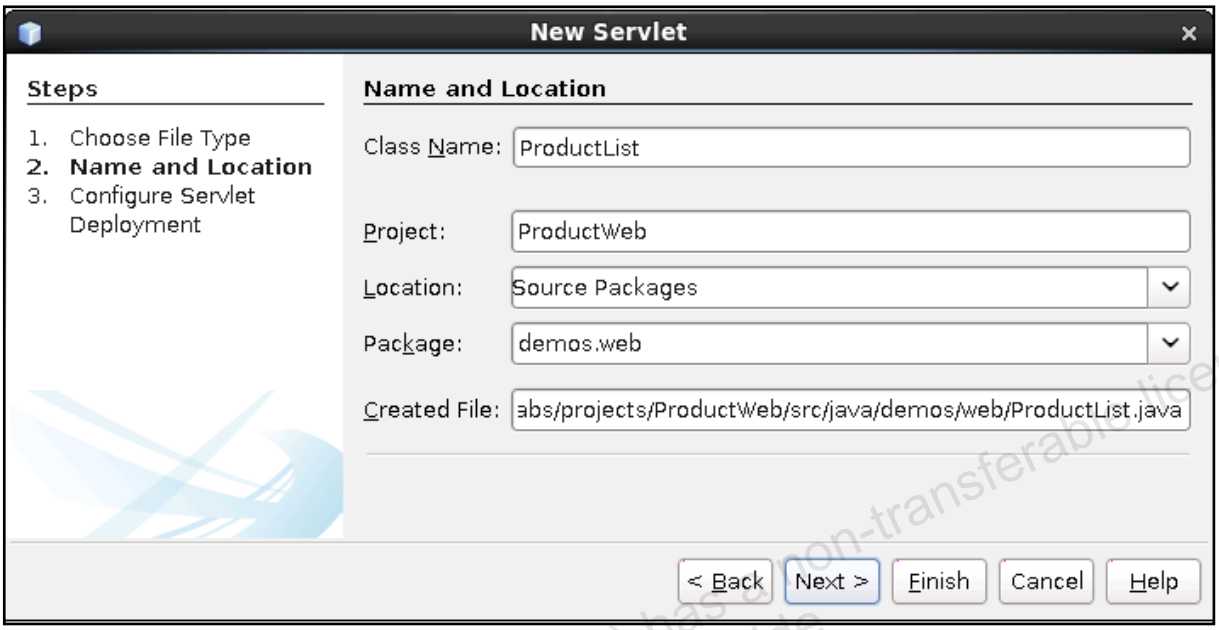
1. Create a new servlet:
   a. Select the **`ProductWeb`** project.
   b. Select **File > New File**.
   c. Select **Web** from the list of Categories and **Servlet** from the list of File Types.



   d. Click **Next**

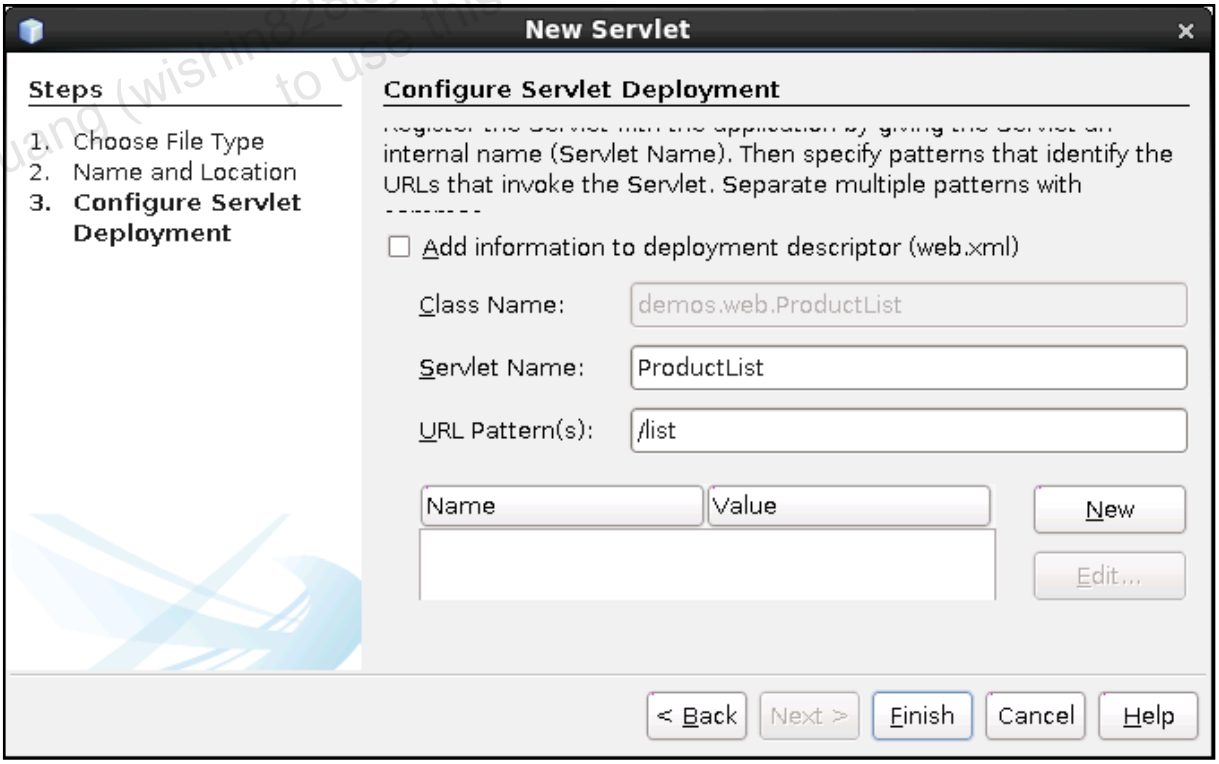
e. In a New Servlet dialog box, set the following properties:

- Class Name: `ProductList`
- Package Name: `demos.web`



f. Click **Next**.

g. Map the `ProductList` servlet to a URL by setting the **URL Pattern(s)** property to `/list`.



h. Click **Finish**.

2. Enable the `ProductList` servlet to handle parameters submitted by the form that you added to the `ProductSearch.html` file:
   a. Open the `ProductList` class from the `demos.web` package in the `ProductWeb` project.
   b. Inside the body of the `ProductList` class, after the line of code that contains the `ProductList` class definition, add code that injects a reference to the `ProductManager` CDI bean.
      ```
      @Inject
      private ProductManager pm;
      ```
   c. Add imports: `demos.model.ProductManager` and `javax.inject.Inject`
      Note: The `ProductManager` class in the `ProductWeb` project is marked with the `RequestScoped` annotation. This enables the CDI container to automatically present a new instance of the `ProductManager` bean for every request that this web application will have to handle.
   d. Inside a `try` block in the `processRequest` method, replace the line that contains a "TODO" comment with a new line of code that declares a `StringBuilder` object. Later you will add page content text to this `StringBuilder` object.
      ```
      StringBuilder content = new StringBuilder();
      ```
   e. After this line, add a new line of code that extracts a parameter called `p_name` from the request object.
      ```
      String name = request.getParameter("p_name");
      ```
      Note: This parameter contains a value of the input text item called p_name from the ProductSearch page
   f. Use the `ProductManager` CDI bean to find products that match the name value supplied via the `p_name` parameter.
      ```
      List<Product> products = pm.findProductByName(name);
      ```
   g. Add imports: `java.util.List` and `demos.db.Product` classes
   h. Check if the list of products is not empty. If there are some products within the list, you should append an HTML divider to the `StringBuilder` content and place product information inside it. Otherwise, you should append a divider containing an error message indicating that no products matching this name were found. Use CSS classes data and `error` to apply styles to these two dividers.

```
if (!products.isEmpty()) {
   products.stream().forEach
   (p -> content.append("<div class='data'>" + p + "</div>"));
} else {
   content.append("<div class='error'>");
   content.append("Unable to find any products matching name '" +
   name +"'</div>");
}
```

3. Replace the output produced by this servlet with template code:

   a. Remove all `out.println(...);` statements from the `processRequest` method.

   b. In place of the removed code, add a block of code from the `servlet_template.txt` file contained in the `/home/oracle/labs/resources` folder.

```
out.println("<!DOCTYPE html>");
out.println("<html>");
out.println("<head>");
out.println("<meta charset='UTF-8'>");
out.println("<meta name='viewport' "
            +"content='width=device-width, initial-scale=1.0'>");
out.println("<link rel='stylesheet' "
            +"type='text/css' href='css/pm.css'>");
out.println("<title>PAGE TITLE</title>");
out.println("</head>");
out.println("<body>");
out.println("<header class='header'>PAGE HEADER</header>");
out.println("<nav class='nav'>PAGE NAVIGATION</nav>");
out.println("<section class='content'>");
out.println("PAGE CONTENT");
out.println("</section>");
out.println("<footer class='footer'>PAGE FOOTER</footer>");
out.println("</body>");
out.println("</html>");
```

   Note: String concatenation is only present in this example because the code did not fit the line width of this file.

   Note: This block of code contains the same HTML as the `template.html` file, just wrapped into the `println` statements.

c. Change the `ProductList` servlet output to produce page title, header, navigation, content, and footer elements.
   - Replace the text PAGE TITLE with: `Product List`
   - Replace the text PAGE HEADER with: `Products`
   - Replace the text PAGE NAVIGATION with the link back to the `ProductSearch.html` page:
     ```
     <a href='/pm/ProductSearch.html'>Back to product search</a>
     ```
   - Replace the text PAGE CONTENT with the printout of the content object. The resulting code should look like this:
     ```
     out.println(content);
     ```
   - Replace the text PAGE FOOTER so that the footer prints out the HTTP method that was used to invoke this servlet. The resulting footer element should contain this code:
     ```
     out.println("<footer class='footer'>Invoker used method " +
     request.getMethod() + "</footer>");
     ```

4. Compile the **`ProductWeb`** project using **Clean and Build**.

5. Test the `ProductWeb` project:
   a. Right-click the **`ProductWeb`** project and select **Deploy**.
   b. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm`

      Note: The `index.html` page would be displayed, because it is considered to be a default welcome page in Java web modules. If you want to designate another page as your welcome page, you would have to modify `web.xml` file and include a `welcome-file-list` element to reference alternative pages.
   c. Click the **Product Search** link to navigate from `index.html` to the `ProductSearch.html` page.
   d. Into the Product Name field, type `Co%` and click the **Find** button.
   e. Observe a list of products matching the provided product name, displayed by the `ProductList` servlet mapped to the list URL. Also look at the footer of the product list page to see what was invoked via the `POST` method.
   f. Click the **ProductSearch** link to navigate back to the ProductSearch page.
   g. Type any nonexistent product name, for example `acme` and click the **Find** button. Observe an error message displayed by the product list page.
   h. In the browser address bar, append `C%` to the `p_name` parameter at the end of the URL. It should look like this: `http://localhost:7001/pm/list?p_name=C%` Then press **Enter**.

i. Observe a list of products and the page footer showing that the page was invoked via method `GET`.

j. Click the **ProductSearch** link to navigate back to the ProductSearch page.

k. Do not enter any product value and click the **Find** button. Observe an error message.

Note: The error message you have just observed is produced by the browser interpreting the 'required' attribute of the input field. However, a direct invocation of the `ProductList` servlet via the method `GET` would have bypassed this check.

6. Modify the `ProductList` servlet to refuse to serve the list of products if this servlet is invoked via the HTTP method `GET`:

a. Return to the **`ProductWeb`** project in NetBeans.

b. Open the **`ProductList`** class.

c. Expand the block of code at the bottom of this file.

Note: In this section of code you will find methods `doGet` and `doPost`. The purpose of these methods is to contain any code that you want this servlet to execute only if it is invoked via a particular HTTP method. To make the servlet respond only to a given HTTP method (for example, `POST`) you can simply use operation such as `doPost` instead of `processRequest`. However, what you will do next would be to write code to produce different handling for the `GET` and `POST` invocations of this servlet.

d. Locate the `doGet` method.

e. Replace the line of code that invokes the `processRequest` operation, with the code that forwards the request back to the ProductSearch page by using the `RequestDispatcher` object.

```
RequestDispatcher rd =
request.getRequestDispatcher("ProductSearch.html");
rd.forward(request, response);
```

Note: As a result, `ProductList` servlet will not attempt to query and display the product list if it is invoked with the `GET` method. Instead it will send a `ProductSearch.html` page back to the caller.

f. Add an import: `javax.servlet.RequestDispatcher`

7. Test the `ProductWeb` project again:
   a. Compile the **`ProductWeb`** project using **Clean and Build**.
   b. Right-click the **`ProductWeb`** project and select **Deploy**.
   c. After this project has been deployed, open the browser and navigate to the following URL:
      `http://localhost:7001/pm/list?p_name=Tea`
   d. Observe the `ProductSearch.html` page content returned from the server, and note that the actual URL still shows that you have called a list servlet.
   e. Now perform a different test: Invoke a nonexistent URL within the `ProductWeb` application. For example, enter this URL: `http://localhost:7001/pm/acme`

      Note: Because no servlets or pages are mapped to this URL, the server returns an HTTP status code 404, indicating that the requested resource is not found.
      Your next assignment will be to add error-handling capabilities to the `ProductWeb` application to produce custom error pages.

Practice 7-3: Creating an Error-Handling Servlet

Overview

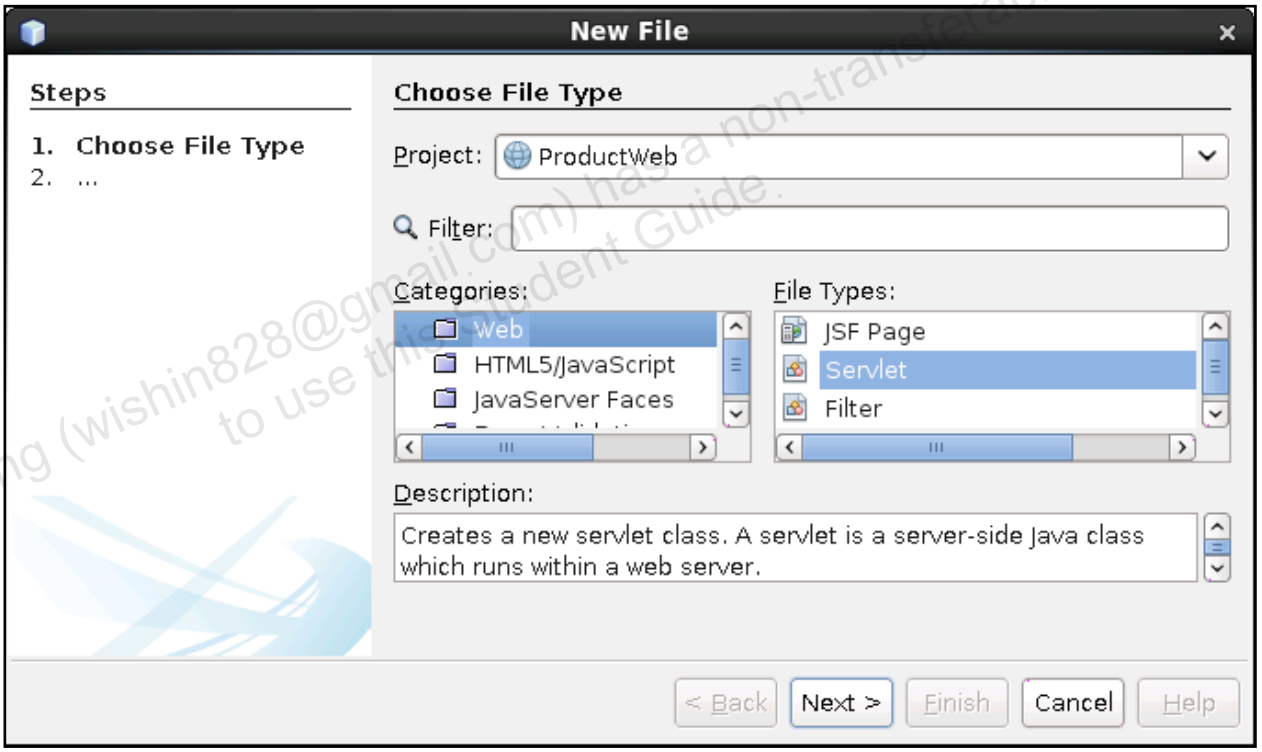
In this practice, you add an error-handling servlet to the `ProductWeb` application.

Assumptions

You have successfully completed all previous practices.

Tasks

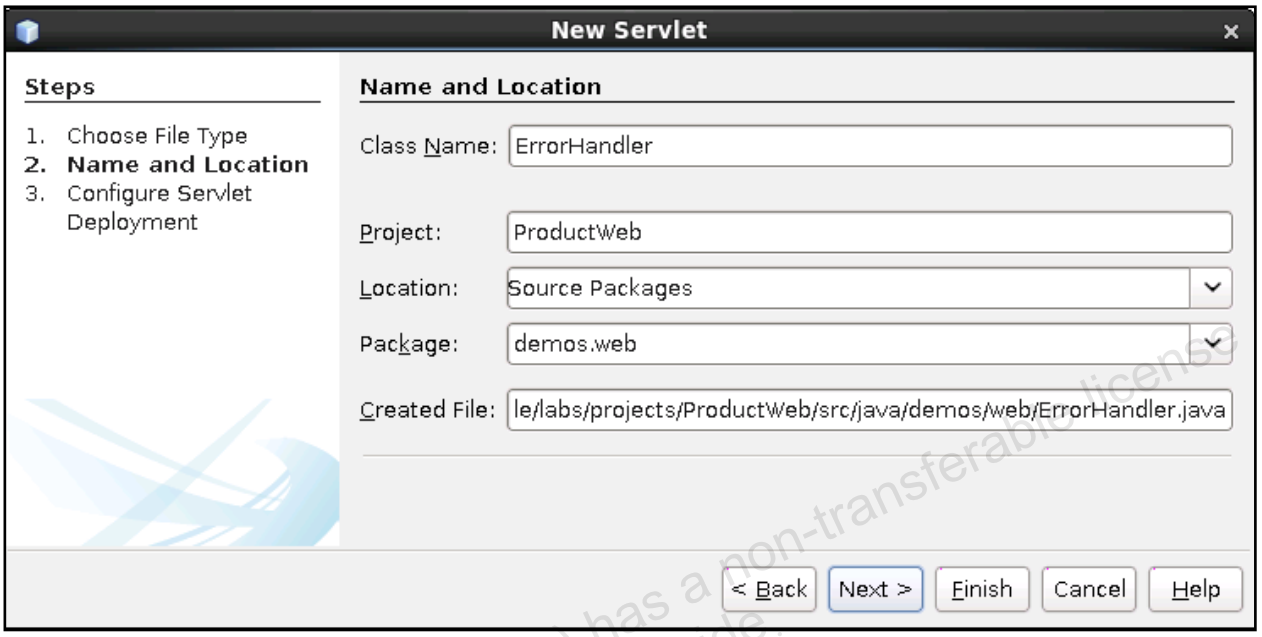
1. Create an error-handling servlet:
   a. Select the **`ProductWeb`** project in NetBeans.
   b. Select **File > New File**.
   c. Select **Web** from the list of Categories and **Servlet** from the list of File Types.



   d. Click **Next**

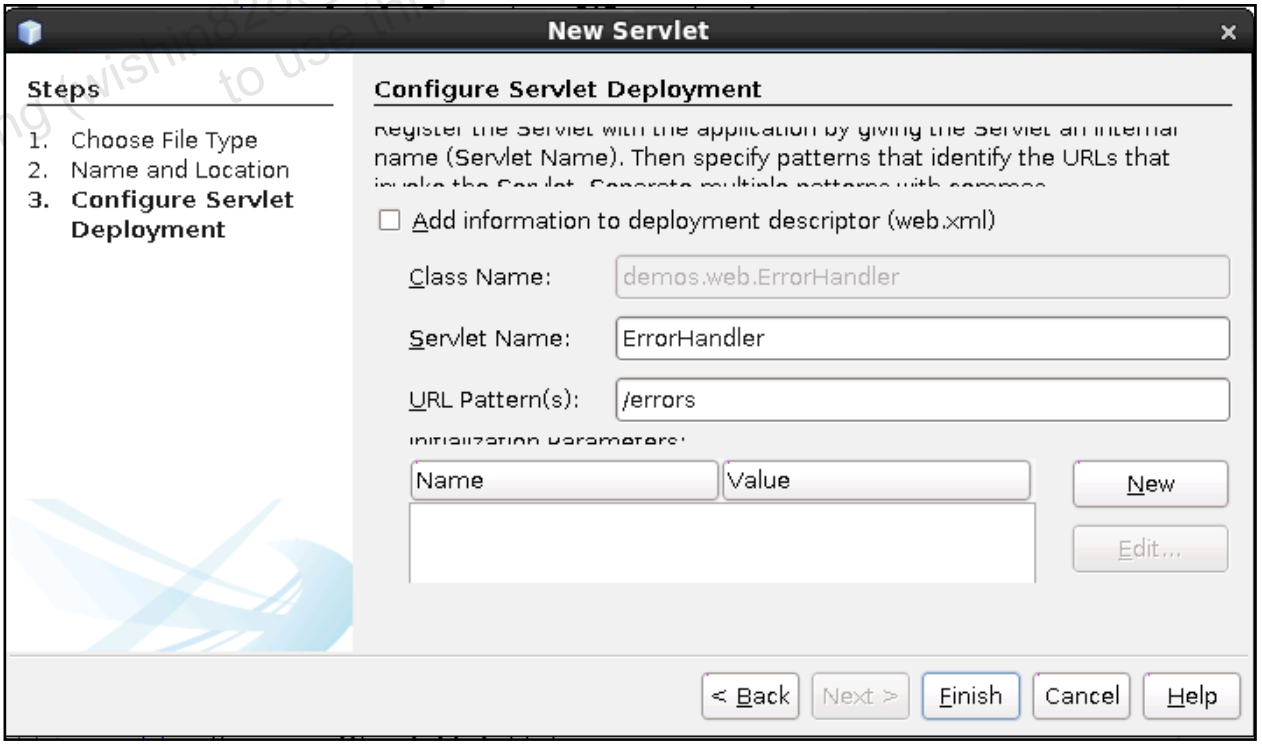
e. In the New Servlet dialog box, set the following properties:

- Class Name: `ErrorHandler`
- Package Name: `demos.web`



f. Click **Next**.

g. Map the `ErrorHandler` servlet to a URL by setting a **URL Pattern(s)** property to `/errors`.



h. Click **Finish**.

2. Add code to the `ErrorHandler` servlet to retrieve information about the error from the request object:
   a. Open the **`ErrorHandler`** servlet.
   b. Inside a `try` block in the `processRequest` method, replace the line that contains a "TODO" comment with a new line of code that gets a `Throwable` object from the request.

```
Throwable ex = (Throwable)
request.getAttribute("javax.servlet.error.exception");
```

   c. After this, add another line of code that gets an error message from the request.

```
String errorMessage =
(String)request.getAttribute("javax.servlet.error.message");
```

   d. After this, add another line of code that gets an HTTP status code from the request.

```
Integer status = (Integer)
request.getAttribute("javax.servlet.error.status_code");
```

   e. After this, add another line of code that prepares a String object that will contain a future error message.

```
String message = "Error: ";
```

3. Analyze the HTTP status code and produce different error-handling for the codes 404 (not found) and code 500 (internal server error):
   a. On a new line of code after the declaration of the `String` message variable, add a switch statement that looks at the status code and two cases to handle 404 and 500 error status codes. The 500 error indicates a server-side problem; therefore, it should be written into the log as well as reposted back to the user. The 404 error may simply indicate a user attempting to access a nonexistent URL; therefore, a simple error message returned to the user may be considered to be sufficient to handle this error.

```
switch (status) {
     case HttpServletResponse.SC_INTERNAL_SERVER_ERROR:
          request.getServletContext().log(errorMessage,ex);
          message += errorMessage;
          message += ". - Please contact server administrator.";
          break;
     case HttpServletResponse.SC_NOT_FOUND:
          message +="Requested page not found";
          break;
}
```

   Note: It could be more convenient to use constants for all HTTP status code located in the `HttpServletRespose` class instead of just numbers such as 404 or 500 to represent repose status codes.

4. Replace the output produced by this servlet with template code:
   a. Remove all `out.println(...);` statements from the `processRequest` method.
   b. In place of the removed code, add a block of code from the `servlet_template.txt` file contained in the `/home/oracle/labs/resources` folder.
   c. Change the `ErrorHandler` servlet output to produce page title, header, navigation, content, and footer elements.
      - Replace the text PAGE TITLE with: `Error Page`
      - Replace the text PAGE HEADER with: `Errors`
      - Replace the text PAGE NAVIGATION with the link back to the `index.html` page:
        `<a href='/pm'>Home</a>`
      - Replace the text PAGE CONTENT with the printout of the error message. The resulting code should look like this:
        `out.println("<div class='error'>"+message+"</div>");`
      - Replace the text PAGE FOOTER with the following text:
        `Click on the Home page link to start over`

5. Register the `ErrorHander` servlet as an error-handling resource in the `web.xml` file:
   a. Expand **WebPages > WEB-INF** node.
   b. Open the **web.xml** file.
   c. After the end of the `session-config` element, add two error-page elements to designate the `ErrorHandler` servlet mapped to the `/errors` URL as an error handler for error codes 404 and 500.

```
<error-page>
    <error-code>404</error-code>
    <location>/errors</location>
</error-page>
<error-page>
    <error-code>500</error-code>
    <location>/errors</location>
</error-page>
```

6. Add code to the `ProductList` servlet to artificially produce an internal server error, just to be able to test the `ErrorHandler` servlet capabilities.
   a. Open the **`ProductList`** servlet and locate the `doGet` operation.
   b. At the beginning of this operation, just before the line of code that creates a `RequestDispatcher` object, place code that can conditionally raise `ServletException`. Use a parameter to indicate that you want to produce an exception.

```
String error = request.getParameter("error");
if (error != null) {
   throw new ServletException("Test Servlet Error");
}
```

7. Test the `ProductWeb` project:
   a. Compile the **`ProductWeb`** project using **Clean and Build**.
   b. Right-click the **`ProductWeb`** project and select **Deploy**.
   c. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm/list?error`
   d. Observe a custom error page displaying full details of the Test Servlet Error that you just produced.
   e. Now perform a different test: Invoke a nonexistent URL within the `ProductWeb` application. For example enter this URL: `http://localhost:7001/pm/acme`
   f. You may now observe the same `ErrorHandler` servlet producing a custom error page for you.

Note: The rest of the functionality of this application should be the same, as tested in Practice 7-2.

Practices for Lesson 8: Creating Java Web Applications by Using Java Server Pages

Practices for Lesson 8: Overview

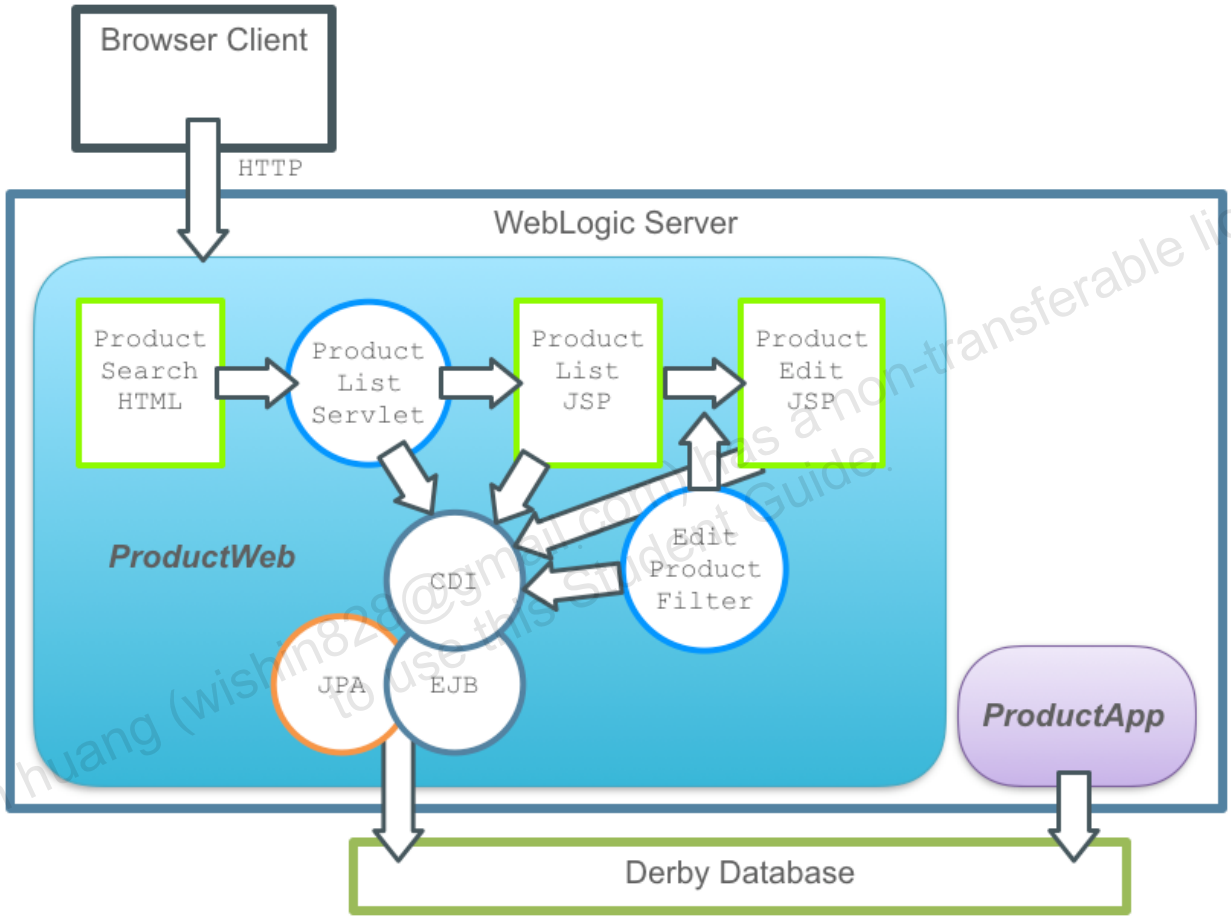
Overview

In these practices, you create two Java Server Pages to display a list of products and to update a product. You use a `ProductManager` CDI bean to maintain application state and reference values stored in this bean from JSPs using EL 3.0 expressions. You use two different mechanisms, servlet and `WebFilter` to handle interactions with these JSP pages.

You will create `ProductList` and `ProductEdit` JSP pages. In the previous practice you mixed UI generation and application logic handling inside the same class `ProductList` servlet. In this practice, your will structure your code according to the Model-View-Controller pattern.

You will modify an existing `ProductList` servlet and remove all UI generation code from it. Instead this servlet will perform all application logic handling actions related to querying of the list of products, store information inside the `ProductManager` CDI bean, and forward requests to the `ProductList` JSP page. The JSP page will then be able to get all of its values from the `ProductManager` CDI bean and produce a UI.

For the `ProductEdit` JSP page you will use a different Controller approach: `WebFilter`. Just as `ProductList` servlet did for a `ProductList` JSP, EditProductFilter will handle application logic on behalf of the `ProductEdit` JSP and store required information into the `ProductManager` CDI bean. However, unlike the servlet scenario where your HTML form submitted a request to a servlet, with `WebFilter` you will invoke a target page (in this case, `ProductEdit` JSP) directly and Filter will intercept all calls to this page to perform its tasks.



Practice 8-1: Creating a JSP to Display the Product List

Overview

In this practice, you add code to the `ProductManager` CDI bean that will be used to hold application values used by Java Server Pages. You modify the code of `ProductServlet` to forward a request to this new JSP that displays the list of products instead of generating output itself.

Assumptions

You have successfully completed all previous practices.

Tasks

1. Prepare the `ProductManager` CDI bean to support Java Server Pages:

   Note: In accordance with the Model-View-Controller design pattern, you should avoid writing direct Java code implementing application logic into Java Server Pages. Java code should be placed into servlets or Web Filters, which can handle client requests and trigger application logic execution. Application state can be maintained by CDI bean classes such as `ProductManager`. Java Server Page should use expression language to reference this code. EL 3.0 expressions expect your CDI beans to maintain application state, so they can access it using getter and setter method patterns.

   a. Expand **Source Packages > demos.model** node in the **ProductWeb** project.

   b. Open the **`ProductManager`** class.

   c. Add a new line of code after the `@RequestScoped` annotation and place there a Named annotation to allow this bean to be referenced from EL 3.0 expressions.

   ```
   @Named("pm")
   ```

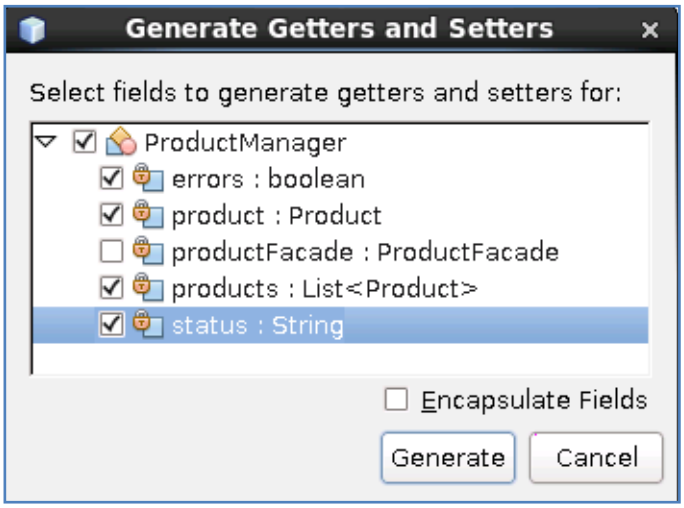
   d. Add an import: `javax.inject.Named`

2. Add instance variables and access methods to the `ProductManager` bean to hold values of a List of Products, a specific Product, a String with a status message, and a Boolean errors indicator.

   a. Inside the body of the `ProductManager` class, after the line of code that declared a `ProductFacade` instance variable, add four new instance variable declarations.

   ```
   private List<Product> products;
   private Product product;
   private String status;
   private boolean errors;
   ```

b. Create get and set operations for these instance fields. A quick way of generating getter and setter operations for a number of variables is to create a new line of code. Just before the end of this class, right-click this line and select **Insert Code > Getter and Setter**. Then in the Generate Getters and Setters dialog box, select check boxes for the four variables you have just added.



c. Click the **Generate** button.

3. Modify the `ProductList` servlet so it will handle product query logic, but will not produce any output of its own:
   a. Open **`ProductList`** located in the **demos.web** package.
   b. Inside a `processRequest` method, remove (or place comments on) all code except the part that extracts parameter, calls `findProductByName` operation, and checks if there are any results returned within the `if` / `else` block.

      This is the code you should keep:

```
String name = request.getParameter("p_name");
List<Product> products = pm.findProductByName(name);
if (!products.isEmpty()) {

} else {

}
```

   c. Inside the `if` block, add code to store the products list into the `ProductManager` bean.

```
pm.setProducts(products);
```

   d. Inside the `else` block, set errors and status variables in the `ProductManager` to indicate an error that no products were found.

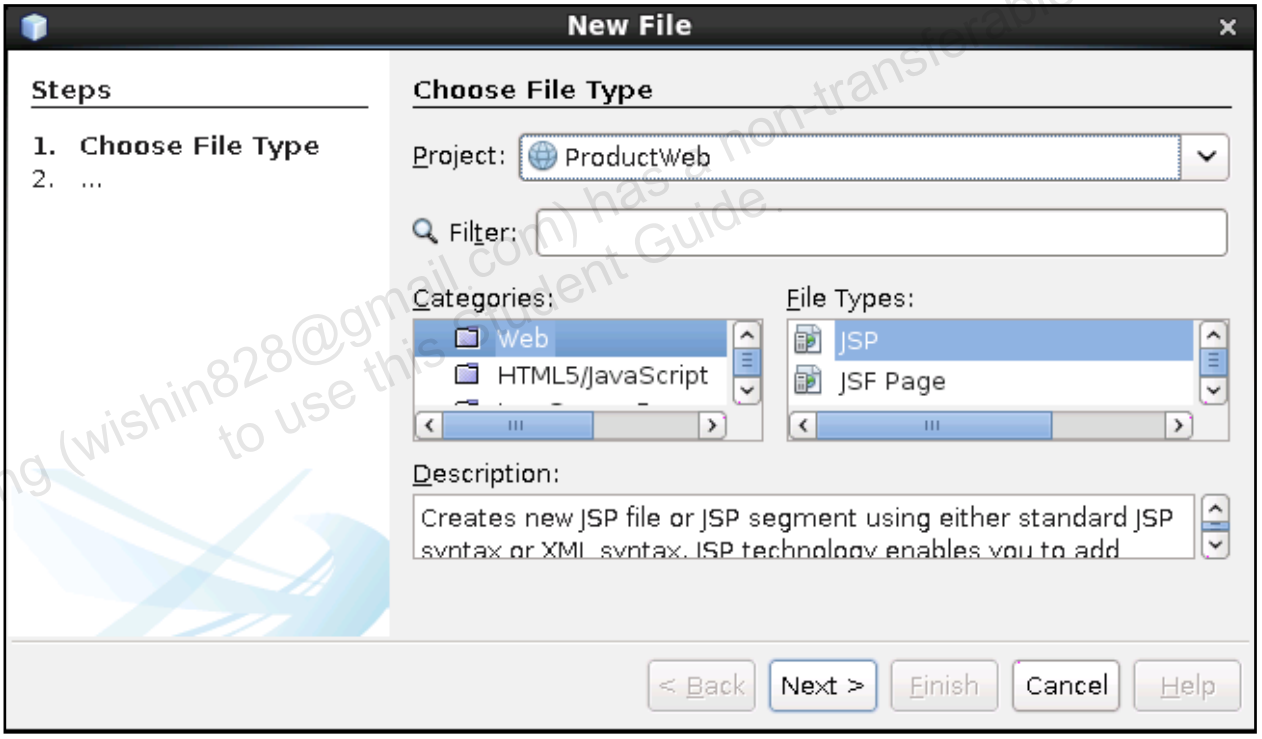
```
pm.setErrors(true);
pm.setStatus("Unable to find any products matching the name '"
+name+"'");
```

e. After the end of the `else` block, place code inside the `processRequest` method to forward request to the `ProductList.jsp` page by using `RequestDispatcher`.

```
RequestDispatcher rd =
request.getRequestDispatcher("ProductList.jsp");
rd.forward(request, response);
```

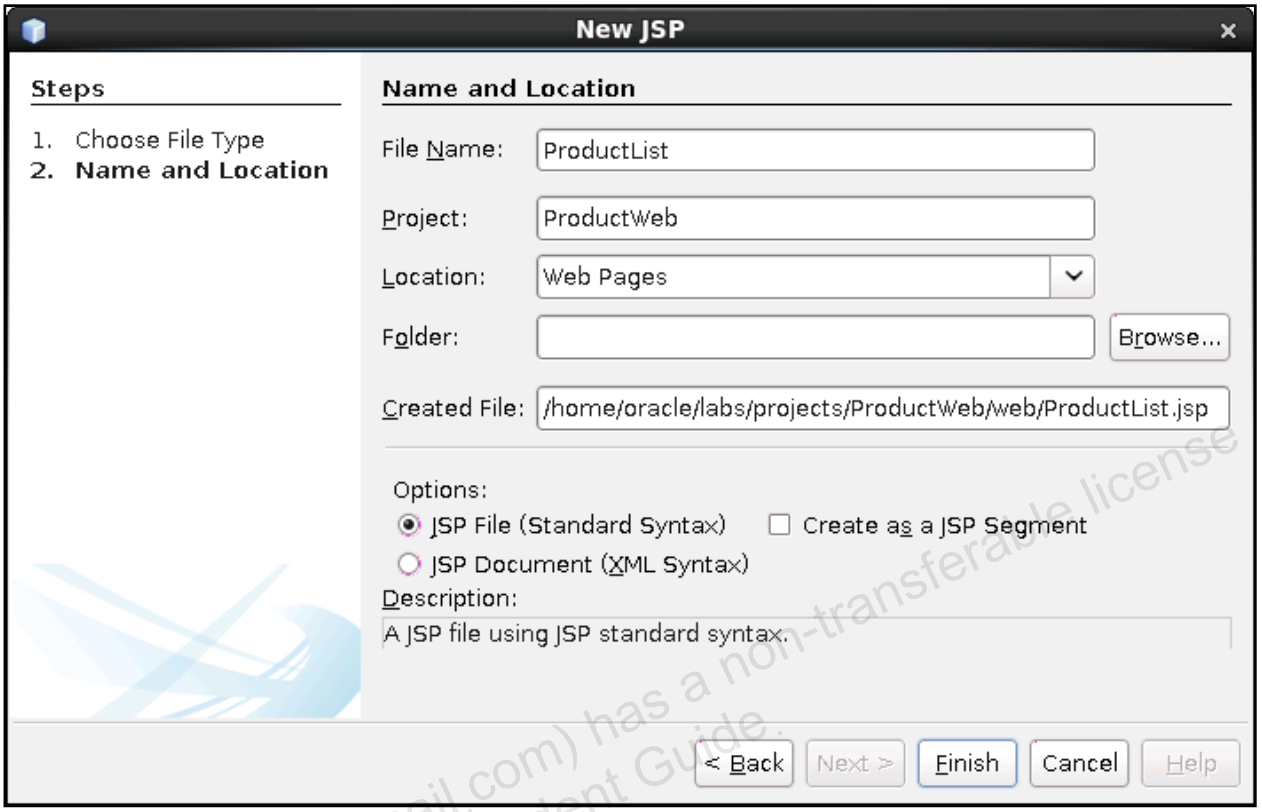
Note: You will create the `ProductList.jsp` page in the step of this practice.

4. Create a new Java Server Page to display a list of products:
   a. Select the **ProductWeb** project.
   b. Select **File > New File**.
   c. Select **Web** from the list of Categories and **JSP** from the list of File Types.



   d. Click **Next**.

e. In the New JSP dialog box, set the **File Name** property as: `ProductList`



f. Click **Finish**.

5. Modify the `ProductList.jsp` file to define the page structure:

   a. Replace content starting from the `<!DOCTYPE html>` element of the `ProductList.jsp` file with text from the `template.html` file.

   The file is located in the `/home/oracle/labs/resources` folder.

   Keep the first line of code:

   ```
   <%@page contentType="text/html" pageEncoding="UTF-8"%>
   ```

   Replace the rest with the text from template.

   b. Change the `ProductList.jsp` page's title, header, navigation, and footer elements.

   - Replace the text PAGE TITLE with: `Product List`
   - Replace the text PAGE HEADER with: `Products`
   - Replace the text PAGE NAVIGATION with the link back to the `ProductSearch.html` page:

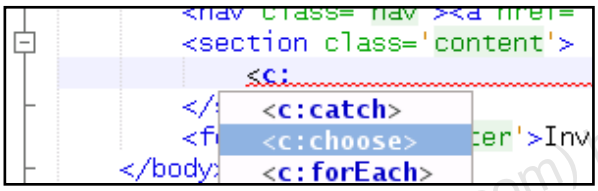
     ```
     <a href='/pm/ProductSearch.html'>Product Search</a>
     ```

   - Replace the text PAGE FOOTER so that the footer prints out the HTTP method that was used to invoke this servlet. The resulting footer element should contain this code: `Invoker used method ${pageContext.request.method}`

Note: Java Server Pages may get access to their environment by using the `pageContext` object from EL 3.0 expressions. In this case, you are using `pageContext` to get access to the request object and get the value of the HTTP method that was used to invoke this page.

6. Modify the `ProductList.jsp` page to access the `ProductManager` CDI bean and request a list of products from it:
   a. Replace the text PAGE CONTENT inside the `ProductList.jsp` with a condition that checks if your search operation did manage to find at least one product. This is done by adding JSTL **choose / when / otherwise** elements. The test condition should reference **errors** valuable of the `ProductManager` CDI bean to check that there were no errors.

      Note: When you start typing the `<c:` prefix, a drop-down list of elements will be displayed. Select the `choose` element from it and press Enter. This will not only add the element to a page but also automatically add a tag lib declaration to the top of this page.




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

This is the code you should produce:

```
<c:choose>
    <c:when test="${!pm.errors}">
</c:when>
     <c:otherwise>
     </c:otherwise>
</c:choose>
```

Note: Remember, you have used a `@Named("pm")` annotation in the `ProductManager` class.

b. Inside a **when** element, add code that iterates through the list of products and displays each product. Use a JSTL **forEach** element to iterate through the list of products. Assign each product to a variable called **p** and output **id, name, price**, and **bestBefore** elements.

```
<c:forEach items="${pm.products}" var="p">
    <div class='data'>
      ${p.id}
      ${p.name}
      ${p.price}
      ${p.bestBefore}
    </div>
</c:forEach>
```

c. Inside the **otherwise** element, display a status message stored in the `ProductManager` bean.

```
<div class='error'>${pm.status}</div>
```

Note: The `ProductManager` CDI bean has been instantiated for you by the CDI container, and this bean instance is hiding a list of products placed there by the `ProductList` servlet. Remember that this bean had a `@RequestScoped` annotation, which makes its instance specific to each new request. Another caller accessing the same bean at the same time is going to get a separate instance of this bean, containing a different list of products. Error, status, product, and other information will not be retained after the JSP page renders the output and the request-repose cycle completes.

7. Test the ProductWeb project:
   a. Compile the **ProductWeb** project using **Clean and Build**.
   b. Right-click the **ProductWeb** project and select **Deploy**.
   c. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm`
   d. Click the **Product Search** link to navigate from the `index.html` to the `ProductSearch.html` page.
   e. In the Product Name field, enter `Co%` and click the **Find** button.
   f. Observe the following details:
      - A list of products matching the provided product name was prepared by `ProductServlet` and displayed by the `ProductList.jsp`.
      - Check the page footer of the product list page to see what was invoked via the `Post` method.
      - The URL in the address bar of the browser does not show the name of the JSP, but a servlet-mapped URL instead. This is because the request was submitted to the `ProductList` servlet (mapped to the URL 'list'), and then this servlet forwarded the request to the JSP page to render the response to the client.

Therefore, the client browser is only interacting with the `ProductList` servlet and is unaware of the `ProductList.jsp` page's existence.

g. Click the **Product Search** link to navigate back to the `ProductSearch.html` page.

h. In the Product Name field, enter the name of a nonexistent product (for example, `acme)` and click the **Find** button.

i. Observe the error status message displayed by the `ProductList` page.

Note: If you try to invoke `ProductList.jsp` directly, you will get an output with no products or messages of any kind, because the `ProductList` servlet was not used to handle this request and did not prepare values in the CDI bean to be used by the page.

Practice 8-2: Creating a JSP for Editing a Product

Overview

In this practice, you create a new Java Server Page to display an edit form for a product. To support this page you first have to create a `WebFilter` class that can manage page-interaction logic and provide Java exception handling on behalf of this JSP.

Assumptions

You have successfully completed all previous practices.

Tasks

1. Create a `WebFilter` Java class to handle query and update behaviors on behalf of the `ProductEdit.jsp` page. This filter should intercept all calls to the `ProductEdit.jsp` page, process all parameters on behalf of this page, execute query and update actions, maintain status information, and handle exceptions.

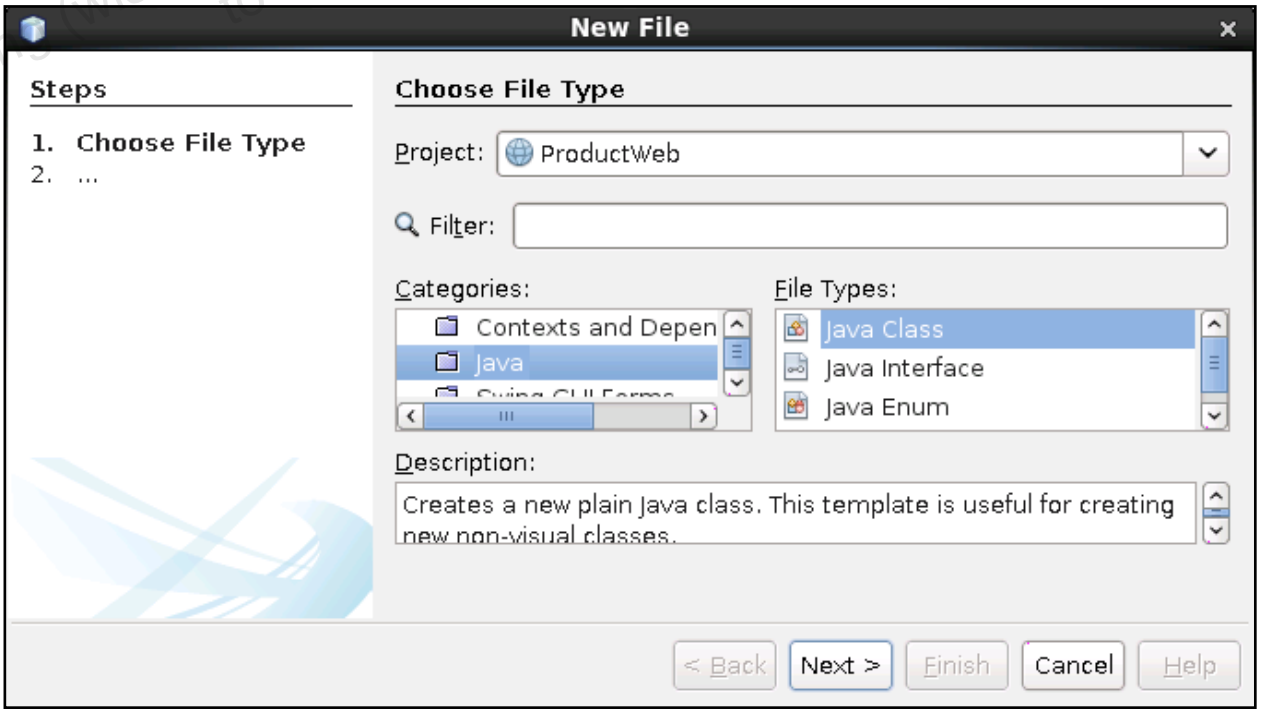
   Note: There is a wizard in NetBeans that can generate a `WebFilter` class for you, but for training purposes you will be instructed to create a normal Java class and then turn it into a `WebFilter` by implementing a Filter interface and annotating the class with a `WebFilter` annotation.

   Note: You will create `ProductEdit.jsp` in a later step of this practice.

   a. Select the **ProductWeb** project.

   b. Select **File > New File**.

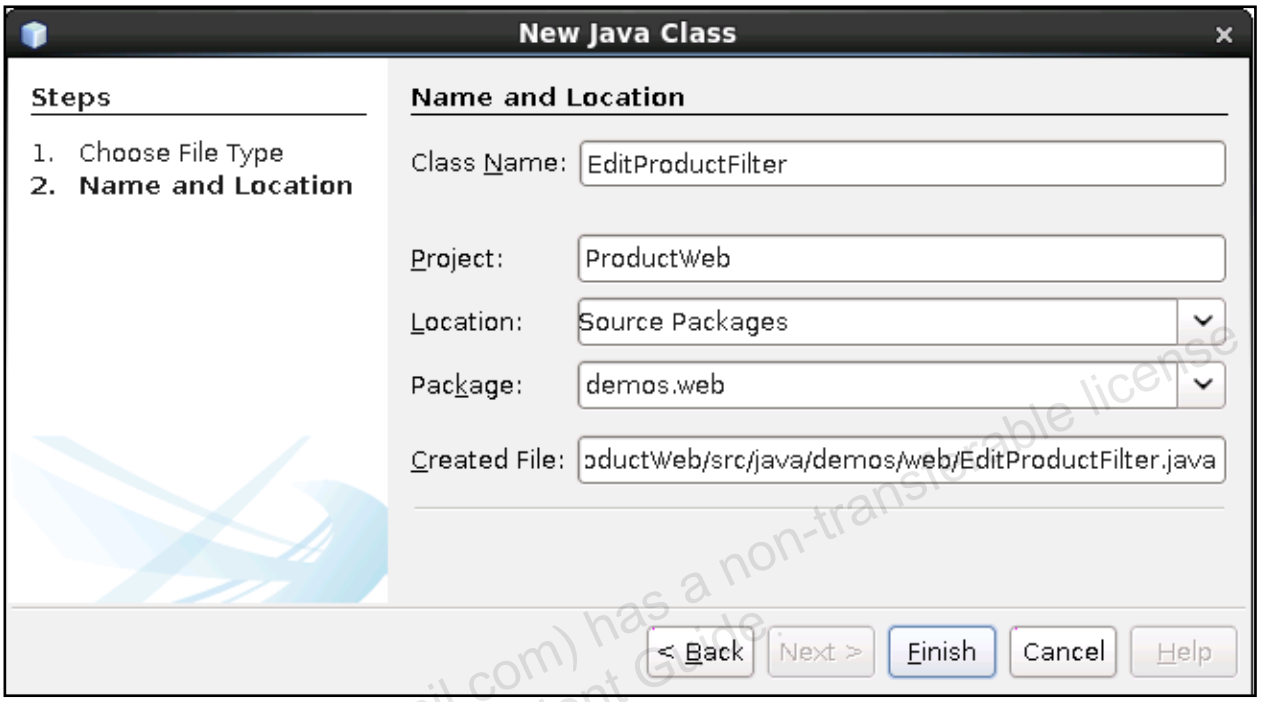
   c. Select **Java** from the list of Categories and **Java Class** from the list of File Types.



   d. Click **Next**.

e. In the New Java Class dialog box, set the following properties:

- Class Name: `EditProductFilter`
- Package: `demos.web`

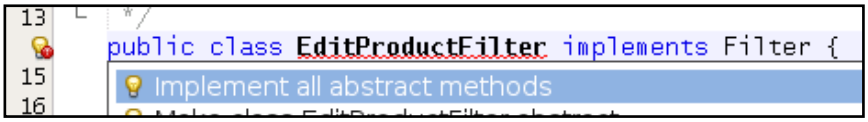


f. Click **Finish**.

g. Add an `implements Filter` interface clause to the `EditProductFilter` class definition.

h. Click the light-bulb icon on the left side of the class definition statement to add an import of `javax.servlet.Filter`.

i. Click the same light-bulb icon again and select **Implement all abstract methods.**



Note: Three operations were added to this class: `init`, `destroy`, and `doFilter`. You will add implementation code to these operations a little later. First, you should map this filter to intercept requests to the `ProductEdit.jsp` page.

j. Add a **`WebFilter`** annotation just above the line of code that contains the `EditProductFilter` class definition. This annotation should set two properties: **`filterName`** and **`urlPatterns`**.

```
@WebFilter(filterName = "EditProductFilter",
           urlPatterns = {"/ProductEdit.jsp"})
```

k. Add an import: `javax.servlet.annotation.WebFilter`

2. Add instance variables and initialization code to the `EditProductFilter` class:

a. Add in injection of the `ProductManager` CDI bean.

```
@Inject
private ProductManager pm;
```

b. Add imports: `demos.model.ProductManager` and `javax.inject.Inject`

c. Add an instance variable type of `FilterConfig`.

```
private FilterConfig config;
```

d. Inside the **`init`** method remove the line of code that throws an exception, and in its place add a line of code that assigns the `init` method parameter of the `FilterConfig` instance variable.

```
config = filterConfig;
```

e. Inside the **`destroy`** method, remove the line of code that throws an exception.

3. Implement the `doFilter` operation logic that extracts the product ID parameter and uses it to find the product object:

a. Inside the **`doFilter`** method, remove the line of code that throws an exception.

b. In place of this line of code, add a declaration of the `HttpServletResponse` object, which should be initialized as a type-cast reference of the `ServletRequest` parameter.

```
HttpServletRequest req = (HttpServletRequest)request;
```

c. Add an import: `javax.servlet.http.HttpServletRequest`

Note: Servlet filters are designed to be applied to any servlets and may theoretically be designed to use other protocols, not just http. This is why the `doFilter` operation refers to more generic types `ServletRequest` and `ServletResponse`. However, in this application you mapped `EditProductFilter` to the `ProductEdit.jsp` page, which is an http resource. Therefore, it is safe to perform type casting of the `ServletRequest` type to `HttpServletRequest` in this case.

d. On the next line of code after the type casting of the `ServletRequest`, add code to extract the product ID parameter from the `HttpServletRequest` object (the one you just declared). The product ID value would be of a String type because http protocol transmits all values as just text.

```
String pid = req.getParameter("p_id");
```

e. On the next line convert the String text value into Integer type.

```
Integer id = Integer.parseInt(pid);
```

f. Use the Integer product ID value to execute the `findProduct` operation upon the `ProductManager` class.

```
Product product = pm.findProduct(id);
```

g. Add an import: `demos.db.Product`

h. Check if the product object returned by the `findProduct` operation is not null. If the product has not been found, set the Boolean error indicator in the `ProductManager` to `true` and set the status message to inform user about this. If this product is found, apply this product object to the `ProductManager` object.

```
if (product == null) {
    pm.setErrors(true);
    pm.setStatus("Product with id '" + pid + "' not found");
 } else {
    pm.setProduct(product);
    // you will add more code here
}
```

Note: The comment line in the `else` block indicates a place where you will add more code in the next step of this practice.

4. Extract product update parameters submitted via the HTTP `POST` operation:

Note: When this filter has intercepted an HTTP `GET` operation, then there is really nothing else it should be doing—just query a product and let the `ProductEdit.jsp` page display it. The assumption is that the `ProductEdit.jsp` page is first invoked via the `GET` method just to display a product with a specific ID that you want to update. However, when you click the update button in the `ProductEdit.jsp` page, it submits another request, this time using the HTTP method `POST`, and `EditProductFilter` should perform a product update on this occasion.

a. Inside the `else` block, after the `setProduct` statement, add another `if` statement to check if this filter was invoked via HTTP method `POST`.

```
if (req.getMethod().equalsIgnoreCase("POST")) {
    // you will add more code handling product update here
}
```

b. Inside this `if` statement, add code that extracts product name, price, and date parameters as String objects.

```
String pname = req.getParameter("p_name");
String pprice = req.getParameter("p_price");
String pdate = req.getParameter("p_date");
```

Note: You will use these values to update the product. However, first you need to convert them from String types to other Java types. This conversion code as well as

the update action can produce exceptions. Therefore, your next step will be to create exception handling code in this method.

5. Add exception-handling logic to catch errors related to type conversions, value validations, optimistic locking, and other possible problems:
   a. Inside this `if` statement, on the next line of code after the parameter extraction, create a `try` block.

```
try {
// conversion and product update logic will be placed here
}
```

   b. Add a `catch` block to handle `NumberFormatException` and `DateTimeParseException` objects. Both of these blocks should set Boolean error indicators and status messages in the `ProductManager` bean.

```
 catch (NumberFormatException ex) {
     pm.setErrors(true);
     pm.setStatus("Price '"+pprice+"' is not a valid number");
} catch (DateTimeParseException ex) {
     pm.setErrors(true);
     pm.setStatus("Best before date '"
     +pdate+"' format should be: yyyy-MM-dd");
}
```

   c. Add an import: `java.time.format.DateTimeParseException`
   d. Add one more `catch` block that handles any other exceptions. Set an error indicator in the `ProductManagement` bean. You should also check the cause of the exception.

```
catch (Exception ex) {
     Throwable cause = ex.getCause();
     pm.setErrors(true);
// exception handling logic will be placed here
}
```

   e. Inside this last `catch` block, check if the cause of the exception was **`ConstraintViolationException`**, or **`OptimisticLockException`**, or **anything else**. In the last `else` block, simply throw an original exception.

```
if (cause instanceof ConstraintViolationException) {
// constraint violation handling logic will be placed here
} else if (cause instanceof OptimisticLockException) {
// optimistic lock exception handling logic will be placed here
} else {
     throw ex;
}
```

   f. Add imports: `javax.validation.ConstraintViolationException` and `javax.persistence.OptimisticLockException` classes

Note: Throwing an exception will interrupt the request handling cycle; therefore, in this case the filter will not pass the handling to the target page. However, you have already registered an exception handling servlet. An unhandled exception like the one you just produced will result in the HTTP code 500 and, therefore, will be intercepted by the `ErrorHandler` servlet.

g. Inside the `if` block that handles **`ConstraintViolationException`**, create a `StringBuilder` object to construct a status message.

```
StringBuilder status =
new StringBuilder("Error updating product: ");
```

h. Cast the exception to a constraint violation exception type.

```
ConstraintViolationException e =
(ConstraintViolationException)cause;
```

i. Construct a status message out of the error messages contained within the set of `ConstraintViolations` contained within this exception.

```
e.getConstraintViolations().stream()
      .forEach(v -> status.append(v.getMessage() + " "));
```

j. Apply the status to the `ProductManager` bean.

```
pm.setStatus(status.toString());
```

k. Inside the `if` block that handles **`OptimisticLockException`**, simply set the status message to the `ProductManager` bean indicating that records has been changed by another user.

```
pm.setStatus("Product has been changed by another user.");
```

6. Inside the `try` block, add logic to handle value type conversions, apply changed values to the product object, and update this product:

a. Product name is a String, so does not require any conversion. You should just assign it to the product.

```
product.setName(pname);
```

b. Price needs to be converted from String to BigDecimal before applying it to the product object.

```
product.setPrice(new BigDecimal(pprice));
```

c. Add an import: `java.math.BigDecimal`

d. `BestBefore` date should be converted to `LocalDate`. However, it is possible that `BestBefore` date could be missing entirely, because it is not a mandatory attribute. Do not perform date conversion if the date value is not set.

```
if (pdate.length() != 0) {
      DateTimeFormatter fmt =
      DateTimeFormatter.ofPattern("yyyy-MM-dd");
      product.setBestBefore(LocalDate.parse(pdate, fmt));
} else {
      product.setBestBefore(null);
}
```

e. Add imports of the following classes:

- `java.time.LocalDate`
- `java.time.format.DateTimeFormatter`

f. Pass the modified product object to the update operation of the `ProductManager` bean. If update is successful, set a status message to inform the user.

```
pm.update(product);
pm.setStatus("Product updated successfully");
```

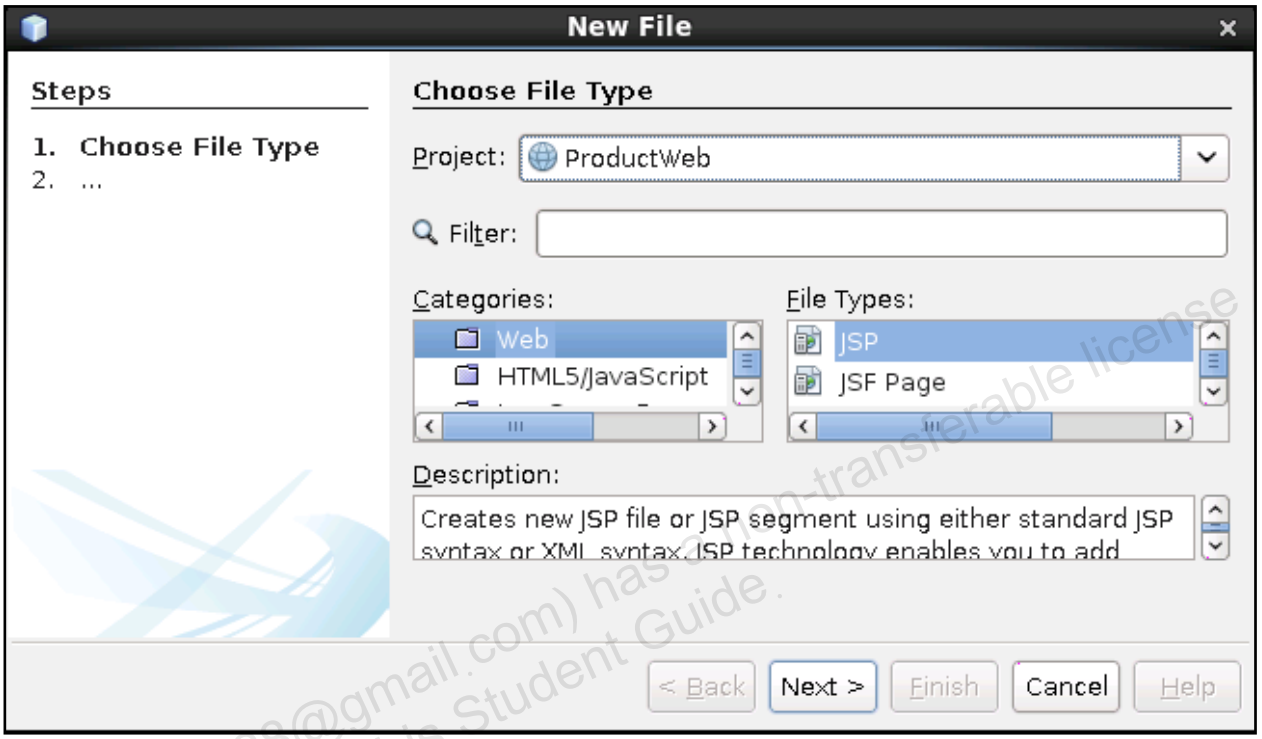
g. At the very end of the `doFilter` method just before the method closer, add a new line of code that allows this filter to send request processing down the filter chain.

```
chain.doFilter(request, response);
```

Note: If there are no other filters registered for this URL, request handing will be transferred to the target page—in this case, `ProductEdit.jsp`.

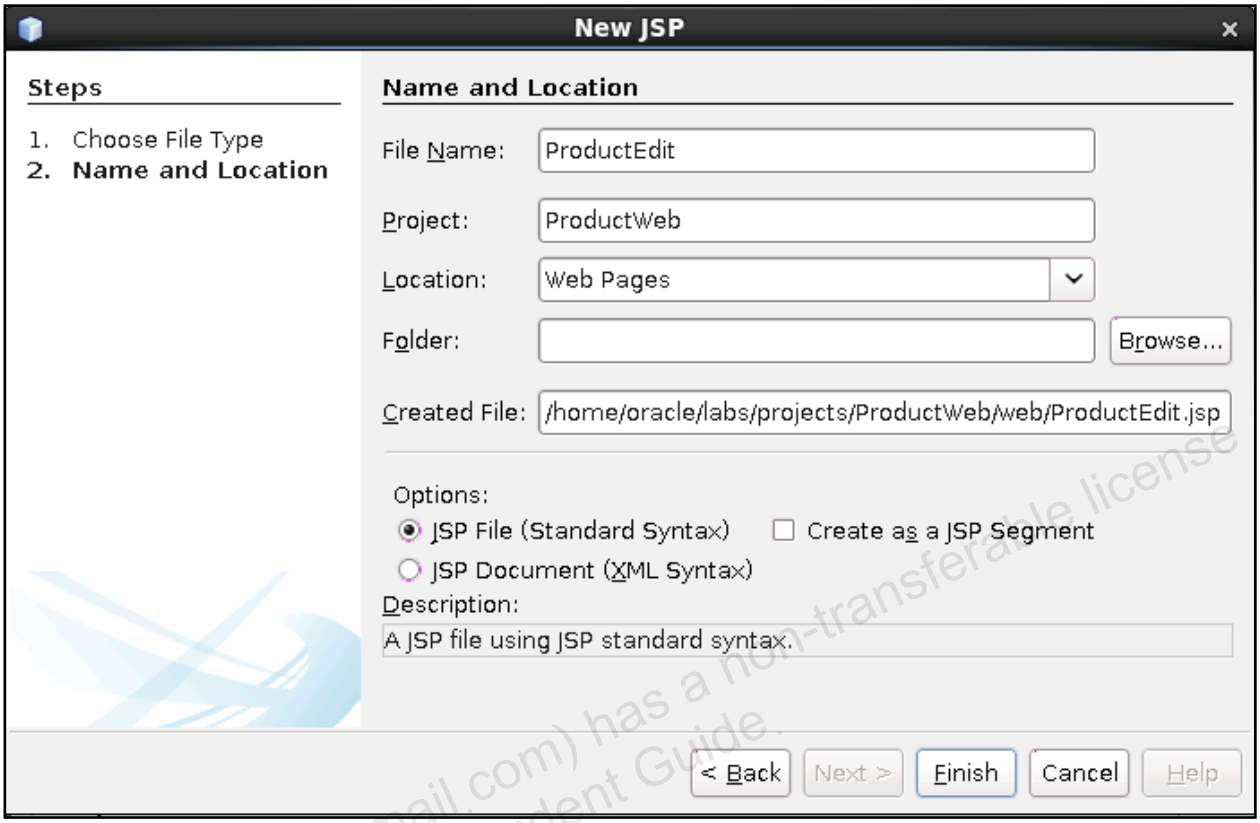
7. Compile the **ProductWeb** project using **Clean and Build**.

8. Create a new **Java Server Page** to edit a product.
   a. Select the **ProductWeb** project.
   b. Select **File > New File**.
   c. Select **Web** from the list of Categories and **JSP** from the list of File Types.



   d. Click **Next**.

e. In the New JSP dialog box, set File Name property to: `ProductEdit`



f. Click **Finish**.

9. Modify the `ProductEdit.jsp` file to define the page structure.

   a. Replace the content staring from the `<!DOCTYPE html>` element of the `ProductEdit.jsp` file with text from `template.html`.

   The file is located in the `/home/oracle/labs/resources` folder.

   Keep the first line of code:

   ```
   <%@page contentType="text/html" pageEncoding="UTF-8"%>
   ```

   Replace the rest with the text from template.

   b. Change the `ProductEdit.jsp` page's title, header, navigation, and footer elements.

   - Replace the text PAGE TITLE with: `Edit Product`
   - Replace the text PAGE HEADER with: `Edit Product`
   - Replace the text PAGE NAVIGATION with the link back to the `ProductSearch.html` page:

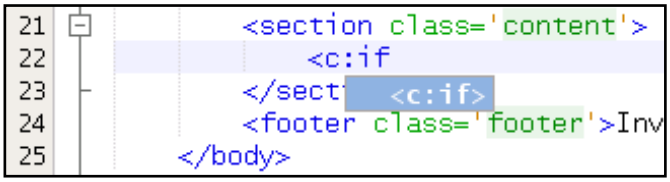
     ```
     <a href='/pm/ProductSearch.html'>Product Search</a>
     ```

   - Replace the text PAGE FOOTER so that the footer prints out the HTTP method that was used to invoke this servlet. The resulting footer element should contain this code: `Invoker used method ${pageContext.request.method}`

10. Modify the `ProductEdit.jsp` page to produce a product update form.

    a. Replace the PAGE CONTENT text with an `if` condition that checks if there is an update status available in the `ProductManager` CDI bean. You should use a JSTL `if` element to perform this check.

    Note: Start typing `<c:` and when a drop-down list of elements is displayed, select the `if` element and press Enter. This will not only add the element to a page but also automatically add a tag lib declaration to the top of this page.




```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:if test="${pm.status !=null}">
</c:if>
```

    b. Inside this `if` element place, an HTML divider that displays a status message. Assign an 'error' or 'info' CSS style class to this divider based on the error's Boolean variable value.

```
<div class="${(pm.errors)?'error':'info'}">${pm.status}</div>
```

    c. After the end of the `if` element, add a `form` element that should submit its data recursively to the same page, `ProductEdit.jsp`, using the `POST` method.

```
<form action="ProductEdit.jsp" method="POST">
</form>
```

    d. Inside this form, add input elements to represent attributes of the product that will be updated by this form. Each field should comprise label and input elements enclosed by an HTML divider with the visual class 'field' assigned to it. You should construct the following form fields:

    - Product ID should be represented by the text input item called `p_id`. The form should not allow users to update this item, so it should be marked as `readonly`. You bind this item value to the `id` attribute of the product object located in the `ProductManager` CDI bean. This is what this field code should look like:

```
<div class="field">
 <label for="pid">ID:</label>
 <input type="text" id="pid" name="p_id"
            value="${pm.product.id}" readonly>
</div>
```

- Product name should be represented by the text input item called `p_name`. This should be a mandatory item, so mark it as required. You bind this item value to the name attribute of the product object located in `ProductManager` CDI bean. This is what this field code should look like:

```
<div class="field">
 <label for="pname">Name:</label>
 <input type="text" id="pname" name="p_name"
           value="${pm.product.name}" required>
</div>
```

- Product price should be represented by the text input item called `p_price`. This should be a mandatory item, so mark it as required. You bind this item value to the price attribute of the product object located in `ProductManager` CDI bean. This is what this field code should look like:

```
<div class="field">
 <label for="pprice">Price:</label>
 <input type="text" id="pprice" name="p_price"
           value="${pm.product.price}" required>
</div>
```

- Product best-before-date should be represented by the date input item called `p_date`. You bind this item value to the `bestBefore` attribute of the product object located in `ProductManager` CDI bean. This is what this field code should look like:

```
<div class="field">
 <label for="pdate">Best Before:</label>
 <input type="date" id="pdate" name="p_date"
           value="${pm.product.bestBefore}" >
</div>
```

e. The last item in this form should be a submit button. This is what its code should look like:

```
<div class="field">
    <input type="submit" id="submit" value="Update">
</div>
```

11. Modify the `ProductList.jsp` page to provide navigation to the `ProductEdit.jsp` page:
    a. Open the `ProductList.jsp` page.
    b. Modify the line of code that produces product ID, so that it is displayed to the user as a clickable URL pointing to the `ProductEdit.jsp` page. Replace this line of code:

```
${p.id}
```

    with this:

```
<a href="ProductEdit.jsp?p_id=${p.id}">${p.id}</a>
```

12. Test the ProductWeb project:
    a. Compile the **ProductWeb** project using **Clean and Build**.
    b. Right-click the **ProductWeb** project and select **Deploy**.
    c. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm`
    d. Click the **Product Search** link to navigate from `index.html` to `ProductSearch.html`.
    e. In the Product Name field, enter `Co%` and click the **Find** button
    f. Click any product ID value in the `ProductList` page to navigate to the `ProductEdit` page.
    g. Once you are on the edit page, try to change the product price to any valid value (for example, `3.99`).
    h. Click the **Update** button.
    i. Observe the message that indicates successful update.

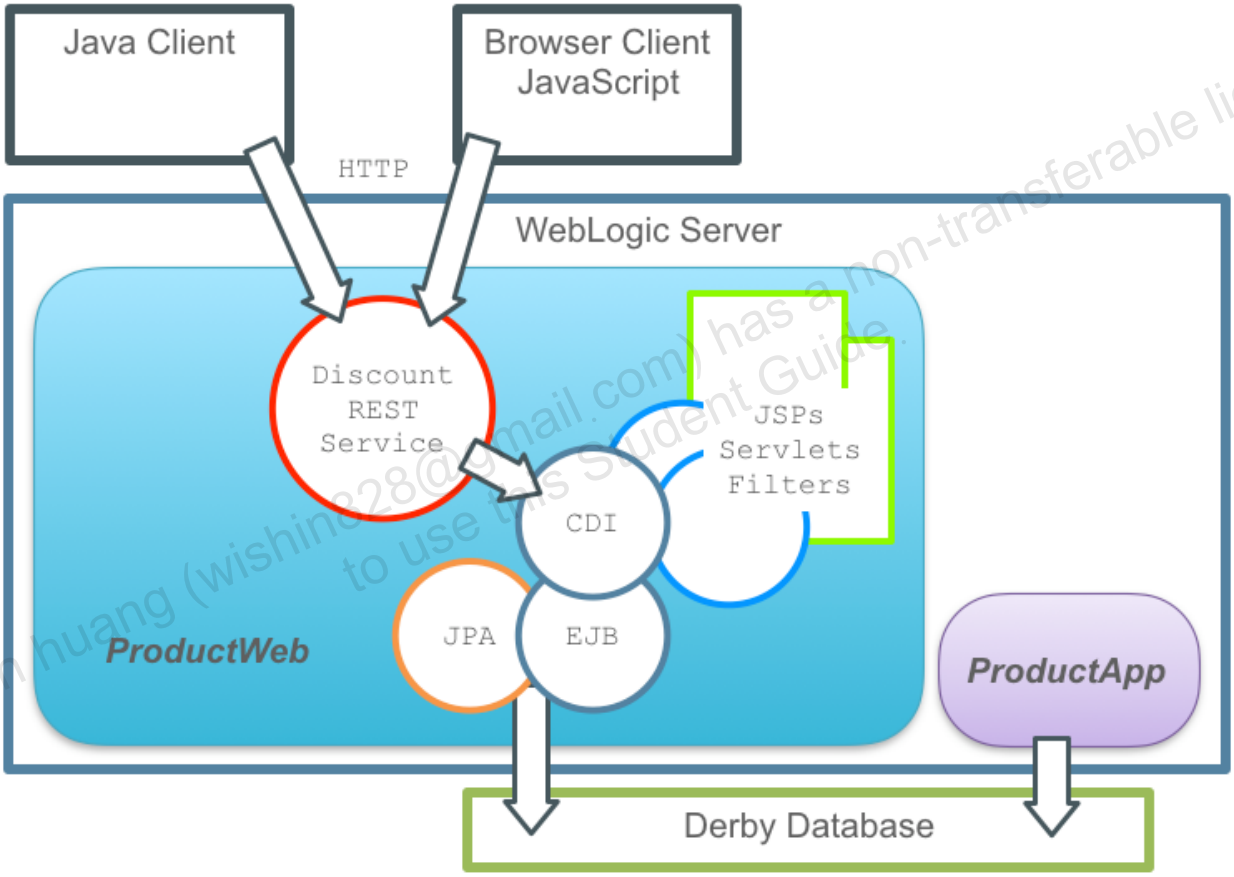
       Note: Look at the URL in the browser address bar. It shows `ProductEdit.jsp` page. The filter that has intercepting and handling calls is invisible to the client.
    j. Now change the product price to an invalid value (for example, `0.1`).
    k. Click the **Update** button.
    l. Observe the error message.
    m. Optionally (if you have time) test other cases, such as when product name is invalid (shorter than two symbols), or best-before-date format is wrong, or when the record is changed by another user to produce an optimistic lock exception. You can use your ProductApp-client or SQL Console to update the same product you selected on the Product Edit page to produce the optimistic lock error.

Practices for Lesson 9: Implementing REST Services by Using JAX-RS API

Practices for Lesson 9: Overview

Overview

In these practices, you create a REST service to check what discount could be available for a product. You invoke this service from an HTML page by using JavaScript. You also create a Java client to invoke this service.



Practice 9-1: Creating a REST Service

Overview

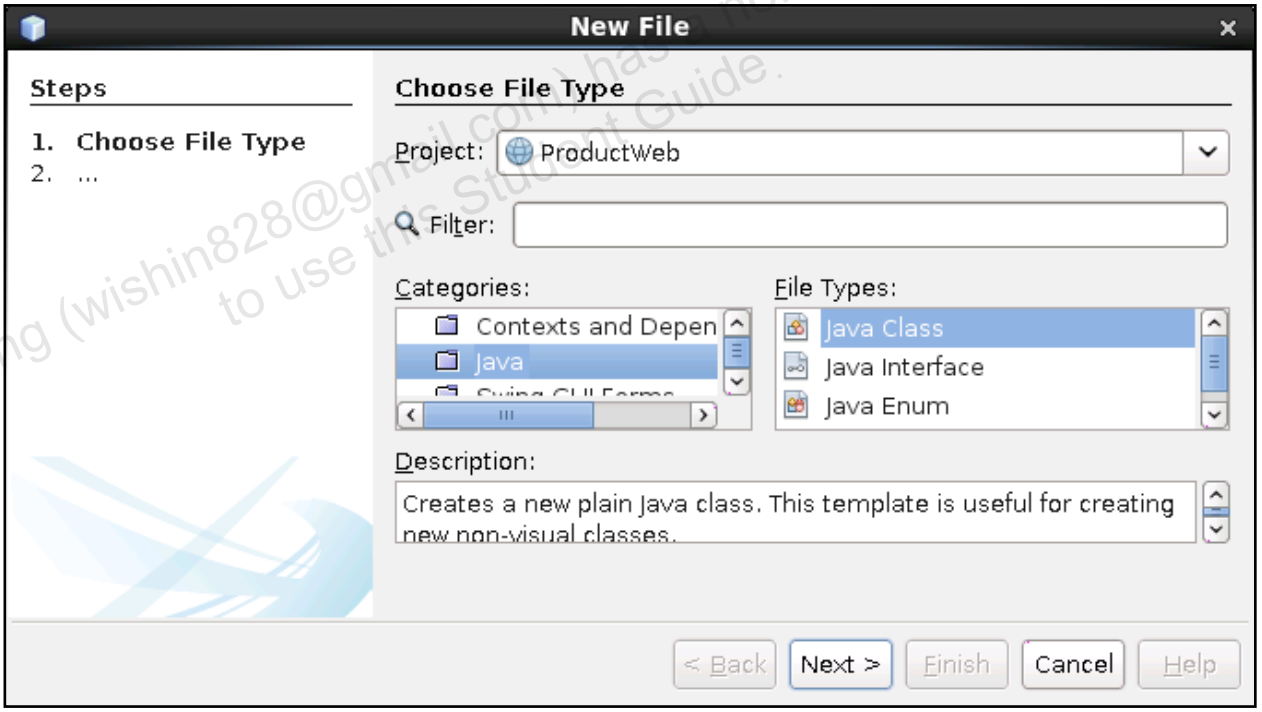
In this practice, you create new REST Service using JAX-RS API. This service should check if the product is eligible for a discount a day before its expiry date, provided that the discounted price is bigger than 1.

Assumptions

You have successfully completed all previous practices.

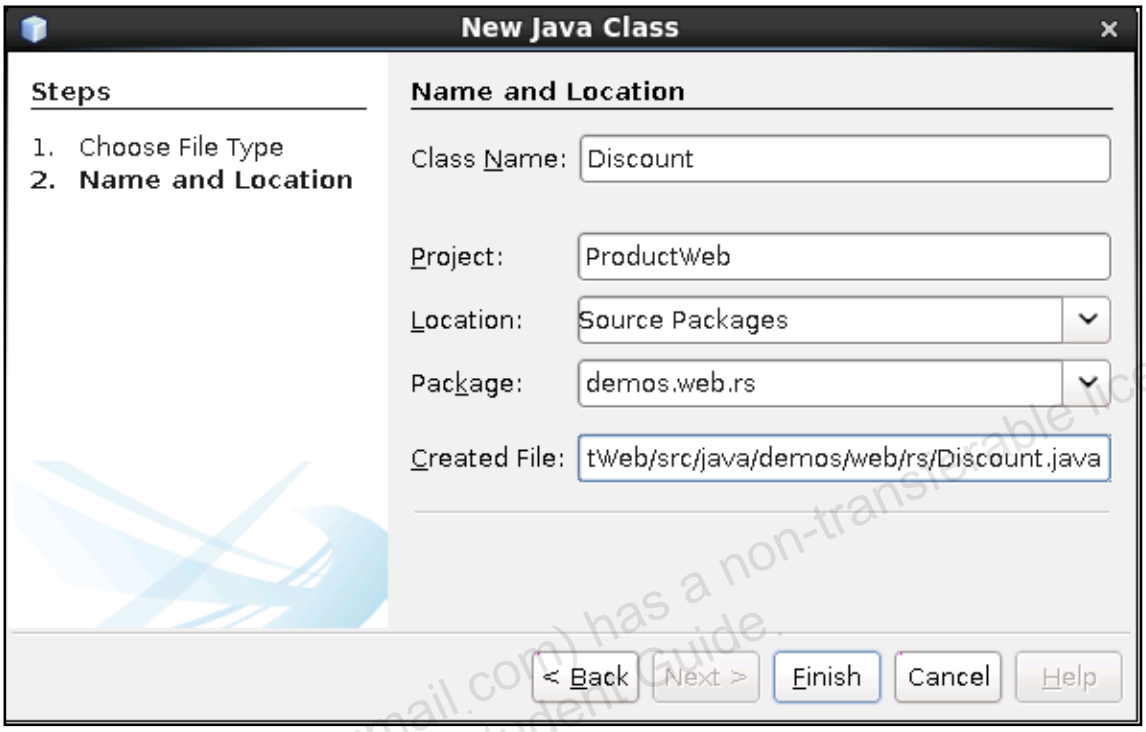
Tasks

1. Create a new Java class to represent a Discount object. This class should store values of the discount and of the date when this discount will be available.
   a. Select **ProductWeb** project in NetBeans.
   b. Select **File > New File**.
   c. Select **Java** from the list of Categories and **Java Class** from the list of File Types.



   d. Click **Next.**

e. In the New Java Class dialog box, set the following properties:
   - Class Name: `Discount`
   - Package: `demos.web.rs`



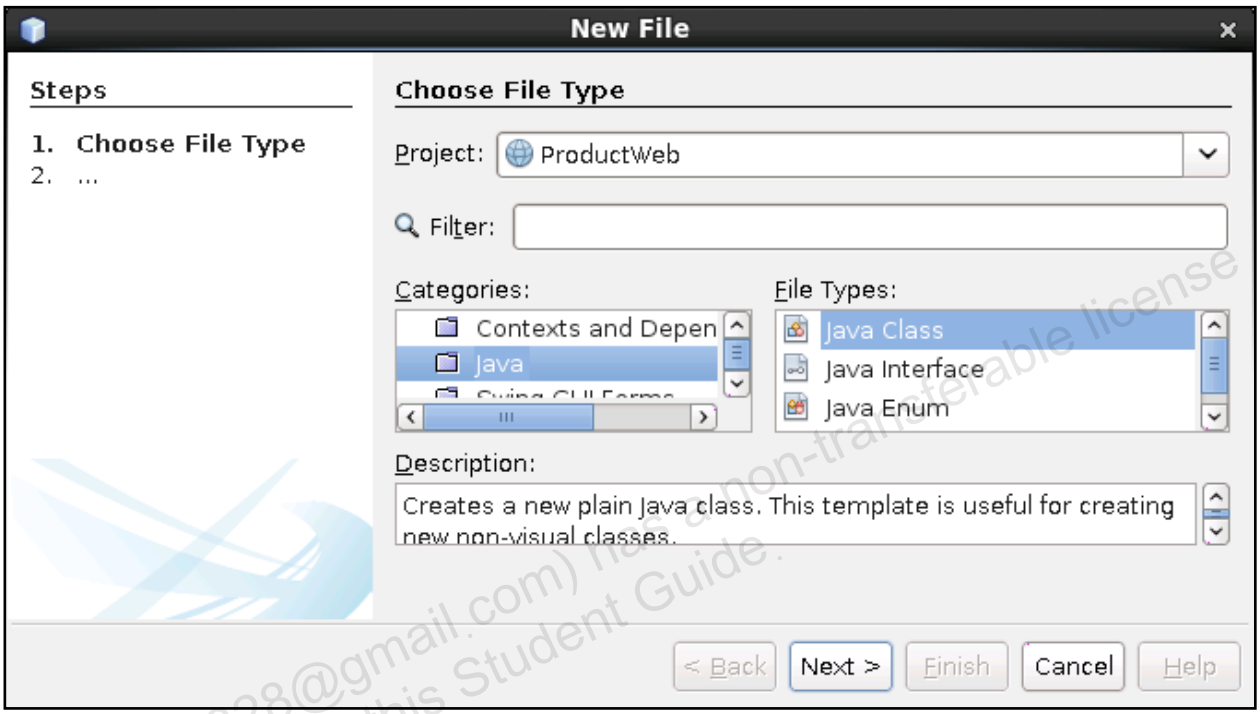
f. Click **Finish.**

2. Modify the Discount class to be capable of holding values of discount and availability date:
   a. Open the **Discount** class.
   b. Add a `implements Serializable` interface clause to the Discount class definition.
   c. Add an import: `java.io. Serializable`
   d. Add two instance variables to store a BigDecimal value of a discount and a LocalDate value of a date when it will be available.

```
private BigDecimal value;
private LocalDate date;
```

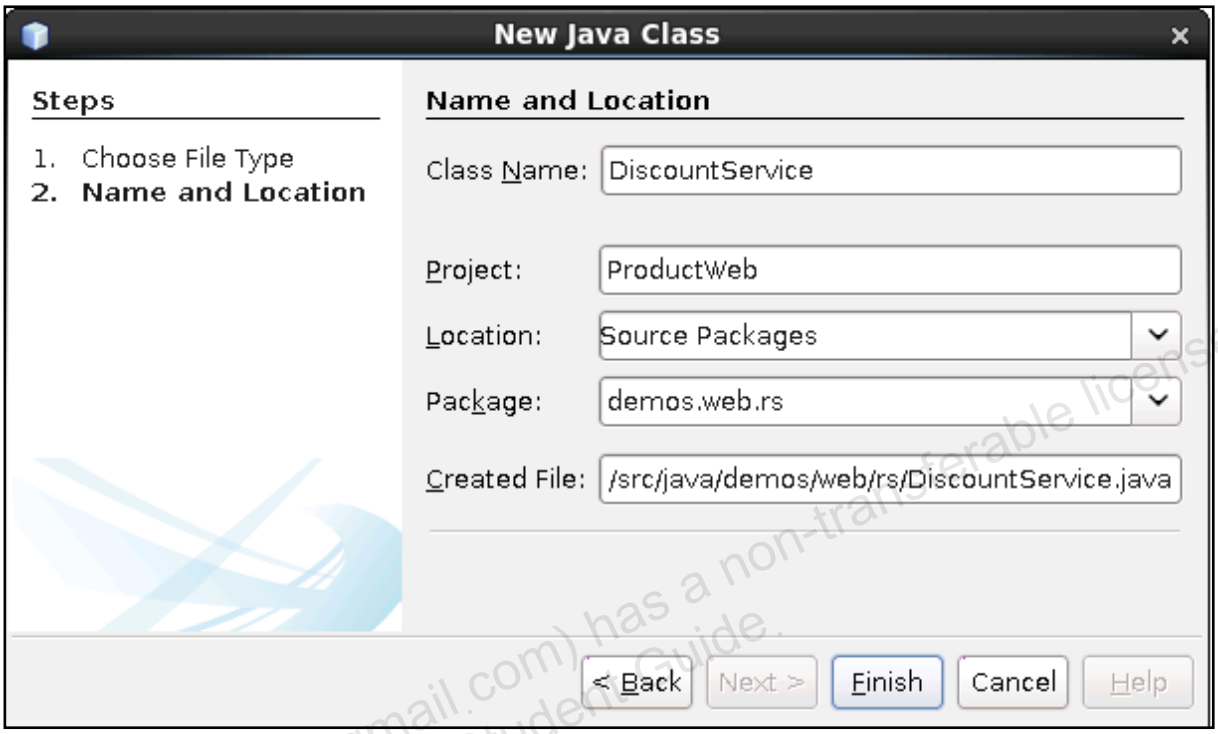
   e. Add imports: `java.math.BigDecimal` and `java.time.LocalDate`
   f. Produce getter and setter operations for these two fields. You can right-click an empty line of code inside the body of the Discount class, select **Insert Code > Getter and Setter**, and in the code generation dialog box select all check boxes to generate these operations.

3. Create a REST Resource class called DiscountService that should handle the algorithm determining the discount availability:
   a. Select the **ProductWeb** project.
   b. Select **File > New File**.
   c. Select **Java** from the list of Categories and **Java Class** from the list of File Types.



   d. Click **Next.**

e. In the New Java Class dialog box, set the following properties:
   - Class Name: `DiscountService`
   - Package: `demos.web.rs`



f. Click **Finish.**

4. Annotate the DiscountService class as a REST resource:
   a. Open the **DiscountService** class.
   b. Just before the line of code that defines the class, add an annotation to make this class a CDI request-scoped bean.
      ```
      @RequestScoped
      ```
   c. Add an import: `javax.enterprise.context.RequestScoped`
   d. Add another annotation binding DiscountService class to path 'discount':
      ```
      @Path("discount")
      ```
   e. Add an import: `javax.ws.rs.Path`
   f. Add annotations that specify this resource as a producer and consumer of the JSON media type.
      ```
      @Produces({MediaType.APPLICATION_JSON})
      @Consumes({MediaType.APPLICATION_JSON})
      ```
   g. Add the following imports:
      ```
      javax.ws.rs.Produces
      javax.ws.rs.Consumes
      javax.ws.rs.core.MediaType
      ```

5. Modify the DiscountService class to reference the ProductManager bean:
   a. Inside the DiscountService class body, add an injection of the ProductManager CDI bean.
   ```
   @Inject
   private ProductManager pm;
   ```
   b. Add imports: `javax.inject.Inject` and `demos.model.ProductManager`

6. Add an operation to determine the discounted price:
   a. This operation should:
      - Be mapped to the HTTP method `GET`
      - Be mapped to the Path Parameter ID
      - Return the Discount object
      - Accept the Path Parameter ID as an integer

   ```
   @GET
   @Path("{id}")
   public Discount getDiscount(@PathParam("id") Integer id) {
        // discount calculation will be added here
   }
   ```
   b. Add imports: `javax.ws.rs.GET` and `javax.ws.rs.PathParam`
   c. Inside the getDiscount operation, create a new Discount object and assign it to discount variable. Return this object.
   ```
   Discount discount = new Discount();
   // discount calculation will be added here
   return discount;
   ```
   d. After the line of code that creates a new Discount object, add code to invoke the ProductManager bean to find a product with a specific ID.
   ```
   Product product = pm.findProduct(id);
   ```
   e. Add an import: `demos.db.Product`
   f. Check if the product you were looking for is not null. If it is, you need to send an error to the client.
   ```
   if (product == null) {
      throw new WebApplicationException(Response.Status.NOT_FOUND);
   }
   ```
   g. Add imports: `javax.ws.rs.WebApplicationException` and `javax.ws.rs.core.Response`

h. Get price of this product and calculate a 10% discount value.

```
BigDecimal price = product.getPrice();
BigDecimal value = price.multiply(BigDecimal.valueOf(0.1));
```

i. Add an import: `java.math.BigDecimal`

j. Get best-before-date from the product

```
LocalDate date = product.getBestBefore();
```

k. Add an import: `java.time.LocalDate`

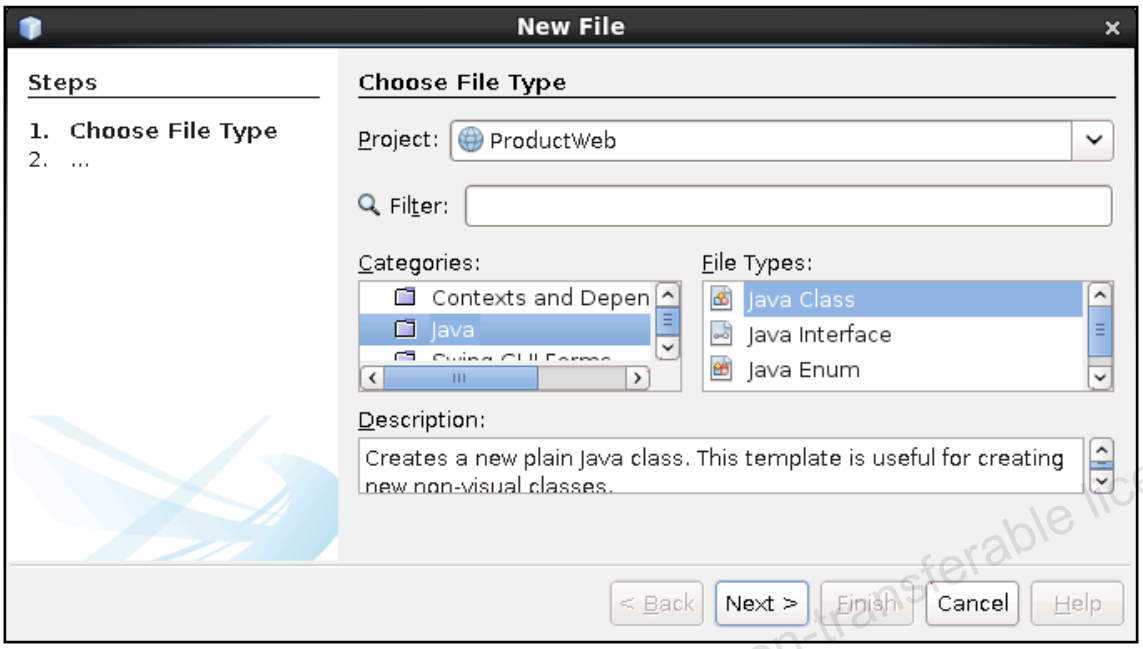
l. Check if the price is still bigger than 1 and that best before date is later than tomorrow. If it is the case, place the discount value and date minus one day value into the discount object; otherwise set a zero discount.

```
if (price.subtract(value).compareTo(BigDecimal.ONE) > 0 &&
   (date != null && date.compareTo(LocalDate.now()) > 0)) {
      discount.setValue(value);
      discount.setDate(date.minusDays(1));
} else {
      discount.setValue(BigDecimal.ZERO);
}
```

Note: This algorithm produces a discount object that will contain a discount value and the date on which it will be available (one day before the product expires), if the discounted price is still greater than 1 and the product has a valid best before date that is in the future. It will return a discount object with a 0 value and no date if this is not the case.

7. Create a REST Application class called RestApplication that should register rest resources:
   a. Select the **ProductWeb** project.
   b. Select **File > New File.**

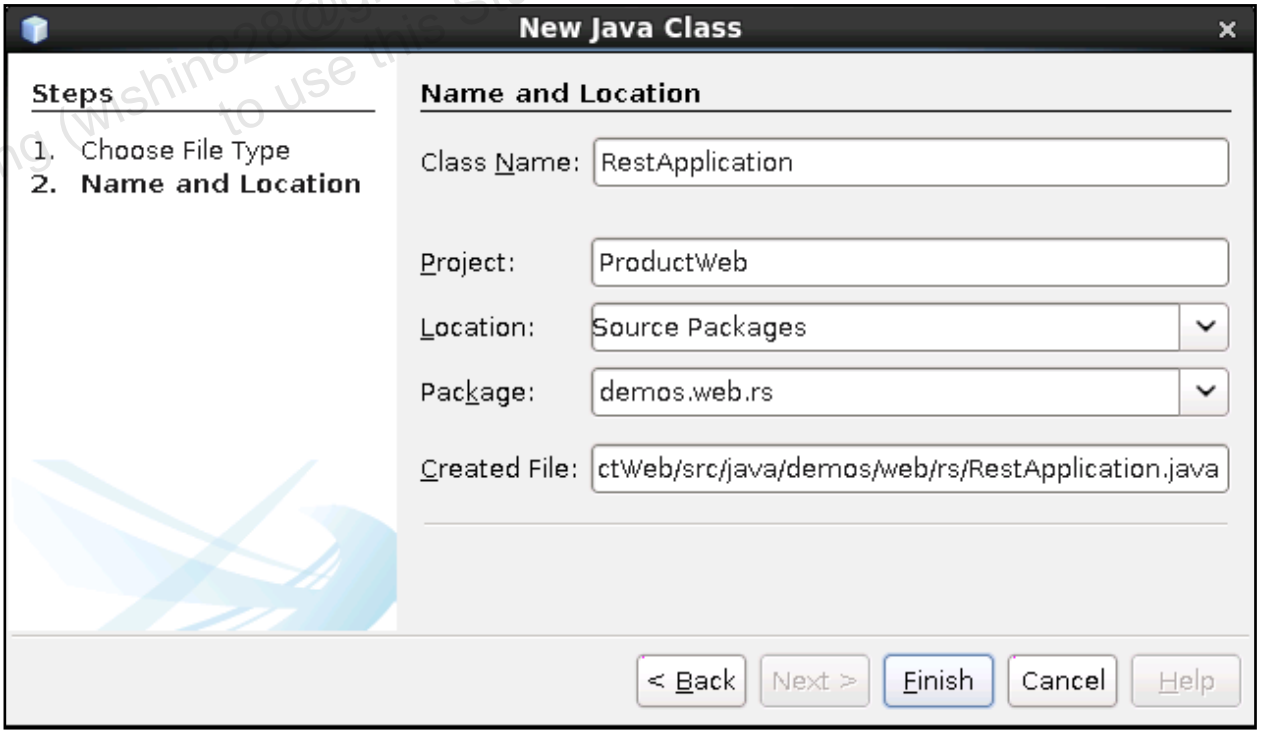
c. Select **Java** from the list of Categories and **Java Class** from the list of File Types.



d. Click **Next.**

e. In the New Java Class dialog box, set the following properties:

- Class Name: `RestApplication`
- Package: `demos.web.rs`



f. Click **Finish.**

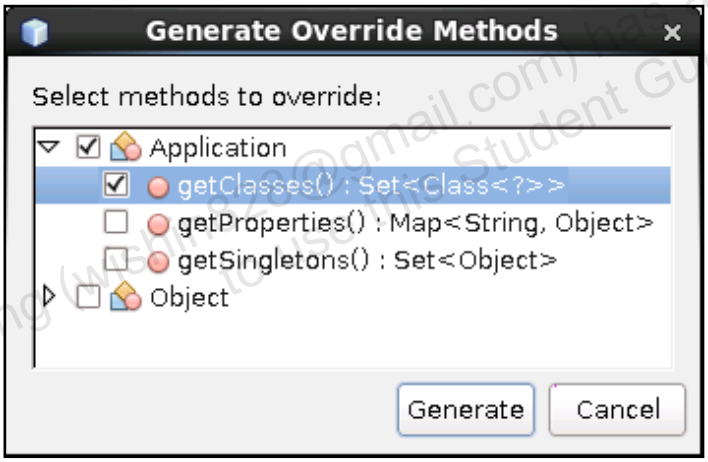
8. Modify the RestApplication class to be your REST Application handler and register the DiscountService class with this application:
   a. Open the **RestApplication** class.
   b. In the line of code just before the class definition, add an annotation to bind the RestApplication class to the 'rs' URL and add an `extends` clause to the RestApplication class definition to extend the Application class.

```
@ApplicationPath("rs")
public class RestApplication extends Application {
    // you will add more code here
}
```

   c. Add imports of the following classes:

```
javax.ws.rs.ApplicationPath
javax.ws.rs.core.Application
```

   d. Override the **`getClasses`** method. You can create a new empty line of code inside the body of the RestApplication class. Right-click this line and select **Insert Code > Override Method**, and select the check box for the `getClasses` method in the **Generate Override Methods** dialog box.



   e. Click **Generate**.

   Note: This is the code that you have just generated with an added comment, explaining where you will add more code:

```
@Override
public Set<Class<?>> getClasses() {
    // add code to register DiscountService class here
    return super.getClasses();
}
```

   f. Add code inside the `getClasses` operation to register the DiscountService class with the HashSet of classes.

```
final Set<Class<?>> classes = new HashSet<>();
classes.add(DiscountService.class);
```

g. Add imports: `java.util.Set` and `java.util.HashSet`

h. Replace the return statement to return your classes object.

```
return classes;
```

9. Test the Discount Service:

a. Compile the **ProductWeb** project using **Clean and Build**.

b. Right-click the **ProductWeb** project and select **Deploy**.

c. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm/rs/discount/1`

Note: It is possible to test this service with such a simple call because DiscountService resource is responding to the `GET` operation. You will create a more realistic application code to use this service in the next practice section.

d. Depending on what your product's best-before-date and price are, you may get a discount value of zero or not. You can update any product by using your ProductWeb application. Change its price to more than 1 and the best-before-date later than tomorrow. Then put this product ID at the end of the rest service URL to test the discount value calculation.

Practice 9-2: Invoking a REST Service by Using JavaScript

Overview

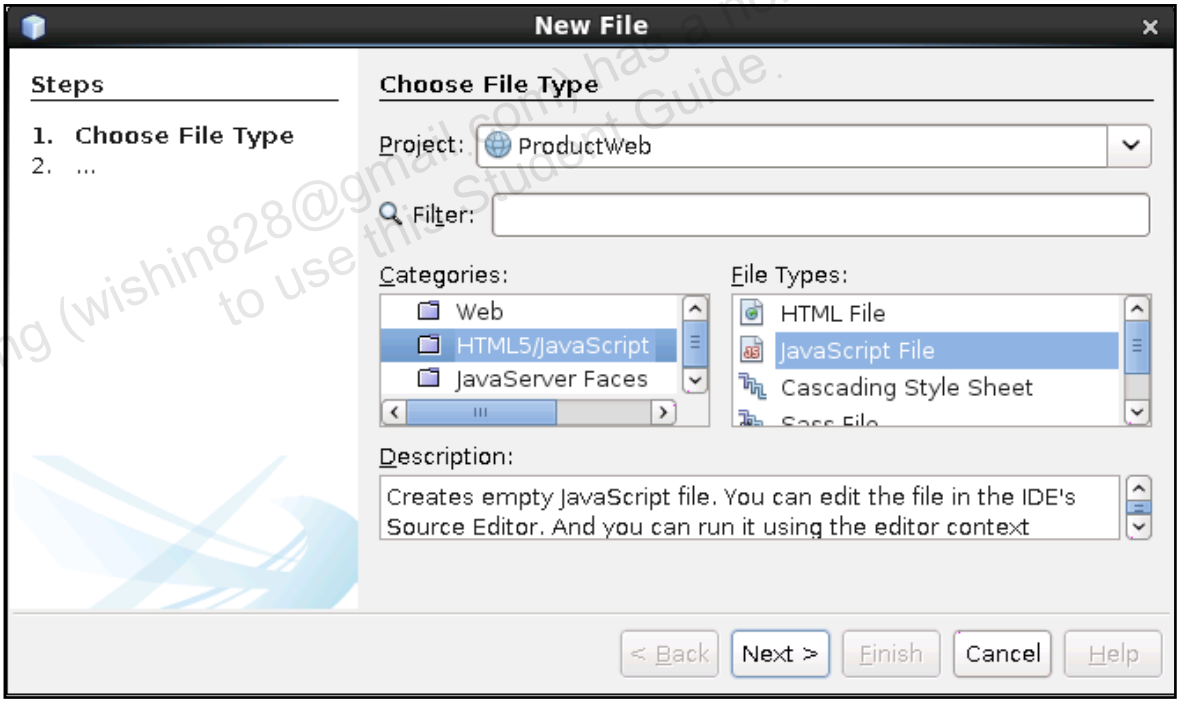
In this practice, you add JavaScript invocation functionality to the ProductEdit.jsp page to invoke the Discount REST service.

Assumptions

You have successfully completed all previous practices.

Tasks

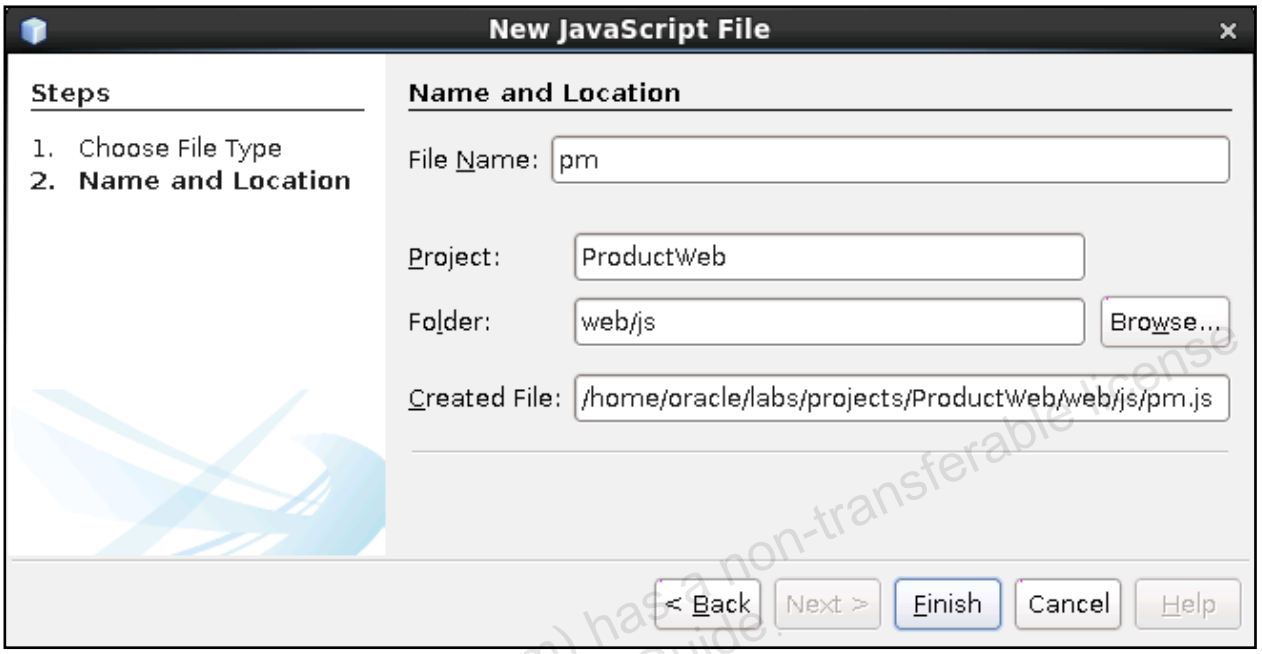
1. Add a JavaScript file to the ProductWeb project. This JavaScript file should contain code that invokes the Discount REST Service:
   a. Select **ProductWeb** project in NetBeans
   b. Select **File > New File**.
   c. Select **HTML5/JavaScript** from the list of Categories and **JavaScript File** from the list of File Types.



   d. Click **Next.**

e. In the New JavaScript File dialog box, set the following properties:

- File Name: `pm`
- Folder: `web/js`



f. Click **Finish.**

g. Add JavaScript code to the `pm.js` file:

Note: You can simply replace entire content of the `pm.js` file with text from the `pm.js` file located in the `/home/oracle/labs/resources` folder.

This is the resulting source code:

```
function getDiscount() {
  var request = new XMLHttpRequest();
  var id = document.getElementById("pid").value;
  var url = "http://localhost:7001/pm/rs/discount/" + id;
  request.open('GET', url, true);
  request.send();
  request.onreadystatechange = function () {
    if (request.readyState == 4) {
      if (request.status == 200) {
        var discount = JSON.parse(request.responseText);
          if (discount.value > 0) {
            alert("Possible discount of " + discount.value +
                  " becomes available on " + discount.date);
          } else {
            alert("This product is not eligible for the discount");
          }
        } else {
          alert("Error getting discount: " + request.responseText);
    }
    }
  };
}
```

Note: This JavaScript file declared a function called `getDiscount`. Here is a brief explanation of the logic of the `getDiscount` function:

- It creates a new XMLHttpRequest—this is a JavaScript object used to invoke HTTP resources, such as REST or SOAP Services, or any other HTTP URLs.
- It extracts the product ID value from the input text item in the HTML document with ID "pid". You are going to attach this JavaScript file to the ProductEdit.jsp page that contains the item.
- It constructed a URL path comprised of the product ID and the URL pointing to the deployed REST Service resource.
- It opens a URL connection and transmits an HTTP `GET` request
- It registers a callback function against the XMLHttpRequest object for the `onreadystatechange` event.
- This function will be invoked to handle the response returned from the server.

- This callback function checks if the response has been received by looking at its status, and then it also checks if the response contains an HTTP response code 200, which indicates a normal response to the `GET` operation.
- If the code is not 200, a normal response has not been produced, so an alert containing a warning is displayed.
- If a normal response has been produced, this function converts the response into a JSON object and checks if this object contains a discount that is greater than 0. If this is true, it displays an alert showing the discount date and value: otherwise, it displays an alert that the discount is not available.

Note: This course is about Java not JavaScript, which is an entirely different programming language. This is why you are given this script rather than being asked to write its code yourself. If you are interested in learning more about what this code does and how to program with JavaScript, Oracle University offers a course *JavaScript and HTML5: Develop Web Applications*.

2. Attach the JavaScript file to the ProductEdit page:
   a. Open the **ProductEdit.jsp** file.
   b. Create a new line inside the **head** section just after the **link** element that attached the **pm.js** file to this page.
   c. To this new line, add a script element that will attach the `pm.js` JavaScript file to this page.
      ```
      <script type="text/javascript" src="/pm/js/pm.js"></script>
      ```
3. Add a button the ProductEdit page to invoke the `getDiscount` function.
   a. Find an **`input type submit`** element inside the ProductEdit page. It is located in the last **`div`** element inside the **`form`** element.
   b. Add a new line of code inside the same **div** element.
   c. Add a **button** to this new line of code that invokes the `getDiscount` function as the **`onclick`** event handler.

```
<input type="button" value="Check Discount" onclick="getDiscount()">
```

4. Test the Discount Service:
   a. Compile the **ProductWeb** project using **Clean and Build**.
   b. Right-click the **ProductWeb** project and select **Deploy.**
   c. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm`
   d. Click the **Product Search** link to navigate from index.html to the ProductSearch.html page.
   e. Into the Product Name field type `%` and then click the Find button. This will display all of your products.

f. Click any product ID value in the ProductList page to navigate to the ProductEdit page.

g. Click the getDiscount button.

h. Depending on what your product's best-before-date and price are, you may get a discount value of zero or not. You can update any product by using same page, so its price will be more than 1 and its best-before-date later than tomorrow, and then recheck discount availability.

Practice 9-3: Invoking a REST Service by Using Java

Overview

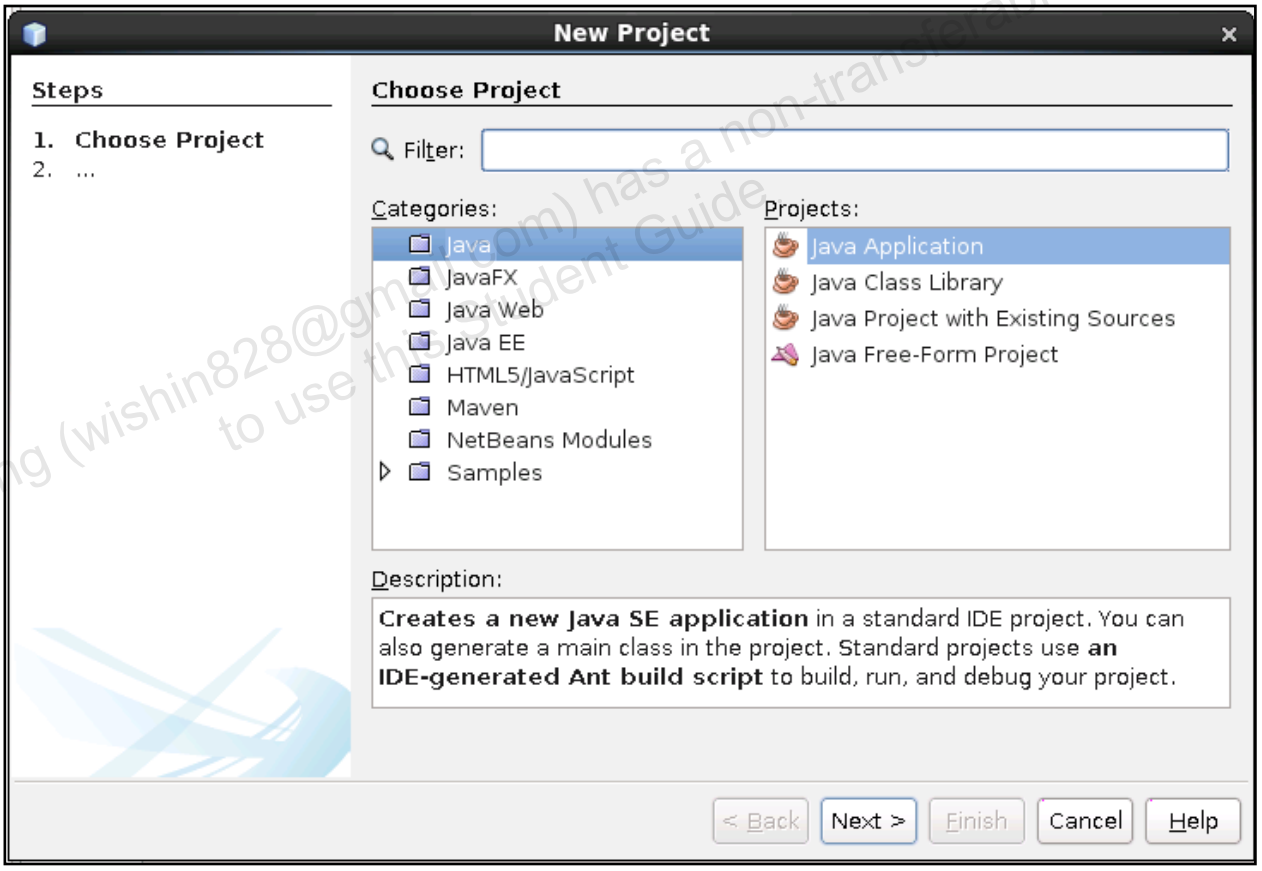
In this practice, you create a new Java Application that invokes the Discount REST service.

Assumptions

You have successfully completed all previous practices.

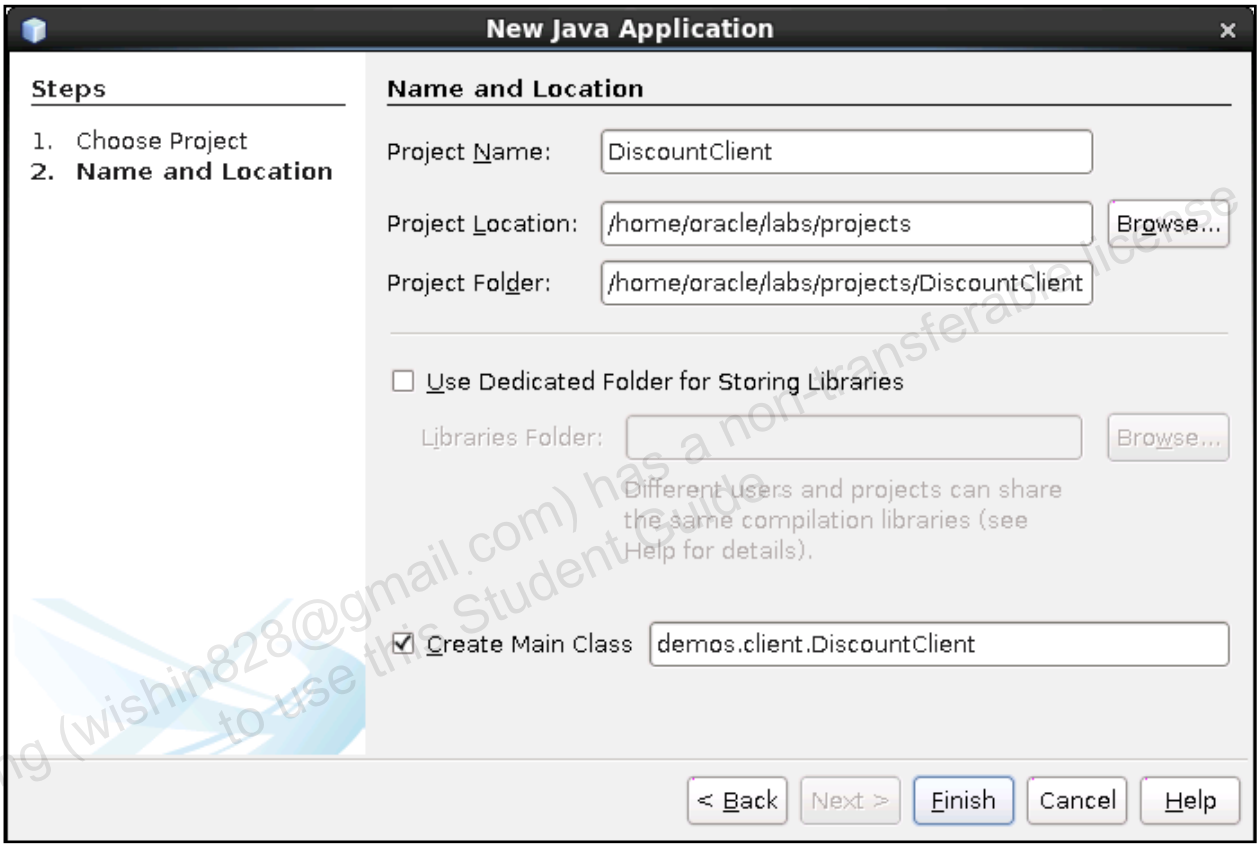
Tasks

1. Create a new Java Application project called DiscountClient to invoke the Discount REST Service:
   a. Select **File > New Project.**
   b. Select **Java** from the list of Categories and **Java Application** from the list of Projects.



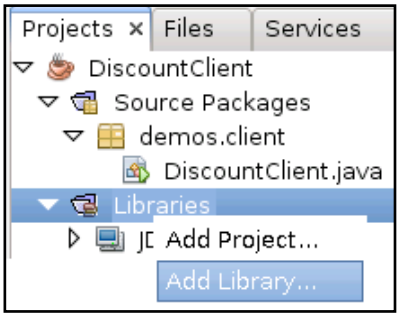
   c. Click **Next.**

d. In the New Java Application dialog box, set the following properties:
   - Project Name: `DiscountClient`
   - Project Location: `/home/oracle/labs/projects`
   - Project Folder: `/home/oracle/labs/projects/DiscountClient`
   - Select the **Create Main Class** check box and set class name to: `demos.client.DiscountClient`

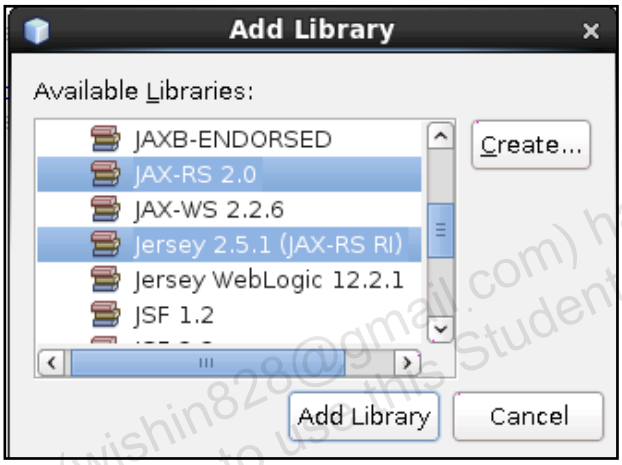


e. Click **Finish.**

2. Add JAX-RS libraries to the DiscountClient project:
   a. Expand the **DiscountClient** Project and right-click the **Libraries** node.
   b. Select **Add Library.**



   c. In the Add Library dialog box, select the **JAX-RS 2.0** and **Jersey 2.5.1 (JAX-RS RI)** libraries.



   d. Click the **Add Library** button.

   Note: This project environment is essentially Java SE, so you had to add extra libraries to its classpath to enable REST service handling.

3. Create a Java class called **Discount** in the **demos.client.rs** package to represent the REST service response within this client. This class should implement the **Serializable** interface. This class should contain **two instance variables**:
   - A **BigDecimal** value representing **discount**
   - A **String** variable representing **date**
   - **Getter and Setter methods** for these variables

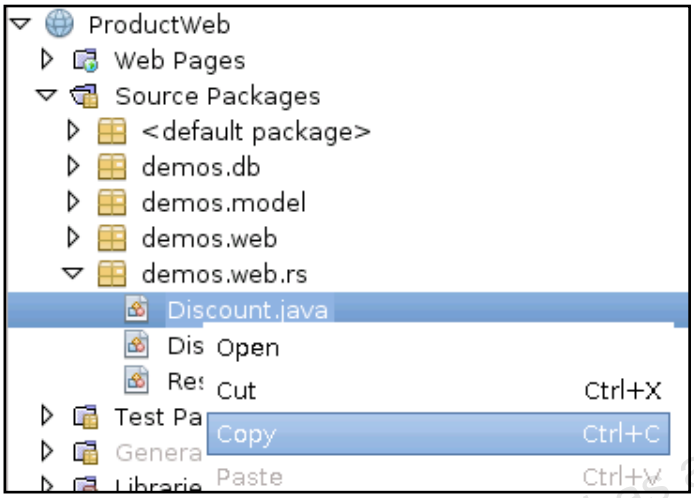
     Note: HTTP protocol that is used by REST Services transmits all data as text. REST API can automatically convert text values to Java types. However, custom converts can also be supplied. Therefore, it is possible to convert the received date value from text to LocalDate. However, this is not required for the purposes of this practice.

Note: You may create this class as usual, like you have done with other classes, or you can do the following to copy and refactor this class from the code that exists in the ProductWeb project.

a. Expand the **ProductWeb** project.

b. Expand **Source Packages > demos.web.rs** node.

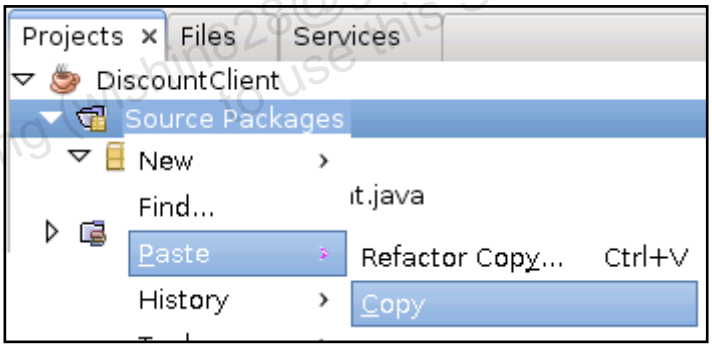
c. Right-click the **Discount** class and select **Copy.**



d. Expand the **DiscountClien**t project.

e. Right-click the **Source Packages** node.

f. Select **Paste > Copy.**

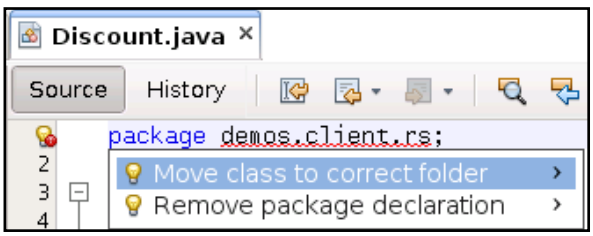


g. Open the **Discount** class located inside the **<default package>** node inside the **DiscountClient** project.

h. Modify its package declaration.

```
package demos.client.rs;
```

i. Right-click the light bulb icon on the left side of this line of code and select **Move class to correct folder.**



j. Change the type of the **date** variable inside this class to **String**.

k. Change the return type of the **getDate** method to **String**.

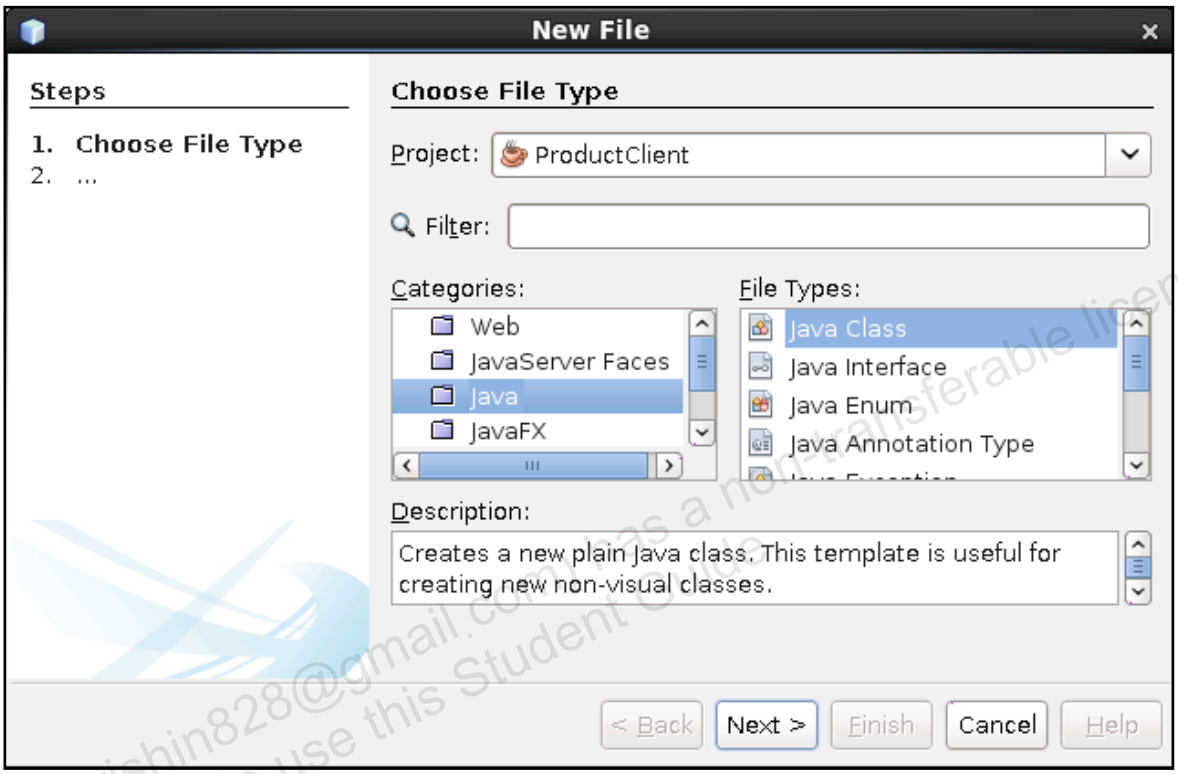
l. Change the parameter type of the **setDate** method to **String**.

m. Right-click anywhere in the source code of the Discount class and select **Fix Imports** menu to remove the unused import of `java.time.LocalDate`.

This is the resulting code:

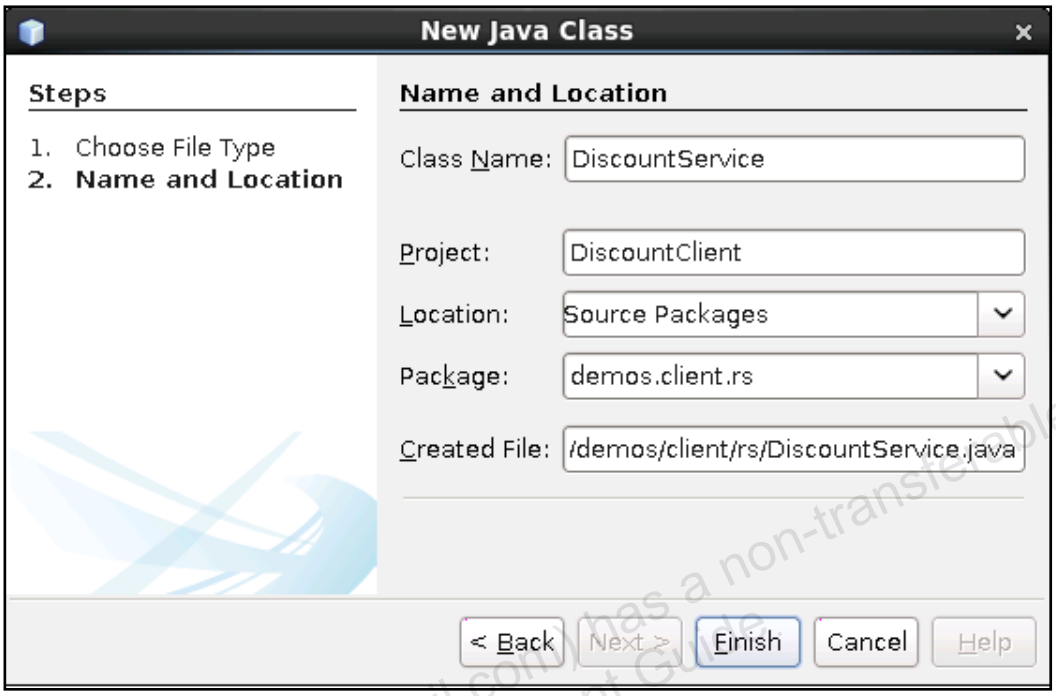
```
package demos.client.rs;
import java.math.BigDecimal;
import java.io.Serializable;
public class Discount implements Serializable {
    private BigDecimal value;
    private String date;
    public BigDecimal getValue() {
        return value;
    }
    public void setValue(BigDecimal value) {
        this.value = value;
    }
    public String getDate() {
        return date;
    }
    public void setDate(String date) {
        this.date = date;
    }
}
```

4. Create a Java class called **DiscountService** in the same **demos.client.rs** package to represent the REST service and its getDiscount operation:
   a. Select the **DiscountClient** project.
   b. Select **File > New File**.
   c. Select **Java** from the list of Categories and **Java Class** from the list of File Types.



   d. Click **Next.**

e. In the New Java Class dialog box, set the following properties
   - Class Name: `DiscountService`
   - Package: `demos.client.rs`



f. Click **Finish.**

5. Add code to the DiscountService class to handle invocation of the Discount REST Service:
   a. Open the **DiscountService** class from the **demos.client.rs** package inside the **DiscountClient** project.
   b. Add an instance variable to represent `WebTarget`.

```
private WebTarget webTarget;
```

   c. Add an import: `javax.ws.rs.client.WebTarget`
   d. Add an instance variable to represent the rest service `Client`.

```
private Client client;
```

   e. Add an import: `javax.ws.rs.client.Client`
   f. Add a constant to represent the remote service base URI.

```
private static final String BASE_URI = "http://localhost:7001/pm/rs";
```

   g. Add a parameter-less constructor and initialize the `client` object. Inside this constructor, initialize the `webTarget` object, by assigning a base URL target and resource URL path to the client object.

```
public DiscountService() {
  client = ClientBuilder.newClient();
  webTarget = client.target(BASE_URI).path("discount");
}
```

h. Add an import: `javax.ws.rs.client.ClientBuilder`

i. Add a `close` operation to the DiscountService class that should close the client.

```
public void close() {
     client.close();
}
```

j. Add an operation to represent the REST service's `getDiscount` method. This operation should be called **getDiscount**. Accept the **Integer** product ID as an argument and return the **Discount** object.

```
public Discount getDiscount(Integer id) {
     // get discount service invocation code will be placed here
}
```

k. Inside the **getDiscount** operation, set a path parameter to the `webTarget` object.

```
webTarget = webTarget.path(id.toString());
```

l. To Invoke a remote service and obtain a response from it, add the following code:

- Register the Discount class with the `webTarget`. This will allow JAX-RS API to perform conversion of the JSON object received from the server into your Java object type of Discount.
- Form a new request with Media Type JSON (this is what the remote service expects).
- Build the HTTP `GET` request.
- Send this request to the REST service by using the `invoke` method.
- This code should be added inside the `getDiscount` method, after the line of code that sets the path parameter to `webTarget`.

```
Response response = webTarget.register(Discount.class)
                    .request(MediaType.APPLICATION_JSON)
                     .buildGet().invoke();
```

m. Inside the `getDiscount` method, add code that reads the response entity you receive from the REST service, converts it to a `discount` object type, and returns the discount object.

```
Discount discount = response.readEntity(Discount.class);
return discount;
```

n. Add an import javax.ws.rs.core.Response.

6. Add code to the `DiscountClient` class to make an invocation of `DiscountService`:

a. Open the **DiscountClient** class from the **demos.client** package inside the **DiscountClient** project.

b. Replace the "TODO" comment inside the method main of this class with a line of code that creates a new DiscountService object.

```
DiscountService ds = new DiscountService();
```

c. Add an import: `demos.client.rs.DiscountService`

d. After that, add another line of code to the `main` method to invoke the `getDiscount` operation upon the `DiscountService` object to retrieve the Discount object.

```
Discount discount = ds.getDiscount(new Integer(2));
```

e. Add an import: `demos.client.rs.Discount`

Note: Instead of product ID 2 in this example, you can add any other valid product ID.

f. Create an `if` / `else` statement to check if the discounted object contains a value of more than 0 or not. If it does contain such a value, print it together with the date to the console; otherwise, print the message that the discount is not available.

```
if (discount.getValue().compareTo(BigDecimal.ZERO) != 0) {
System.out.println("Possible discount of " + discount.getValue()
                    +", becomes available on "+discount.getDate());
}else{
System.out.println("This product is not eligible for the discount");
}
```

g. Add an import: `java.math.BigDecimal`

h. After the end of the `if` / `else` block, just before the end of the `main` method, add code that closes the DiscountService object.

```
ds.close();
```

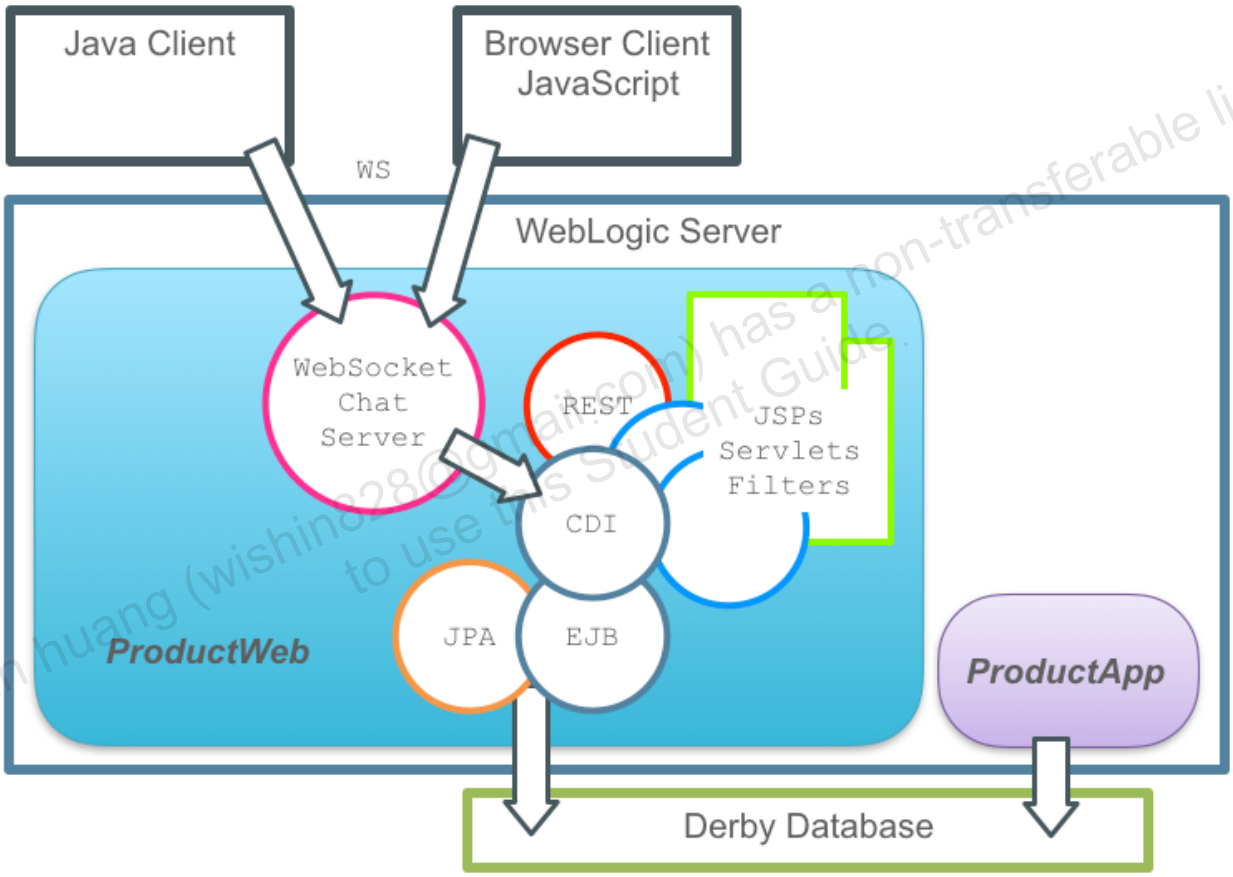
7. Test the Discount Service Java Client:
   a. Compile the **DiscountClient** project using **Clean and Build.**
   b. Run the **DiscountClient.**
   c. Observe messages printed from the `main` method to the console. Depending on what product ID you have used to pass as an Integer input parameter to the `getDiscount` operation, you would get different responses, where some products are or are not eligible for the discount.

Practices for Lesson 10: Creating Java Applications by Using WebSockets

Practices for Lesson 10: Overview

Overview

In these practices, you create a WebSocket Server component that allows different callers to exchange chat messages. You invoke this chat service from HTML 5 and JavaScript code in a browser. You also create a Java client to invoke this chat.



Practice 10-1: Creating a WebSocket Chat Server Endpoint

Overview

In this practice, you create a new WebSocket server endpoint to implement chat service functionalities.

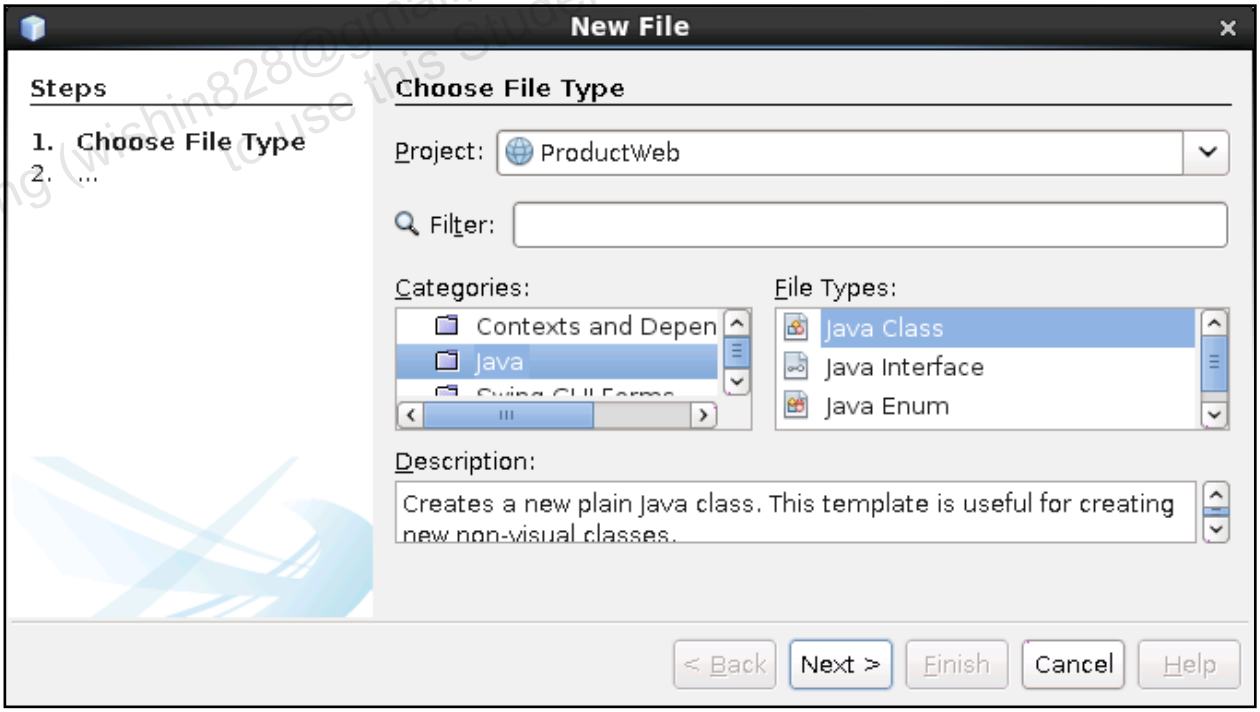
Assumptions

You have successfully completed all previous practices.

Tasks

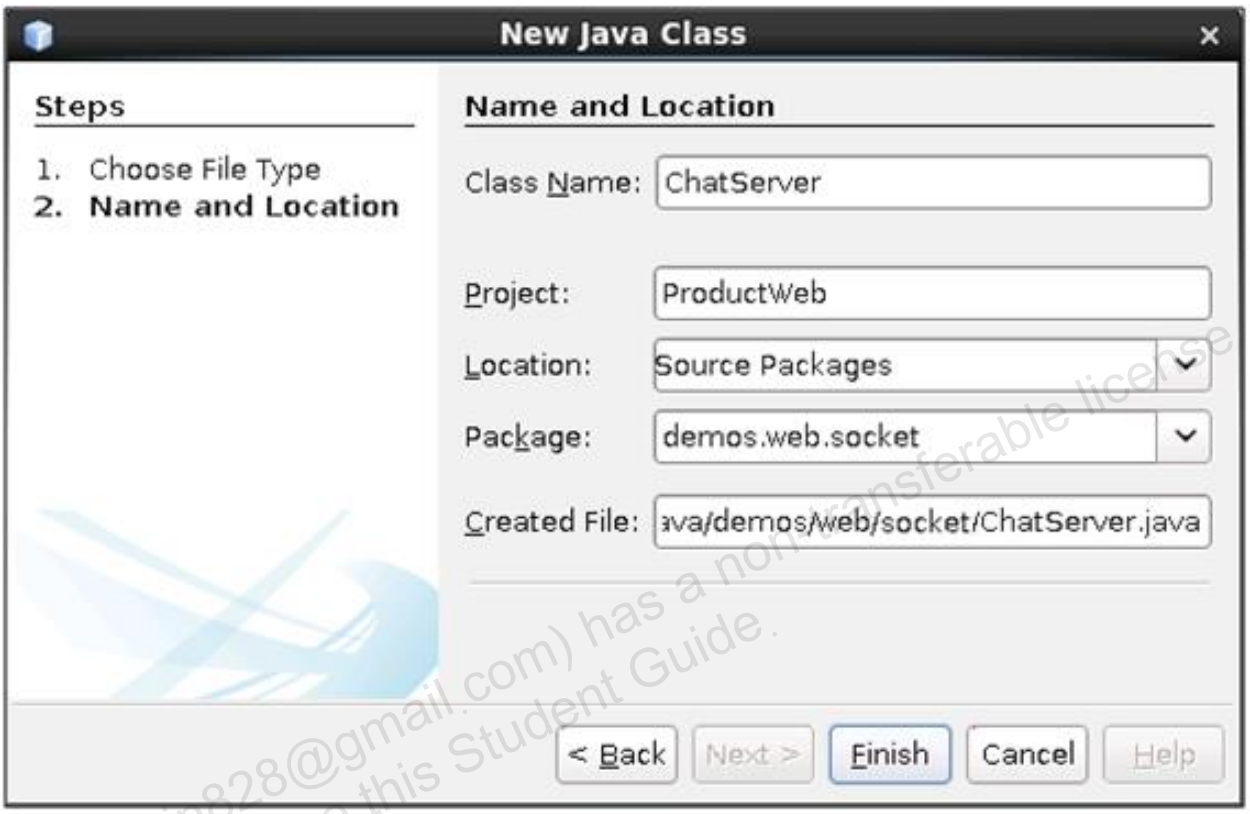
1. Create a new Java class to represent a WebSocket server endpoint. This class should handle web socket communications with the following functionalities:
   - Open and close sockets
   - Receive messages and errors
   - Send messages to WebSocket client endpoints

   a. Select the **ProductWeb** project in NetBeans.
   b. Select **File > New File.**
   c. Select **Java** from the list of Categories and **Java Class** from the list of File Types.



   d. Click **Next.**

e. In the New Java Class dialog box, set the following properties:
   - Class Name: `ChatServer`
   - Package: `demos.web.socket`



f. Click **Finish.**

2. Add code to the ChatServer class to prepare it to handle WebSocket communications:
   a. Open the **ChatServer** class located in the **demos.web.socket** package.
   b. Annotate this class to be a WebSocket **ServerEndpoint** and map it to the URL **'/chat'**. This code should be added on a new line of code before the class declaration.
      ```
      @ServerEndpoint("/chat")
      ```
   c. Add an import: `javax.websocket.server.ServerEndpoint`
   d. Inside the ChatServer class body, declare and initialize a Logger reference to be used by this class.
      ```
      private static final Logger logger =
                      Logger.getLogger(ChatServer.class.getName());
      ```
   e. Add an import: `java.util.logging.Logger`
   f. Declare and initialize another constant to hold a **Set** object containing WebSocket **sessions**. This should be a **thread-safe collection**, because later you are going to add and remove session from it concurrently.
      ```
      private static final Set<Session> sessions =
                      new CopyOnWriteArraySet<>();
      ```

g. Add the following imports:

```
java.util.Set
java.util.concurrent.CopyOnWriteArraySet
javax.websocket.Session
```

Note: The purpose of holding this set of sessions is to be able to broadcast messages to all chat users.

3. Add code to the ChatServer class to respond to WebSocket lifecycle callback operations:
   a. Declare an instance method called **onOpen** in the ChatServer class to be invoked when the client opens a connection with the WebSocket server endpoint. This operation should accept the **Session** parameter.

```
@OnOpen
public void onOpen(Session session) {
    // session open handling logic will be added there later
}
```

   b. Add an import: `javax.websocket.OnOpen`
   c. Declare an instance method called **onClose** in the ChatServer class to be invoked when client closes a connection with the WebSocket server endpoint. This operation should accept **Session** and **CloseReason** parameters.

```
@OnClose
public void onClose(Session session, CloseReason reason) {
    // session close handling logic will be added there later
}
```

   d. Add two imports: `javax.websocket.OnClose` and `javax.websocket.CloseReason`
   e. Declare an instance method called **onError** in the ChatServer class to be invoked when there is an error in socket communications. This operation should accept **Session** and **Throwable** parameters.

```
@OnError
public void onError(Session session, Throwable ex) {
    // session error handling logic will be added there later
}
```

   f. Add an import: `javax.websocket.OnError`
   g. Declare an instance method called **onMessage** in the ChatServer class to be invoked when this WebSocket server receives a message from the client. This operation should accept the **String message** parameter.

```
@OnMessage
public void onMessage(String message) {
    // session message handling logic will be added there later
}
```

h. Add an import: `javax.websocket.OnMessage`

4. Add a custom operation to send messages to all chat sessions:
   a. Declare an instance method called **broadcastMessage** in the ChatServer class to broadcast messages to chat sessions. This operation should accept the **String message** parameter.

```
public void broadcastMessage(String message) {
    // message broadcast logic will be added there later
}
```

Note: Methods annotated with OnOpen, OnClose, OnError, and OnMessage annotations represent standard WebSocket callback operations. The broadcastMessage method is your custom business method that does not correspond to any predefined WebSocket callback.

5. Add implementation logic to all operations of the ChatServer:
   a. Inside the **onOpen** operation, add code to perform the following actions:
      - Add a session object to the set of sessions.
      - Acquire an AsyncRemote object from this session and add a welcome text message to it.
      - Broadcast a message that informs all sessions that a new user has joined the chat.

```
sessions.add(session);
session.getAsyncRemote().sendText("Welcome!");
broadcastMessage("New user has joined the chat");
```

   b. Inside the **onClose** operation, add code to perform the following actions:
      - Remove a session object from the set of sessions.
      - Broadcast a message that informs all sessions that a user has left the chat.

```
sessions.remove(session);
broadcastMessage("User has left the chat");
```

   c. Inside the **onError** operation, add code to perform the following actions:
      - Write information about the error to the log.
      - Acquire an AsyncRemote object from this session and an error text message to it.

```
logger.log(Level.INFO,"WebSocket Error",ex);
session.getAsyncRemote().sendText(ex.getMessage());
```

d. Inside the **onMessage** operation, add code to perform the following actions:
   - Check if the message text is not empty.
   - If the message does not contain text, throw a runtime exception with an error message informing that there is no message text.
   - Otherwise broadcast this message to all sessions.

```
if (message == null|| message.length() == 0) {
     throw new RuntimeException("No actual message received");
}
broadcastMessage(message);
```

e. Inside the **broadcastMessage** operation, add code that iterates through the set of sessions. For each session object, acquire the AsyncRemote object and send a text message to it.

```
sessions.stream().forEach(s ->
s.getAsyncRemote().sendText(message));
```

f. Compile the **ProductWeb** project using **Clean and Build.**

Practice 10-2: Invoking WebSocket Chat Server by Using JavaScript

Overview

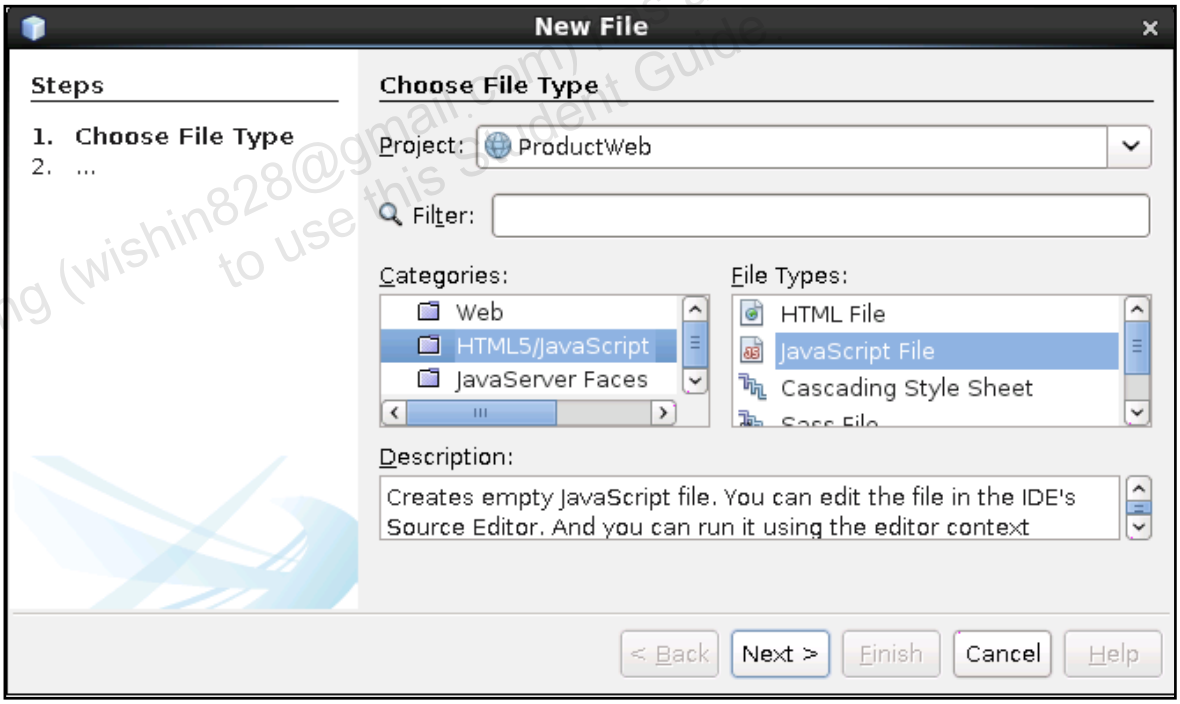
In this practice, you create a new chat HTML page with added JavaScript functionality to interact with WebSocket Chat Server. You modify the index.html page to add an invocation of the new chat page.

Assumptions

You have successfully completed all previous practices.

Tasks

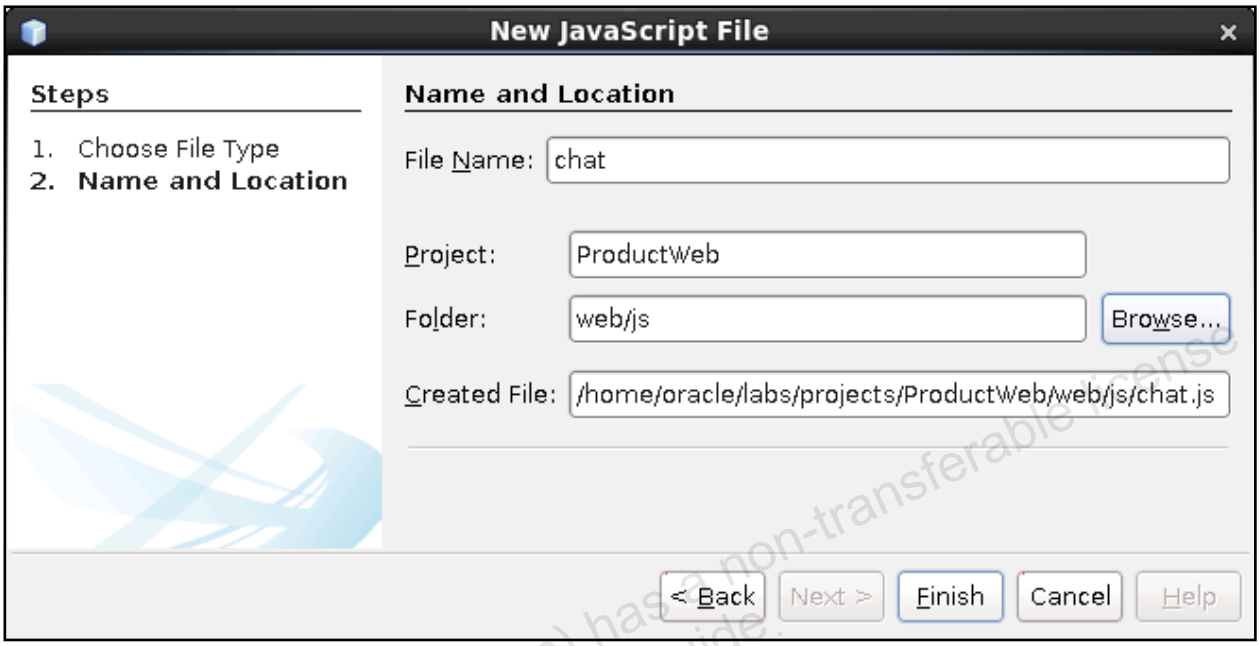
1. Add a JavaScript file to the ProductWeb project. This JavaScript file should contain code that interacts with WebSocket Chat Server that you have created in the previous practice.
   a. Select the **ProductWeb** project in NetBeans.
   b. Select **File > New File.**
   c. Select **HTML5/JavaScript** from the list of Categories and **JavaScript File** from the list of File Types.



   d. Click **Next.**

e. In the New JavaScript File dialog box, set the following properties:

- File Name: `chat`
- Folder: `web/js`



f. Click **Finish.**

g. Add JavaScript code to the `chat.js` file.

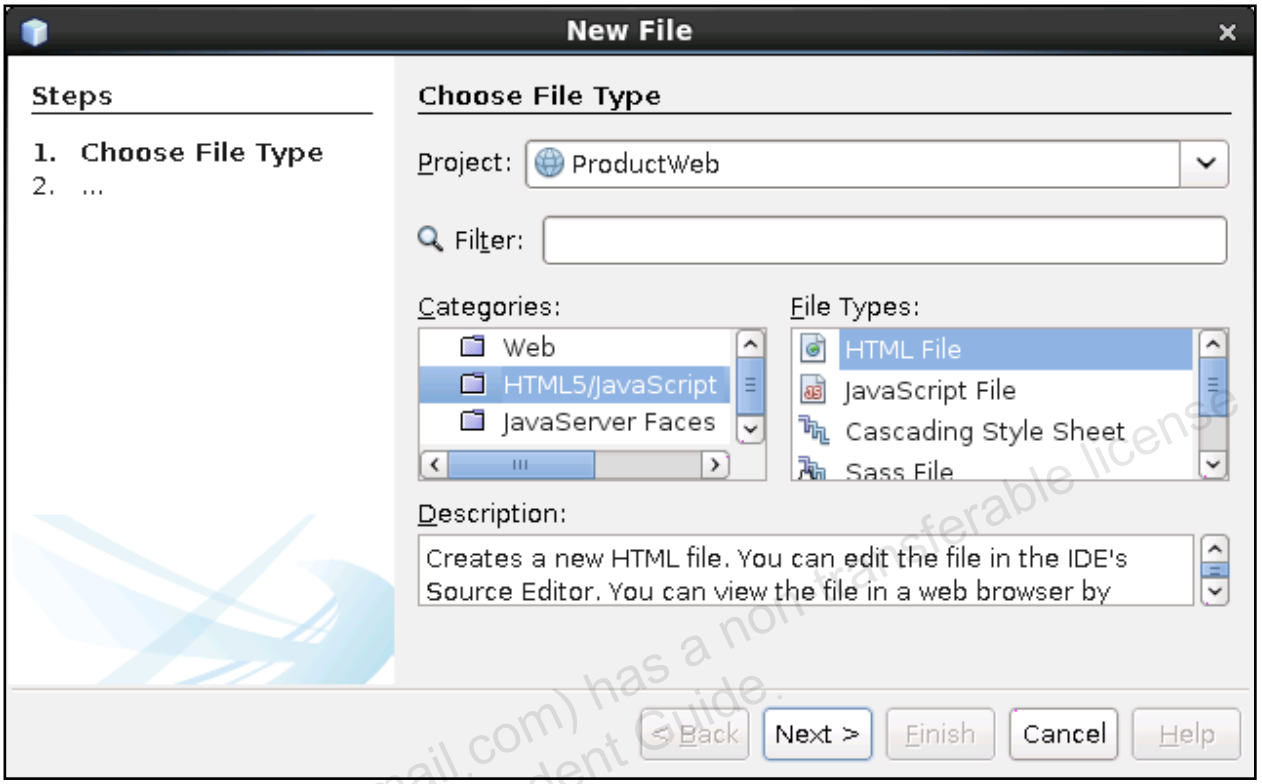
Note: You can simply replace the entire contents of the `chat.js` file with text from the `chat.js` file located in the `/home/oracle/labs/resources` folder.

Note: The JavaScript has comments to help you understand its logic.

Note: This course is about Java not JavaScript, which is an entirely different programming language. This is why you are given this script rather than being asked to write its code yourself. If you are interested in learning more about what this code does and how to program with JavaScript, Oracle University has a course *JavaScript and HTML5: Develop Web Applications*.

2. Create a new Chat HTML page:
   a. Select the **ProductWeb** project.
   b. Select **File > New File.**

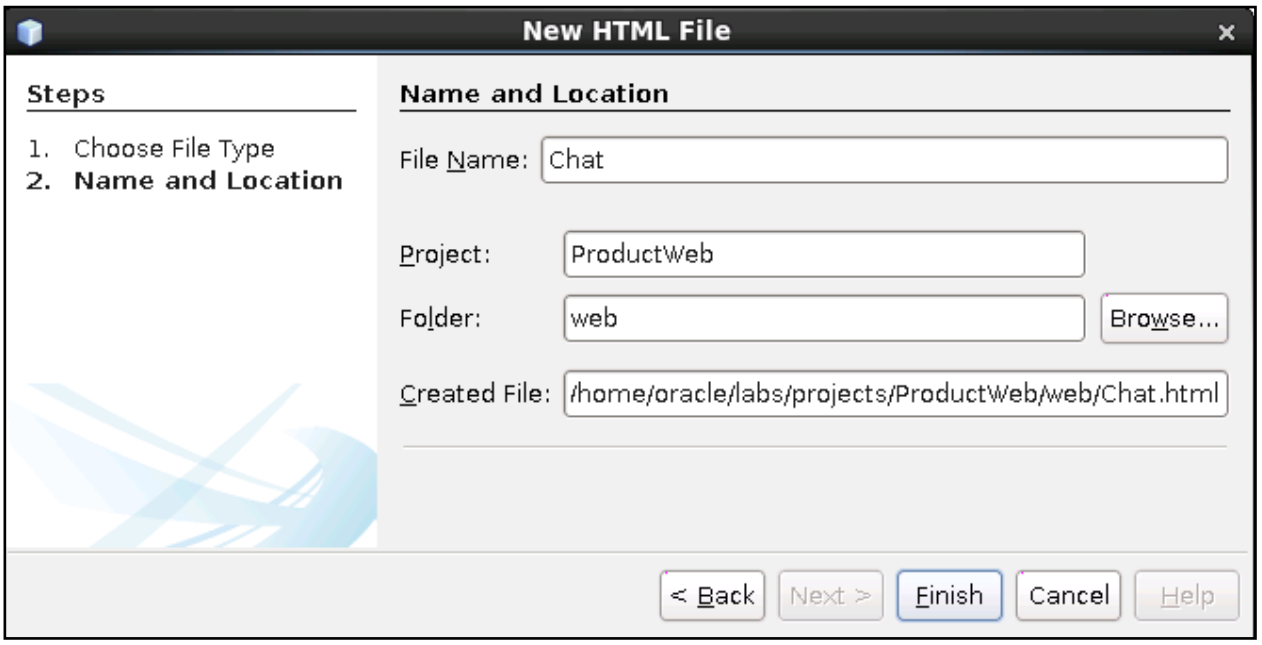
c. Select **HTML5/JavaScript** from the list of Categories and **HTML File** from the list of File Types.



d. Click **Next**..

e. In the New HTML File dialog box, set the following properties:

- File Name: `Chat`
- Folder: `web`



f. Click **Finish**.

3. Design the Chat page structure:
   a. Modify `Chat.html` code to define the page structure. Replace the entire contents of the `Chat.html` file with text from the `template.html` file located inside the `/home/oracle/labs/resources` folder.
   b. Attach the JavaScript file to the Chat page. Insert the following line in the **head** section just after the **link** element that attached **`chat.js`** file to this page.

```
<script type="text/javascript" src="/pm/js/chat.js"></script>
```

   c. Change the ProductSearch page title, header, navigation, and footer elements.
      - Replace the PAGE TITLE text with: `Chat`
      - Replace the PAGE HEADER text with: `Chat`
      - Replace the PAGE FOOTER text with: `Join, Send, Leave`
   d. Remove the entire **nav** element including the PAGE NAVIGATION text. Your page body should contain only header, section, and footer elements.

4. Replace the PAGE CONTENT text in the Chat page with elements that would be used to **display chat messages,** allow user to **enter new messages**, and **buttons to join chat, send messages and leave the chat**.
   a. Add a new `html` divider element with **id chat** inside the section element of the chat page. This element will serve as a target element into which a JavaScript function will dynamically add chat messages as they arrive from chat server.

```
<div class="field" id="chat"></div>
```

   b. Add another HTML divider element that contains a label and an input field for entering chat messages. Set the input field **id msg**, and make it disabled by default. This input field will be used by a JavaScript function to get messages that user want to send to the chat server. A JavaScript function that joins the chat will enable this field, and a function that closes the chat disables it again.

```
<div class="field">
    <label for="msg">Message:</label>
    <input type="text" id="msg" disabled>
</div>
```

c. Add one more HTML divider element that contains three buttons **ids join**, **send**, and **leave**, each defining an **onclick** event handler and invoking **joinChat()**, **sendMessage()**, and **leaveChat()** JavaScript functions. Only the Join button should be enabled by default. Send and Leave buttons will be enabled by a JavaScript function that joins the chat, and a function that closes the chat disables these buttons again.

```
<div class="field">
     <input type="button" id="join"
             value="Join" onclick="joinChat()">
     <input type="button" id="send"
              value="Send" onclick="sendMessage()" disabled>
     <input type="button" id="leave"
              value="Leave" onclick="leaveChat()" disabled>
</div>
```

5. Modify the `index.html` page to open chat windows:
   a. Open the **index.html** file located under the **Web Pages** section in the **ProductWeb** project.
   b. Inside the **nav** element, after the **a** element, add code that will invoke new chat windows.

```
<nav class='nav'>
     <a href='/pm/ProductSearch.html'>Product Search</a>
     ADD CODE HERE
</nav>
```

Note: The code example above helps you find a place inside the `index.html` page where you are going to add code that opens new chat windows.

   c. Add an inline JavaScript element that defines an **`openChat`** function that would open a new window that will contain the `Chat.html` page.

```
<script>
function openChat() {
var chatWindow = window.open('Chat.html',
                 '_blank', height=500, width= 500);
}
</script>
```

   d. Add a divider that contains a button that will invoke this function from the `onclick` event

```
<div class='field'>
     <input type="button" value="Open Chat Window"
                           onclick="openChat()">
</div>
```

6. Test the Chat Server:
    a. Compile the **ProductWeb** project using **Clean and Build.**
    b. Right-click the **ProductWeb** project and select **Deploy**.
    c. Once this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm`
    d. Click the **Open Chat Window** button.
    e. Click the **Open Chat Window** button again.
    f. Place two opened chat windows side-by-side.
    g. Click the **Join** button in both windows and start sending messages.
    h. Attempt to send an empty message in one of the windows.
    i. Click the **Leave** button in one of the windows, and click the Join button again.
    j. Notice that chat is transient and does not retain passed messages. This is because ChatServer did not try to persist any message.
    k. Leave the Chat Windows open and inactive to observe how a chat session will expire. You can change the max idle time interval of a WebSocket session in the ServerEndpoint class.

Practice 10-3: Invoking a WebSocket Chat Server by Using Java

Overview

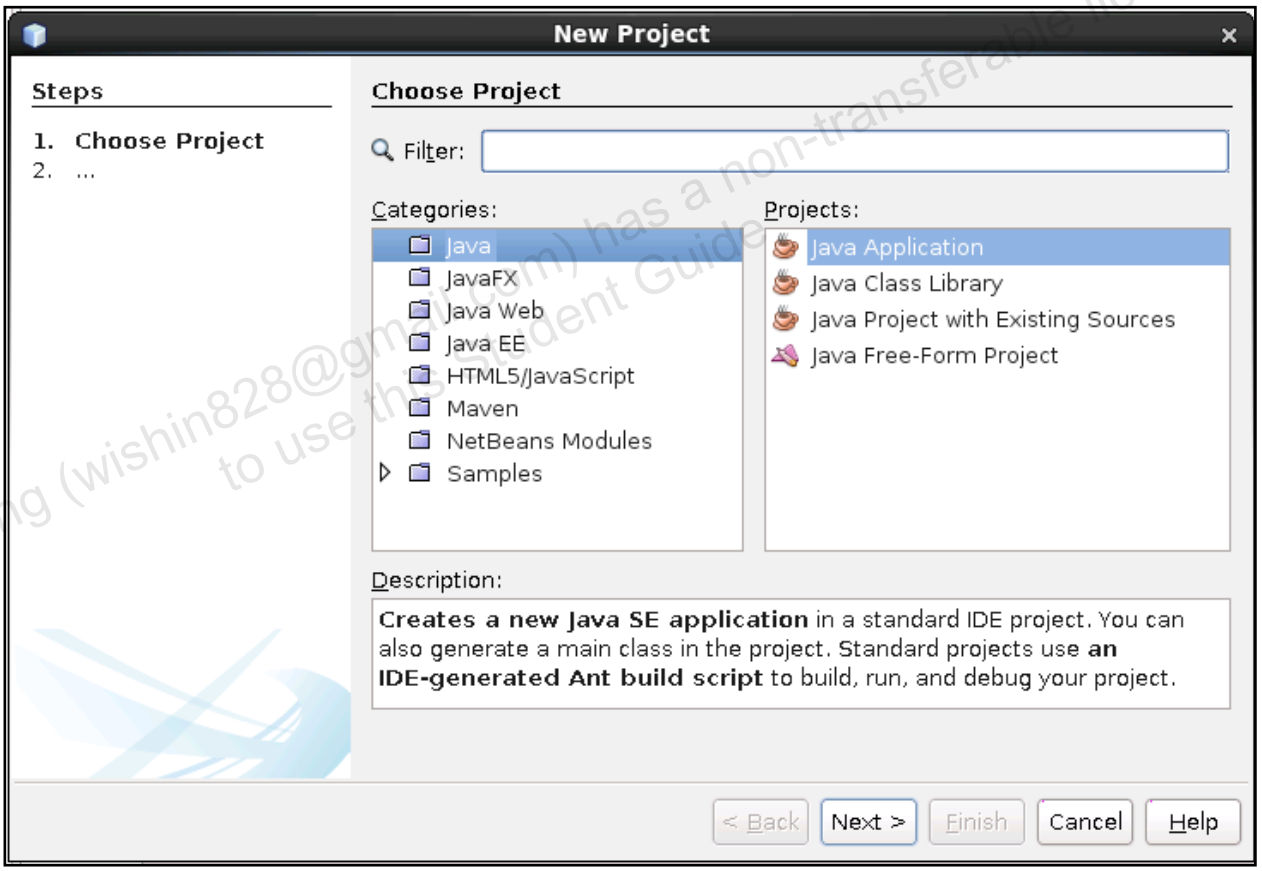
In this practice, you create a new Java Application that invokes Chat WebSocket Server.

Assumptions

You have successfully completed all previous practices.

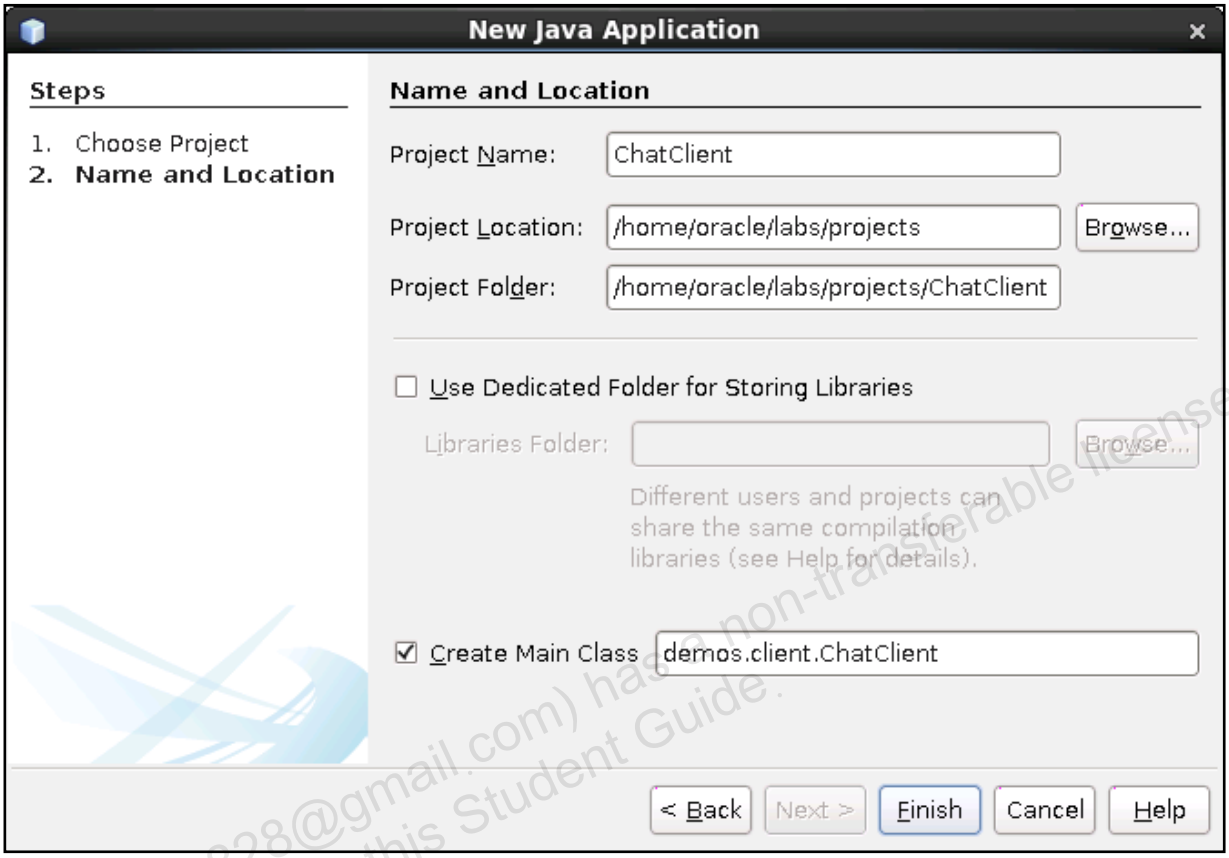
Tasks

1. Create a new Java Application project called ChatClient to invoke Chat WebSocket Server.
   a. Select **File > New Project**.
   b. Select **Java** from the list of Categories and **Java Application** from the list of Projects.



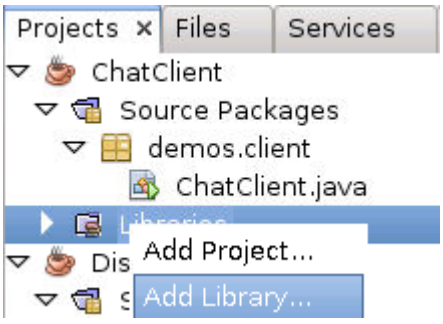
   c. Click **Next**.
   d. In the New Java Application dialog box, set the following properties:
      - Project Name: `ChatClient`
      - Project Location: `/home/oracle/labs/projects`
      - Project Folder: `/home/oracle/labs/projects/ChatClient`

- Select the **Create Main Class** check box and set the class name: `demos.client.ChatClient`

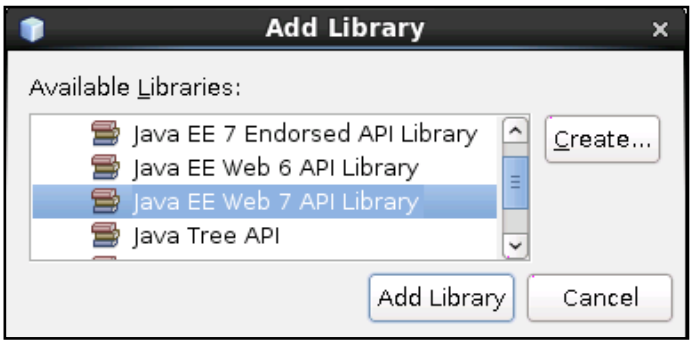


e. Click **Finish**.

2. Add required libraries to support the WebSocket Java API client of the ChatClient project.
   a. Expand the **ChatClient** Project and right-click the **Libraries** node.
   b. Select **Add Library**.



c. In the Add Library dialog box, select **Java EE Web 7 API Library**.



d. Click the **Add Library** button.

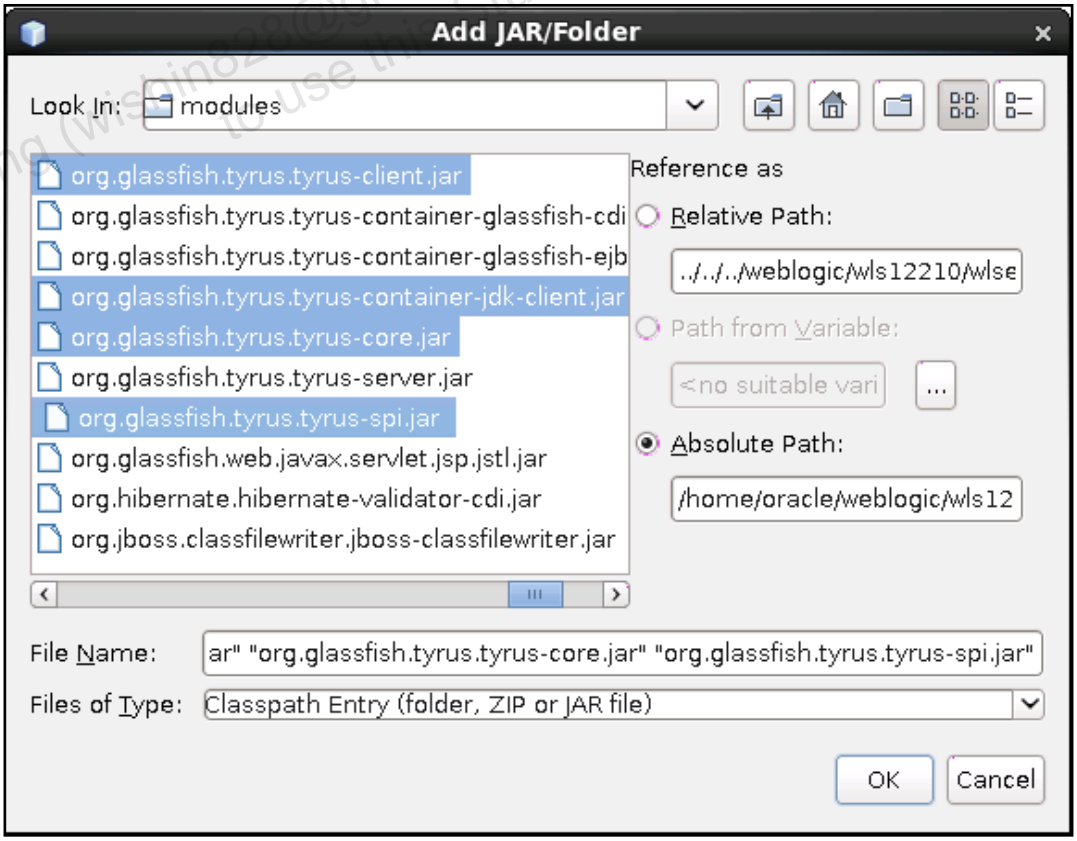
e. Right-click the **Libraries** node again.

f. Select **Add JAR/Folder**.

g. In the Add JAR/Folder dialog box, navigate to this folder:
`/home/oracle/weblogic/wls12210/wlserver/modules`

h. Select the following jar files:

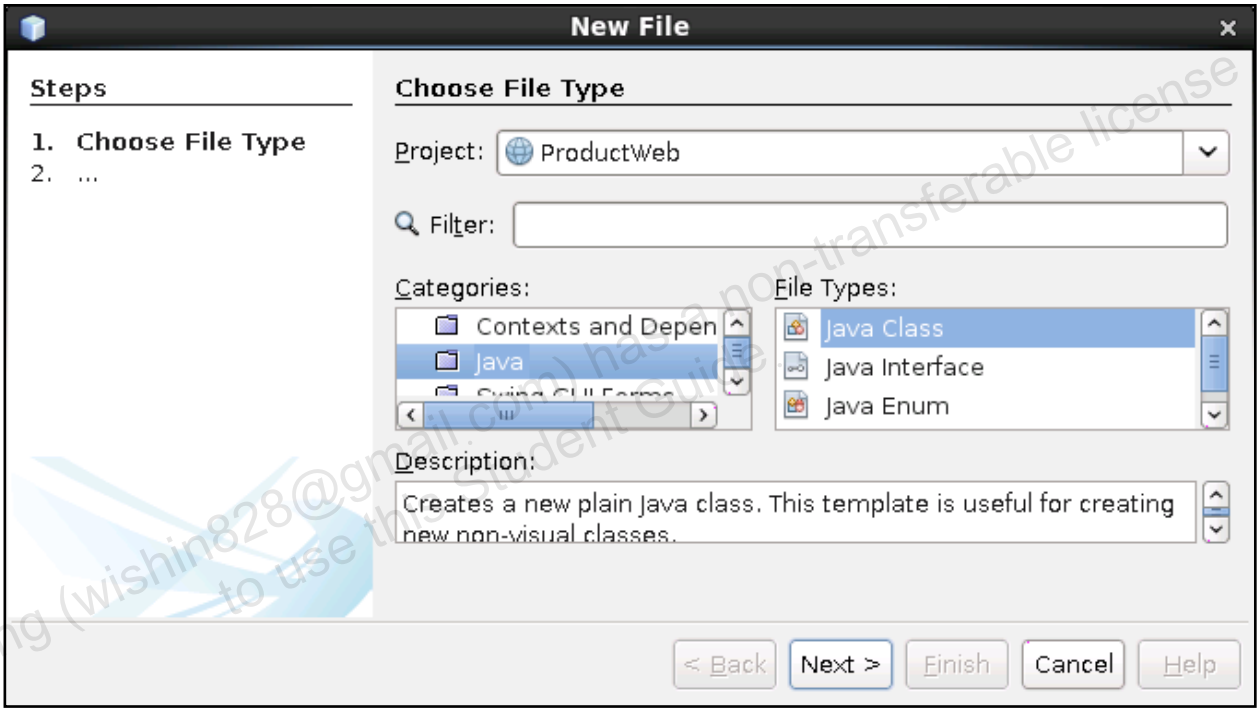
- `org.glassfish.tyrus.tyrus-spi.jar`
- `org.glassfish.tyrus.tyrus-core.jar`
- `org.glassfish.tyrus.tyrus-container-jdk-client.jar`
- `org.glassfish.tyrus.tyrus-client.jar`



i. Click **OK**.

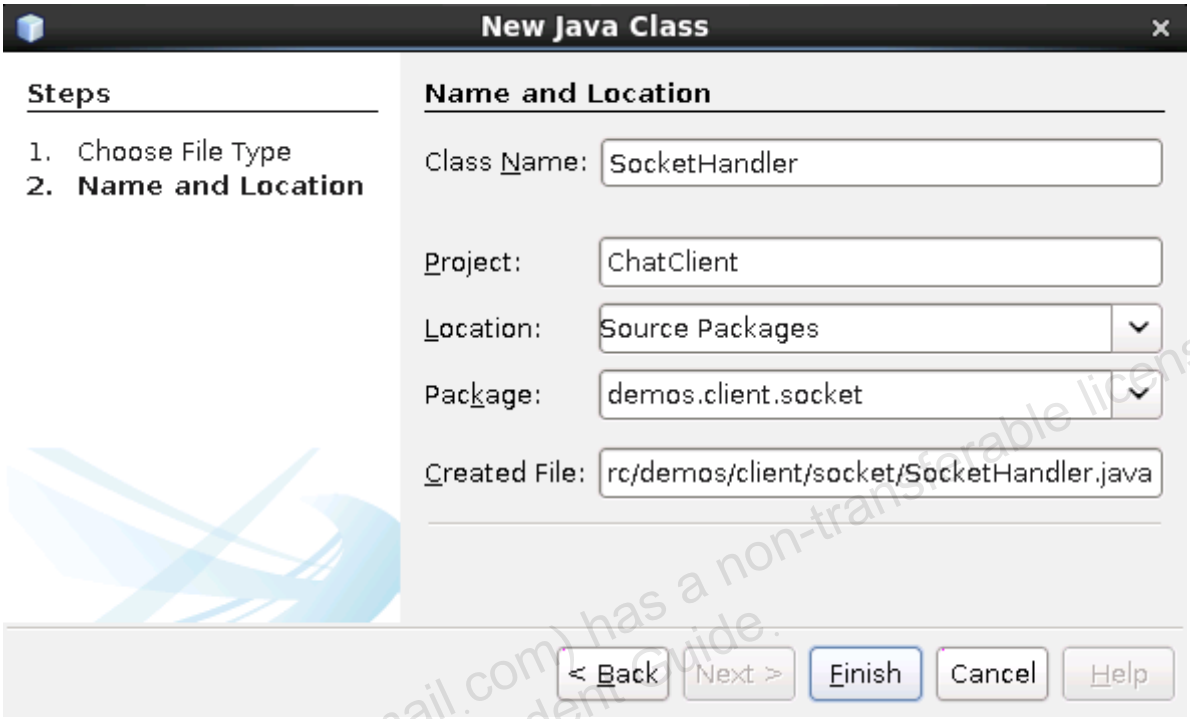
3. Create a new Java class to represent the WebSocket client endpoint. This class should handle web socket communications with the following functionalities:
   - Open and close sockets.
   - Receive messages and errors.

   a. Select the **ChatClient** project in NetBeans.
   b. Select **File > New File**.
   c. Select **Java** from the list of Categories and **Java Class** from the list of File Types.



   d. Click **Next**.

e. In the New Java Class dialog box, set the following properties:
   - Class Name: `SocketHandler`
   - Package: `demos.client.socket`



f. Click **Finish**.

4. Add code to SocketHandler class to prepare it to handle WebSocket communications:
   a. Open the **SocketHandler** class located in the **demos.client.socket** package.
   b. Annotate this class to be a WebSocket **ClientEndpoint**. This code should be added on a new line of code before the class declaration.
      ```
      @ClientEndpoint
      ```
   c. Add an import: `javax.websocket.ClientEndpoint`
   d. Inside the body of the SocketHandler class, declare and initialize a Logger reference to be used by this class.
      ```
      private static final Logger logger =
                      Logger.getLogger(SocketHandler.class.getName());
      ```
   e. Add an import: `java.util.logging.Logger`

5. Add code to the SocketHandler class to respond to WebSocket life-cycle callback operations:
   a. Declare an instance method called **onOpen.** This operation should accept the **Session** parameter. It should write a message to the log indicating that the session has been opened.

```
@OnOpen
public void onOpen(Session session) {
     logger.log(Level.INFO,"WebSocket session started");
}
```

   b. Add the following imports:

```
javax.websocket.OnOpen
java.util.logging.Level
javax.websocket.Session
```

   c. Declare an instance method called **onClose.** This operation should accept the **Session** and **CloseReason** parameters. It should write a message to the log indicating that the session has been closed, and a reason for the closure.

```
@OnClose
public void onClose(Session session, CloseReason reason) {
     logger.log(Level.INFO,"WebSocket session ended "
                     +reason.getReasonPhrase());
}
```

   d. Add two imports: `javax.websocket.OnClose` and `javax.websocket.CloseReason`
   e. Declare an instance method called **onError.** This operation should accept the **Session** and **Throwable** parameters. It should write a message to the log indicating that the session has an error, and the exception message.

```
@OnError
public void onError(Session session, Throwable ex) {
     logger.log(Level.INFO,"WebSocket session errored",
               ex.getMessage());
}
```

   f. Add an import: `javax.websocket.OnError`
   g. Declare an instance method called **onMessage.** This operation should accept the **String message** parameter. It should write a message to the log that it has received a message.

```
@OnMessage
public void onMessage(String message) {
     logger.log(Level.INFO,">"+message);
}
```

   h. Add an import: `javax.websocket.OnMessage`

6. Add code to ChatClient to handle Chat functionalities:
   a. Open the **ChatClient** class located in the `demos.client` package.
   b. In the body of the ChatClient class, declare and initialize a Logger reference to be used by this class.

```
private static final Logger logger =
                Logger.getLogger(ChatClient.class.getName());
```

   c. Add an import: `java.util.logging.Logger`
   d. Inside the **main** method of the **ChatClient** class, print out there messages with help information on how to use this chat:

```
System.out.println("Click inside the log window");
System.out.println("Type your message and press Enter");
System.out.println("Type exit to close chat");
```

   e. After these print statements create a `try` / `catch` block. Add a generic exception handler that writes errors to the log.

```
try {
     // add here chat client code
} catch (Exception ex) {
     logger.log(Level.SEVERE,"Error Accessing Chat",ex);
}
```

   f. Add an import: `java.util.logging.Level`

7. Add code to ChatClient to open a new WebSocket session and to prepare to send and receive messages:
   a. Inside the `try` block, declare and initialize a new URI object that points to the WebSocket ChatServer.

```
URI uri = new URI("ws://localhost:7001/pm/chat");
```

   b. Add an import: `java.net.URI`
   c. Continue inside the `try` block and on the next line declare and initialize a `WebSocketContainer` object using the `ContainerProvider` class.

```
WebSocketContainer container =
          ContainerProvider.getWebSocketContainer();
```

   d. Add two imports: `javax.websocket.ContainerProvider` and `javax.websocket.WebSocketContainer`
   e. Open a new WebSocket Session using the container object `connectToServer` method. Notice that this method accepts an instance of the `ClientEndpoint` class — in your case, that is `SocketHandler` and a URI object pointing to the corresponding ServerEndpoint location.

```
Session session = container.connectToServer(new SocketHandler(), uri);
```

f. Add two imports: `javax.websocket.Session` and `demos.client.socket.SocketHandler`

g. Acquire the `RemoteEndpoint` object to be able to send messages to the `ServerEndpoint`.

```
RemoteEndpoint.Async remote = session.getAsyncRemote();
```

h. Add an import: `javax.websocket.RemoteEndpoint`

8. Read messages from the console:

a. Inside the `try` block after the line of code that acquired the `RemoteEndpoint` object, create and initialize a Scanner object to read text from the console.

```
Scanner s = new Scanner(System.in);
```

b. Add an import: `java.util.Scanner`

c. In a `while` loop, check each line acquired from the console. If this text is a word, 'exit' (terminate the loop). Send text acquired from the console as a message to the chat.

```
while (s.hasNextLine()) {
   String message = s.nextLine();
    if (message.equals("exit")) {
         break;
    }
    remote.sendText(message);
}
```

9. Close WebSocket session:

a. After the while loop end, at the end of the `try` block, add code that closes the session.

```
session.close(new CloseReason(CloseReason.CloseCodes.NORMAL_CLOSURE,
                              "Leaving the chat"));
```

b. Add an import: `javax.websocket.CloseReason`

10. Test the ChatClient application:

a. Compile the **ChatClient** project using **Clean and Build**.

b. Run the ChatClient project.

c. Also open a browser and navigate to the following URL: `http://localhost:7001/pm`

d. Click the **Open Chat Window** button.

e. Click the **Join** chat button in the browser window and send a message.

f. Return to NetBeans and observe received messages logged on to the console.

g. Click inside the log window, type some message, and press Enter.

h. Check that the message appeared in the browser chat window.

i. To close the Chat Client, click inside the log window, type **exit**, and press Enter.

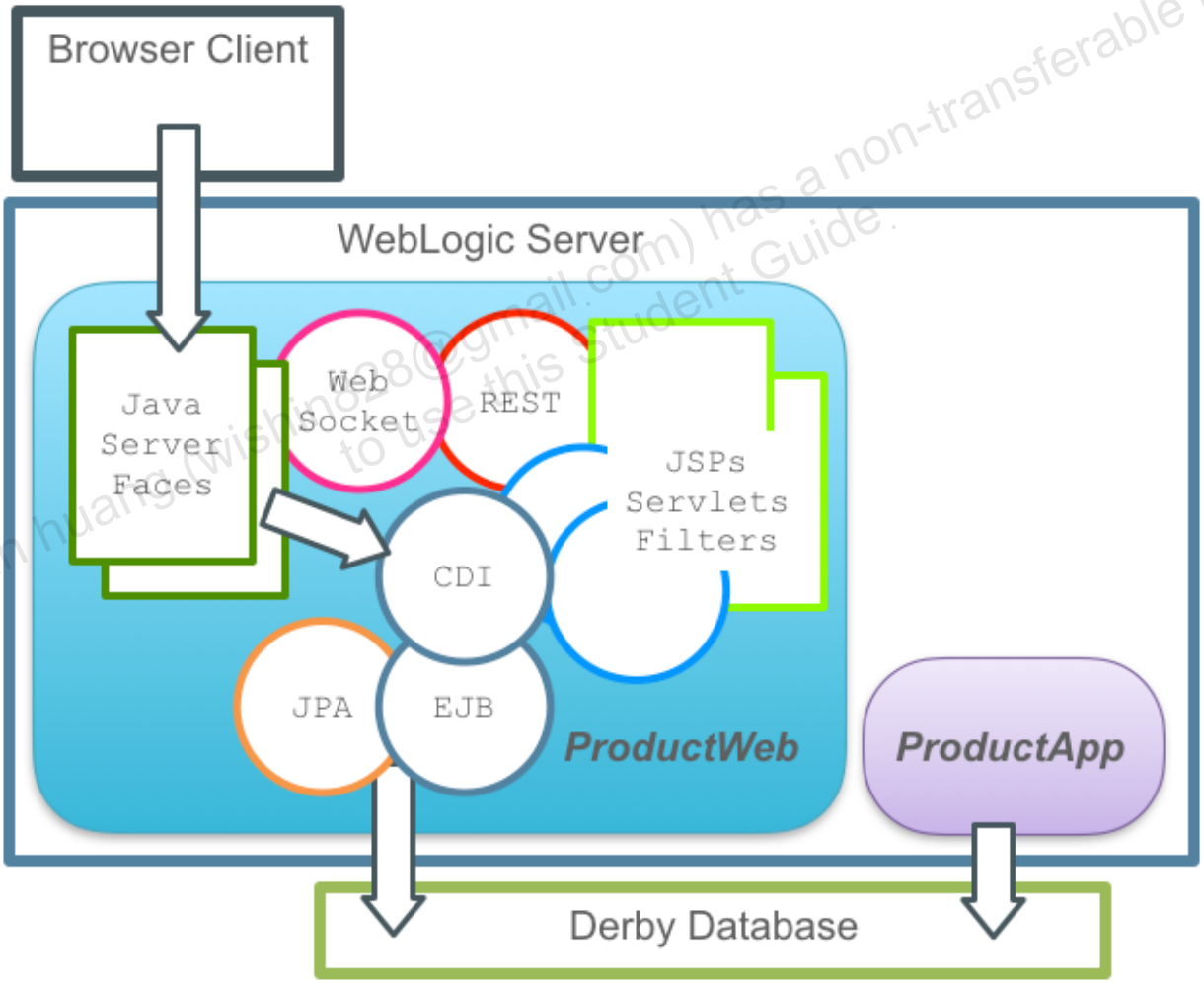
Practices for Lesson 11: Developing Web Applications Using JavaServer Faces

Practices for Lesson 11: Overview

Overview

In these practices, you create Java Server Faces pages to produce a product search form, a list of products, and a product edit form. You also create additional CDI bean operations to handle JSF actions and events and a custom data conversion class.

During this practice, you will design a JSF application that performs the same stages as the existing JSP application. However, you will have to write way less "plumbing" code to achieve same functionalities. Instead of writing your own Servlet or WebFilter code, as you did in earlier practices, you will rely upon JSF Runtime to perform handling of application events, maintaining application state, and managing application lifecycle.



Practice 11-1: Adding JSF Action and Event Handling

Overview

In this practice, you add JSF event-handling operations to the `ProductManager` CDI Bean. You will have to provide three operations to the `ProductManager` bean to support the following actions:

- Search product by name
- Selecting product from list
- Product update

You also create a `DateConverter` class to handle conversion between String and LocalDate types. JSF libraries contain an existing converter that handles conversion between String and Date. You are going to extend this existing converter.

Assumptions

You have successfully completed all previous practices.

Tasks

1. Add a String name variable to the `ProductManager` class to act as a value holder for the field name of the product search form:
   a. Select the **ProductWeb** project in NetBeans.
   b. Open the **`ProductManager`** class located in the **`demos.model`** package.
   c. Add a new String instance variable called `name`.
   ```
   private String name;
   ```
   d. Create getter and setter operations for the instance variable name.
   Hint: Scroll to the end of the `ProductManager` file and, just before the end of the class, add a new line of code inside this class body. Right-click this new line of code and select **Insert Code > Getter and Setter**. Then select the check box for the **name** field and click the **Generate** button.

2. Add a JSF action method to handle products search and navigation from the Search page to the List page:
   a. Inside the `ProductManager` class, after the `get`/`set` pair of operations for the **name** field, add a new method called **`showList`** that returns a **String** object.
   ```
   public String showList() {
       // product search logic will be added here
   }
   ```
   b. This operation should use the variable `name` to execute a product search with the findProductByName operation.
   ```
   products = productFacade.findProductByName(name);
   ```

c. Check if the list of products is empty. If no products were found, you need to produce a new `FacesMessage` warning message and let the Search page render its response again; otherwise, you return navigation rule named **"list"**.

Note: You will configure JSF navigation rules later in this practice.

```
if (products.isEmpty()) {
   FacesMessage message = new FacesMessage(FacesMessage.SEVERITY_WARN,
                          "No products matching'"+name+"' found",
null);
   FacesContext.getCurrentInstance().addMessage(null, message);
   return null;
}
return "list";
```

d. Add the following imports:

```
javax.faces.context.FacesContext
javax.faces.application.FacesMessage
```

3. Add a JSF action method to handle products selection in the List page and navigation from the List page to the Edit page:

a. Add another operation to the `ProductManager` class called **`showEdit`** to handle navigation from the List page to the Edit page. This operation should return a String navigation rule name. The name of the navigation rule you should return is **`"edit"`**.

```
public String showEdit(){
     // product selection logic will be added here
     return "edit";
}
```

b. This time you will try a different way of passing values from the page to the CDI bean — using request parameter, rather than binding to an instance variable in the bean. To get request parameters, you need to get an `ExternalContext` object; it can be acquired via `FacesContext`. You are looking for a parameter called **`"p_id"`** that will contain the product ID value of the product you want to select to display on the edit page. You will have to convert the String product ID value to Integer, because all http values are transmitted as text.

Note: `ExternalContext` gives you access to a Servlet runtime environment, and with that you can access request parameters.

```
Integer id = Integer.parseInt(FacesContext.getCurrentInstance()
                .getExternalContext()
                .getRequestParameterMap().get("p_id"));
```

c. After you have acquired product ID, before you return navigation rule name, add a line of code that finds a product with this ID in the list of products, and set this product to an instance variable.

```
product = products.stream()
          .filter(p->p.getId().equals(id))
          .findFirst().get();
```

Note: Alternatively, you could have selected this product from the database by executing the `findProduct` operation.

4. Add a JSF Action Listener operation to handle product update:
   a. To the `ProductManager` class, add a new void method called **`handleUpdate`** that accepts **`ActionEvent`** as an argument.

```
public void handleUpdate(ActionEvent event) {
     // product update logic will be placed here
}
```

   b. Add an import: `javax.faces.event.ActionEvent`
   c. Inside the `handleUpdate` operation, create a new variable to hold `FacesMessage`.

```
FacesMessage message;
```

   d. Next add a `try` / `catch` block, where `catch` should handle all exceptions.

```
try {
     // product update logic will be placed here
}catch(Exception ex) {
     // product update error handling will be placed here
}
```

   e. Add product update logic to the `try` block. You should call the `update` operation upon `ProductFacade`. Then using `ProductFacade`, fetch this product from the database again (it is possible that a product record can be modified by the database code such as triggers). You also need to construct a new FacesMessage to indicate that product was updated successfully.

```
productFacade.update(product);
product = productFacade.findProduct(product.getId());
message = new FacesMessage(FacesMessage.SEVERITY_INFO,
                           "Product updated successfully", null);
```

f. Add logic to the `catch` block. You should check what the cause of the exception was. If it was an `OptimisticLock`, produce `FacesMassage` indicating that the product was changed by another user; otherwise, you can simply throw an exception to this operation.

```
Throwable cause = ex.getCause();
if (cause instanceof OptimisticLockException){
     message = new FacesMessage(FacesMessage.SEVERITY_ERROR,
                  "Product has been changed by another
user",null);
}else{
     throw ex;
}
```

Note: An unhandled exception will trigger an invocation of the `ErrorHandler` servlet you already have in your project.

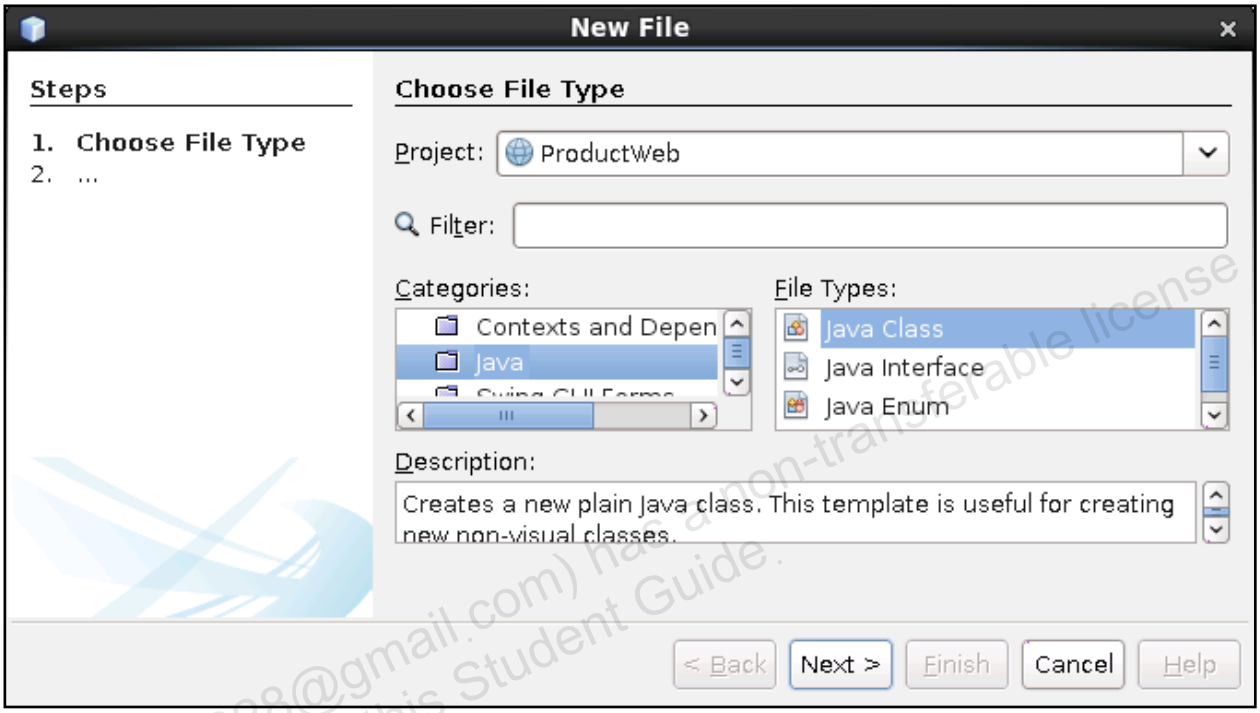
g. Add an import: `javax.persistence.OptimisticLockException`

h. After the end of the `catch` block, just before the end of this method, add a line of code that adds your `FacesMessage` to `FacesContext`, so it can be displayed on your page.

```
FacesContext.getCurrentInstance().addMessage(null, message);
```

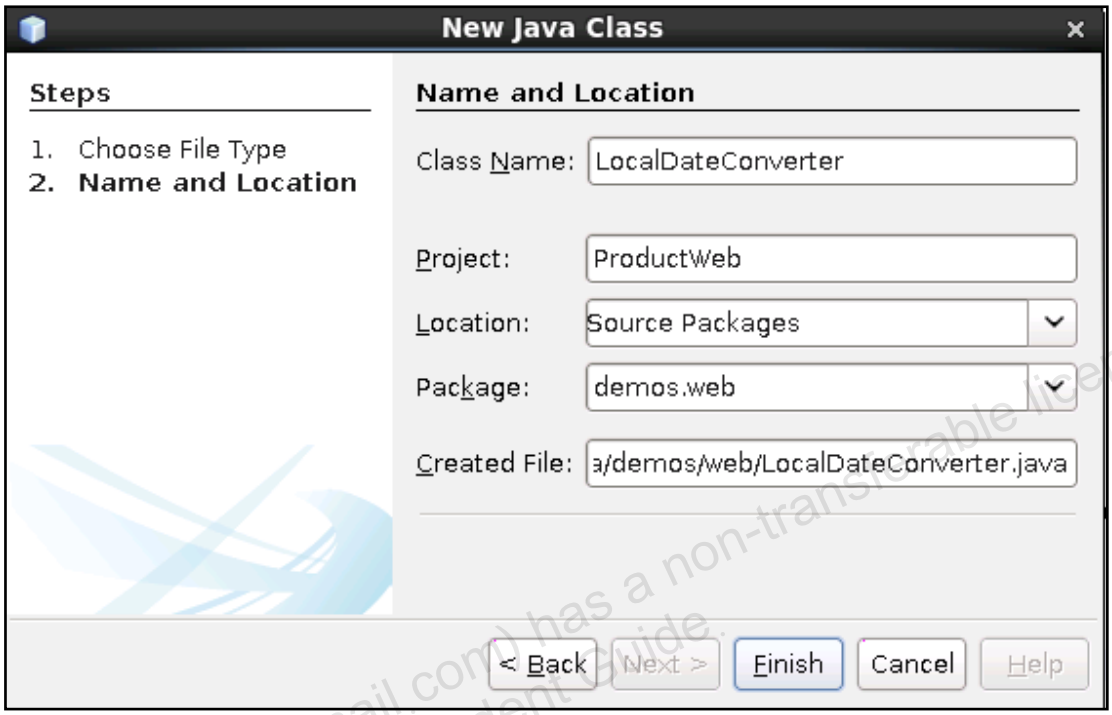
i. Compile **ProductWeb** project using **Clean and Build.**

5. Create a new Java class to represent `DateConverter` to handle conversion between String and LocalDate types:
   a. Select the **ProductWeb** project in NetBeans.
   b. Select **File > New File.**
   c. Select **Java** from the list of Categories and **Java Class** from the list of File Types.



   d. Click **Next.**

e. In the New Java Class dialog box, set the following properties:
   - Class Name: `LocalDateConverter`
   - Package: `demos.web`

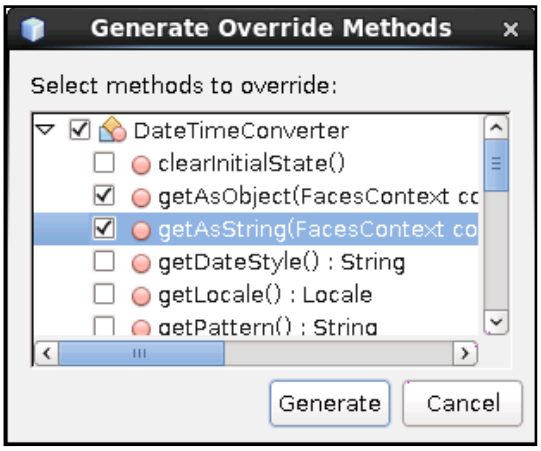


f. Click **Finish.**

6. Add code to the `LocalDateConverter` class to handle conversions between String and LocalDate types:
   a. Open the **`LocalDateConverter`** class located in the **demos.web** package.
   b. Add a line of code before the class definition and place there an annotation that will designate this class as a JSF value converter.
      ```
      @FacesConverter("LocalDateConverter")
      ```
   c. Add an import: javax.faces.convert.FacesConverter
   d. Modify the **`LocalDateConverter`** class definition to extend the existing JSF value converter class called **`DateTimeConverter`**.
      ```
      public class LocalDateConverter extends DateTimeConverter {
      }
      ```
   e. Add an import: `javax.faces.convert.DateTimeConverter`

f. Right-click an entry line of code inside the body of the `LocalDateConverter` class and select **Insert Code > Override Method**.

g. In the Generate Override Methods dialog box, select the check boxes for two methods: **`getAsObject`** and **`getAsString`**.



h. Click **Generate**.

i. Replace code inside the **getAsString operation**. Your code should perform the following actions:

- Set "yyyy-MM-dd" as the date pattern.
- Prepare the Date object.
- Check if the value is null and assign null to the Date object without attempting to perform any value conversion.
- Otherwise, convert LocalDate to Date type and assign it to your Date variable.
- Invoke the `super.getAsString` operation, passing your converted Date object to it.
- Return the result of the super.`getAsString` call.

Note: Because you have extended the existing `DateTimeConverter`, which already has an operation to convert Date objects to String objects, all you need to do is to perform a conversion of LocalDate to Date and let the existing code of the DateTimeConverter do the rest.

```
@Override
public String getAsString(FacesContext context,
                          UIComponent component, Object value) {
   super.setPattern("yyyy-MM-dd");
   Date date = (value == null) ? null : Date.from(((LocalDate)value)
                  .atStartOfDay(ZoneId.systemDefault()).toInstant());
   return super.getAsString(context, component, date);
}
```

j. Replace the code inside the **`getAsObject`** operation. Your code should perform the following actions:

- Set "yyyy-MM-dd" as the date pattern.
- Get the value from the JSF UI component that should contain the date object. This could be done by invoking the `super.getAsObject` operation.
- Check if this value is null. If it is null, then your method should return null; otherwise, perform a conversion of Date to LocalDate and return the result.

Note: Because you have extended the existing `DataTimeConverter` and it already has an operation to convert a String value into a Date object, you can invoke this operation and then convert Date to LocalDate.

```
@Override
public Object getAsObject(FacesContext context,
                          UIComponent component, String value) {
   super.setPattern("yyyy-MM-dd");
   Object obj = super.getAsObject(context, component, value);
   LocalDate date = (obj == null) ? null :
                    Instant.ofEpochMilli(((Date)obj)
                    .getTime()).atZone(ZoneId.systemDefault())
                    .toLocalDate();
return date;
}
```

k. Add the following imports:

```
java.time.Instant
java.time.LocalDate
java.time.ZoneId
java.util.Date
```

7. Compile the **ProductWeb** project using **Clean and Build**.

Practice 11-2: Creating JSF Pages

Overview

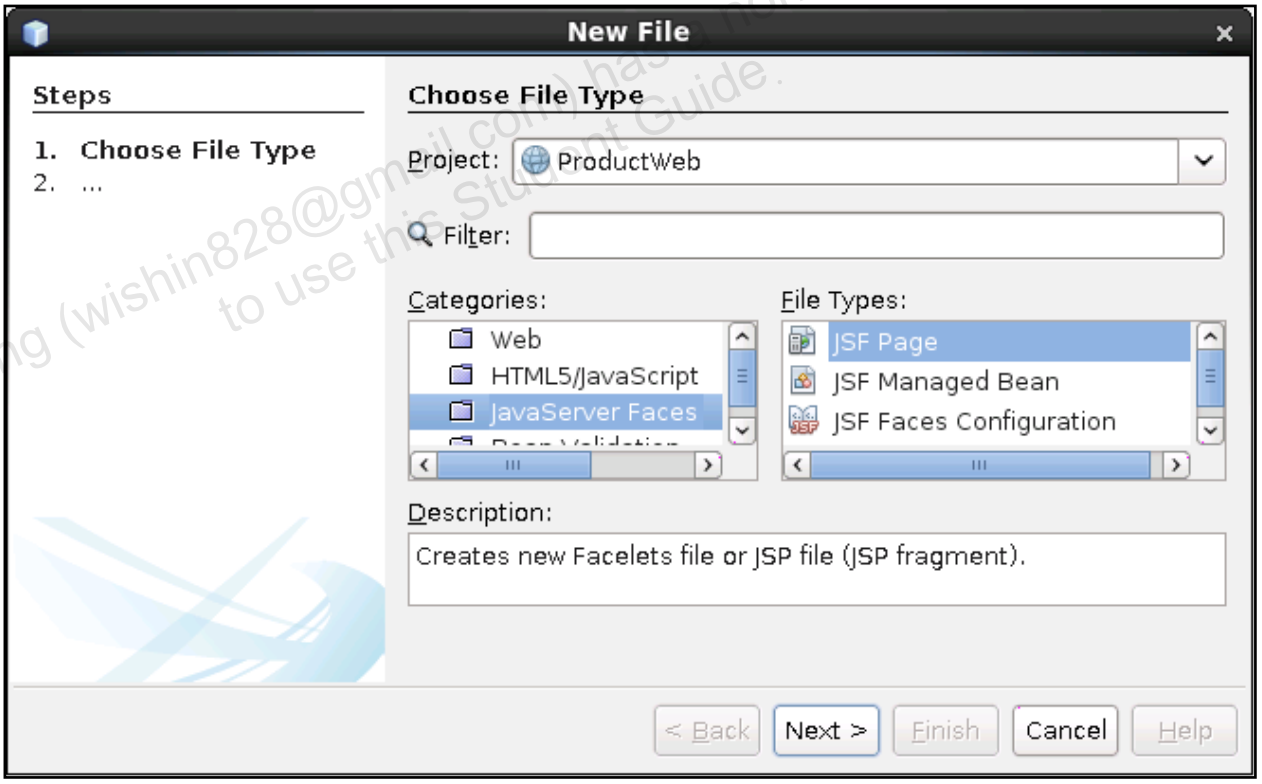
In this practice, you create JSF pages to display a product search form, a list of products, and a product update form. Define the JSF navigation rules. Modify the `index.html` page to add navigation to the product search JSF page.

Assumptions

You have successfully completed all the previous practices.

Tasks

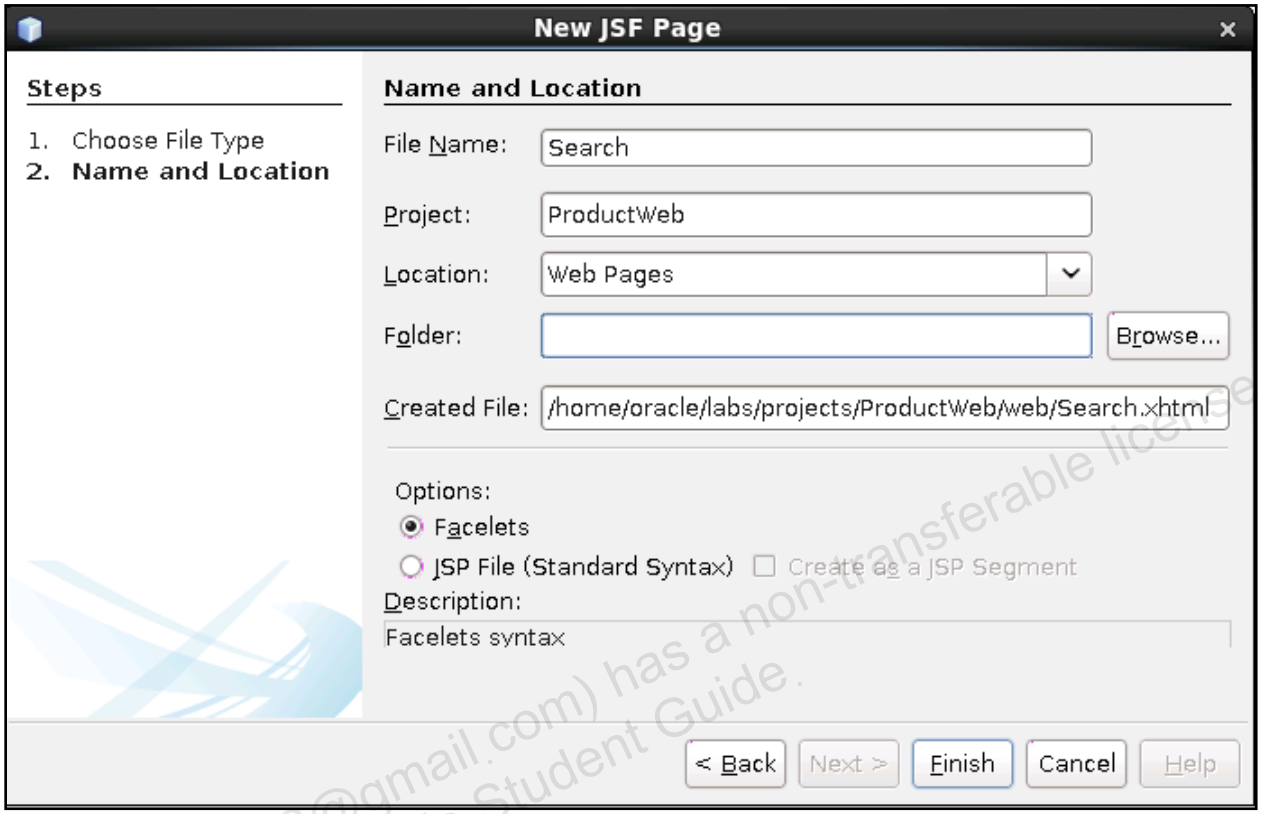
1. Create a new Java Server Faces page to display the product search form:
   a. Select the **ProductWeb** project in NetBeans.
   b. Select **File > New File**.
   c. Select **JavaServer Faces** from the list of Categories and **JSF Page** from the list of File Types.



   d. Click **Next**.

e. In the New JSF Page dialog box, set the following properties:

- File Name: `Search`



f. Click **Finish**.

2. Correct the NetBeans JSF page generation bug.

a. Open the **`web.xml`** file located in the **ProductWeb** project under **Web Pages > WEB-INF** folder.

b. Remove the entire **welcome-file-list** element from it.

Note: When you created a JSF page in this project, NetBeans generated a welcome-file-list entry in the `web.xml` file, pointing to a `faces/index.xhtml` file. However, such a file does not exist in your project. So an attempt to deploy this project is going to fail, unless this reference is removed from this `web.xml` file.

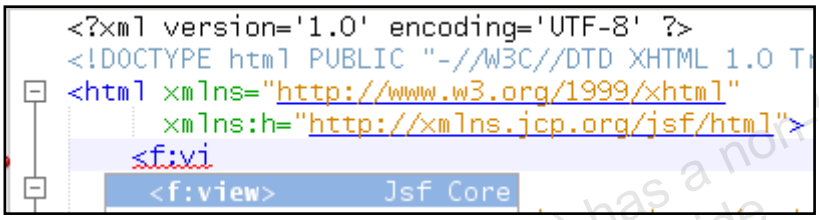
3. Define the `Search.xhtml` JSF file structure:

   a. Open the **`Search.xhtml`** file.

   b. Place an **f:view** element around the entire page contents to make the page transient. This element should start after the end of the `html` element and must be closed just before the closure of `html` element.

```
<html ...>
     <f:view transient="true">
          THE REST OF THIS PAGE CODE IS HERE
     </f:view>
</html>
```

   Hint: The **`view`** element is described by the **JSF core** tag library. Start typing **`<f:vi`** and when the drop-down list of suggested elements appears, press **Enter**. This will automatically add a library reference to the root element of this page.



   This is the tag library reference that your page should now have in the `html` element:

```
xmlns:f="http://xmlns.jcp.org/jsf/core"
```

   c. Replace the content inside the **h:head** element to define the Find Products page title and provide a link to the `pm.css` file.

   Hint: You can copy this code from the `ProductSearch.html` file, but you have to add a **`/`** symbol at the end of the link element, because this page is an XTHML file and must conform to XML as well as HTML standards.

   This is what the resulting code should look like:

```
<h:head>
<link rel='stylesheet' type='text/css' href='/pm/css/pm.css'/>
<title>Find Products</title>
</h:head>
```

   d. Replace the contents inside the **h:body** element with a JSF **form** component.

```
<h:body>
<h:form>
</h:form>
</h:body>
```

e. Inside the **form** component, display **`nav`**, **`section`**, and **`footer`** elements copied from the `ProductSearch.html` file. Do not copy any content from inside the section element.

This is what the resulting code should look like:

```
<h:form>
     <header class='header'>Find Products</header>
     <nav class='nav'><a href="/pm">Home</a></nav>
     <section class='content'>
          REMOVE ALL EXISTING CODE FROM HERE
     </section>
     <footer class='footer'>Enter product name. For partial
product name use % as wildcard.</footer>
</h:form>
```

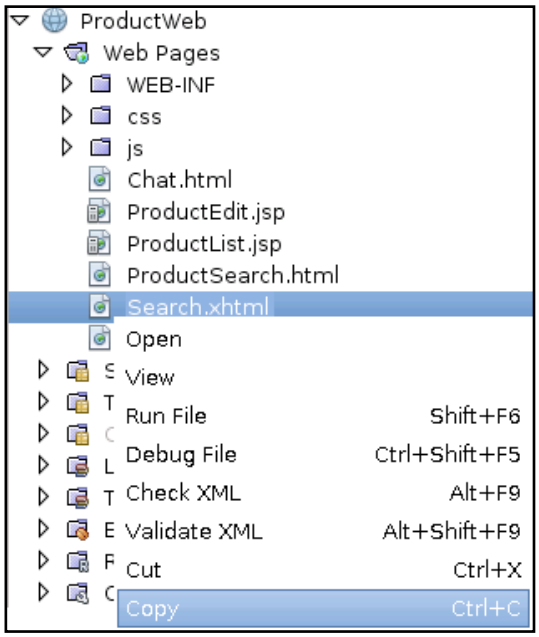
f. Inside the **`section`** element, add a **messages** component to display `FacesMessages`. Add CSS class properties to this component to use different styles for information, warning, and error messages.

```
<h:messages infoClass="info" errorClass="error"
warnClass="warn"/>
```

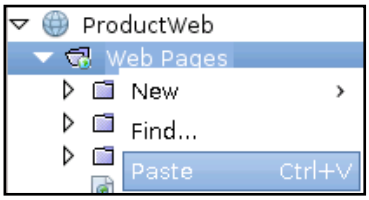
Note: Earlier, in the ProductManager bean you created code to produce `FacesMessages`; the page that you have created now has the capability to display these messages.

4. Create `List.xhtml` and `Edit.xhtml` files to represent the product list and the product update pages.

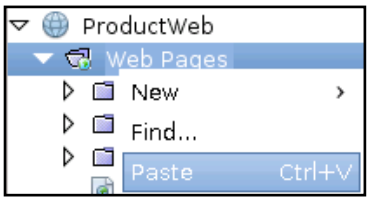
a. Right-click **Search.xhtml** and select **Copy**.



b. Right-click the Web Pages node and select **Paste**.



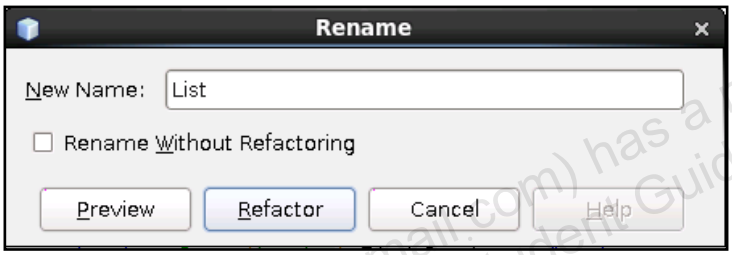
c. Right-click the Web Pages node and select **Paste** again.



Note: You just created two copies of the Search page called Search_1 and Search_2.

d. Right-click the **Search_1.xhtml** and select **Refactor > Rename**.

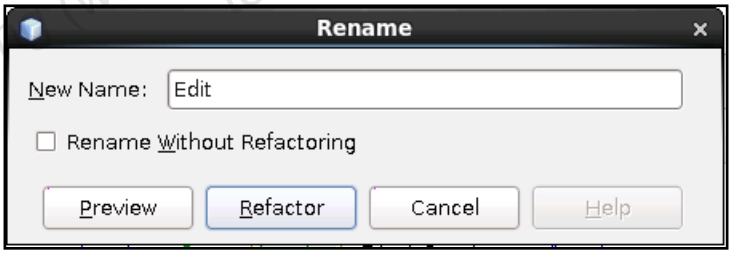
e. In the **Rename** dialog box, enter `List` as the new name.



f. Click the Refactor button.

g. Right-click **Search_2.xhtml** and select **Refactor > Rename**.

h. In the Rename dialog box, enter `Edit` as the new name.



i. Click the **Refactor** button.

5. Add components to the `Search.xhtml` file to display the search form elements in a panel grid layout:

   a. Open the `Search.xhtml` file.

   b. Inside the **`section`** element, after the **`messages`** component, add a **`panelGrid`** component to produce a table with two columns.

```
<h:panelGrid columns="2">
</h:panelGrid>
```

c. Inside the **panelGrid**, add an **outputLabel** component to display a **Product name:** prompt for the text input item that will represent product name.

```
<h:outputLabel for="pname" value="Product name:"/>
```

d. After the **outputLabel** component, add an **inputText** component. Bind the value of this input item to the property name of the ProductManager CDI bean. Make this a required item.

```
<h:inputText id="pname" value="#{pm.name}" required="true"
                requiredMessage="Please specify product name"/>
```

Note: Any user visible text, including error messages can be picked up by the JSF runtime from resource bundles described in faces-config.xml.

After the **inputText** component, add a **commandButton** component. Bind this button's **action** property to a **showList** handler that executes a product search in the ProductManager CDI bean.

```
<h:commandButton value="Find" action="#{pm.showList}"/>
```

Note: In the JSF page, expressions are handled via FacesServlet, so they are prefixed with # rather than with **$**. You can still use the **$** prefix in a JSF page. However, it will execute your expression in a context of the Servlet life cycle, but not the JSF life cycle.

6. Add components to the List.xhtml file to display a list of products:
   a. Open the **List.xhtml** file.
   b. Change the List page **title** located inside the **h:head** section to **Product List**.

```
<title>Product List</title>
```

   c. Change the text inside the **header** element to **Products**.

```
<header class='header'>Products</header>
```

   d. Change the text inside the **footer** element to **Select product from list**.

```
<footer class='footer'>Select product from list</footer>
```

   Note: In a later practice, you will add code to this page to select a product and navigate to the edit page.
   e. Modify the navigation element in the list page. Replace the existing link to the Home page inside the nav element with a JSF commandLink component that navigates back to the Search.xhtml page, using **"search"** as the navigation rule name.

```
<nav class='nav'>
    <h:commandLink value="Product Search" action="search"/>
</nav>
```

f. Inside the **section** element after the **messages** component, add a **dataTable** component to produce a table to display products. The product collection to be displayed in this table should be acquired from the `ProductManager` bean. Each element in this collection should be given an alias of **"p"** — each item from the collection displayed in this table would be referenced using this alias.

```
<h:dataTable value="#{pm.products}" var="p">
</h:dataTable>
```

Inside the **dataTable** component, add four **column** components. Each column should define a **header facet** to display **ID, Name, Price,** and **Best Before** column headers. Also each column should contain an **outputText** element to display the relevant value.

```
<h:column>
     <f:facet name="header">ID</f:facet>
     <h:outputText value="#{p.id}"/>
</h:column>
<h:column>
     <f:facet name="header">Name</f:facet>
     <h:outputText value="#{p.name}"/>
</h:column>
<h:column>
     <f:facet name="header">Price</f:facet>
     <h:outputText value="#{p.price}"/>
</h:column>
<h:column>
     <f:facet name="header">Best Before</f:facet>
     <h:outputText value="#{p.bestBefore}"/>
</h:column>
```

7. Modify the `index.html` page to add navigation to the `Search.xhtml` page:
   a. Open the **index.html** file located under the **Web Pages** section in the **ProductWeb** project.
   b. Inside the **nav element,** find the HTML `anchor` element that points to the **ProductSearch.html** page. Add an HTML divider around it. Copy this element and paste it on a line before. Modify the `href` element to point to the **faces/Search.xhtml** page. Change the prompt to **JSF Product Search**.

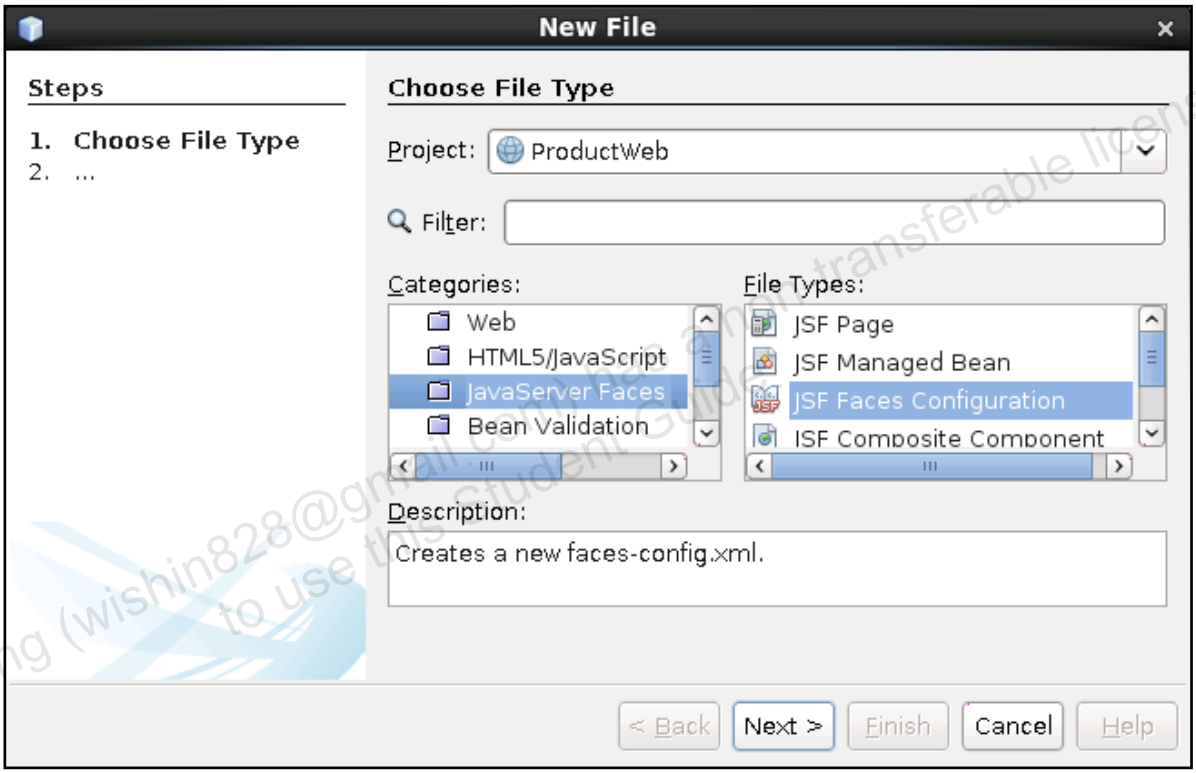
      This is what the resulting code should look like:

```
<div><a href='/pm/faces/Search.xhtml'>JSF Product Search</a></div>
<div><a href='/pm/ProductSearch.html'>Product Search</a></div>
```

Note: Do not remove existing code inside the `nav` element that contains script and button elements to open chat windows.

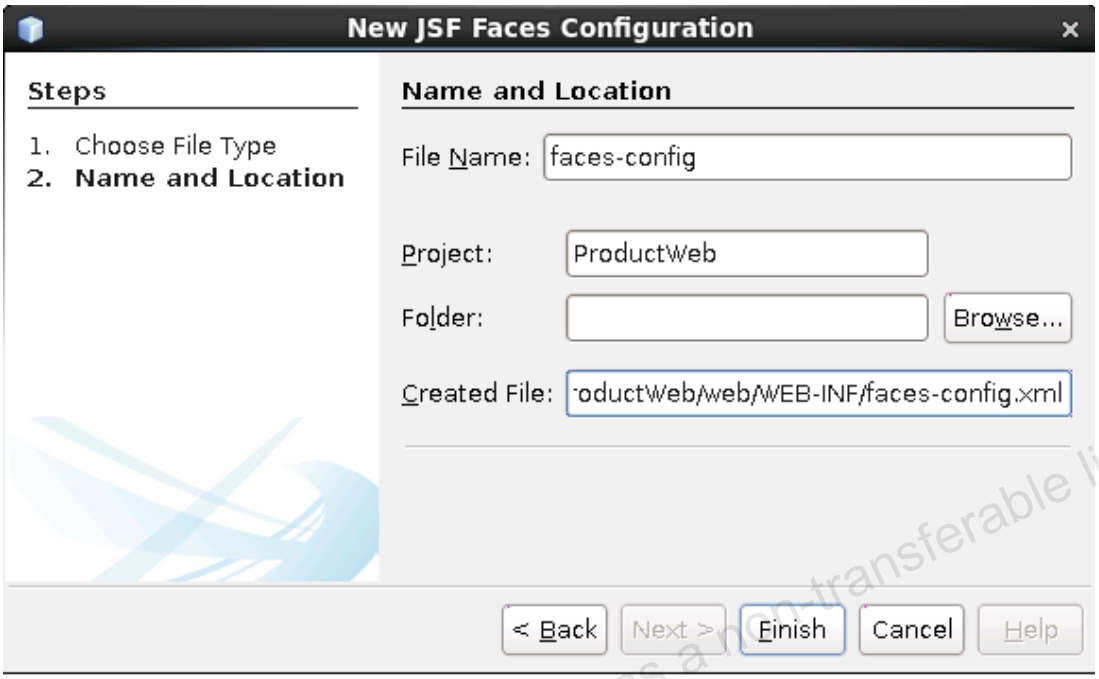
Note: When you invoke a JSF page, you are actually invoking a `FacesServlet`, and you pass the name of the JSF page to it as a path parameter. If you look in the `web.xml` file, you will see that FacesServlet has been mapped to the URL **"/faces"**.

8. Create the `faces-config.xml` file to define JSF navigation rules:
   a. Select the **ProductWeb** project.
   b. Select **File > New File**.
   c. Select **JavaServer Faces** from the list of Categories and **JSF Faces Configuration** from the list of File Types.



   d. Click **Next**.

e. In the New JSF Faces Configuration dialog box, enter `faces-config` as the file name.



f. Click **Finish**.

g. Open the **faces-config.xml** file located under **Web Pages > WEB-INF** node. Inside the `faces-config` element, add three navigation rules:

- Navigate from the Search to the List page using the outcome named **`list`**.
- Navigate from the List to the Edit page using the outcome named **`edit`**.
- Navigate from any page back to the Search page using the outcome named **`search`**.

```
<navigation-rule>
     <from-view-id>/Search.xhtml</from-view-id>
     <navigation-case>
          <from-outcome>list</from-outcome>
          <to-view-id>/List.xhtml</to-view-id>
     </navigation-case>
</navigation-rule>
<navigation-rule>
     <from-view-id>/List.xhtml</from-view-id>
     <navigation-case>
          <from-outcome>edit</from-outcome>
          <to-view-id>/Edit.xhtml</to-view-id>
     </navigation-case>
</navigation-rule>
<navigation-rule>
     <from-view-id>/*</from-view-id>
     <navigation-case>
          <from-outcome>search</from-outcome>
          <to-view-id>/Search.xhtml</to-view-id>
     </navigation-case>
</navigation-rule>
```

Note: It is possible to use page names instead of navigation rules to define navigation between JSF pages. However, navigation rules allow you to define more loosely coupled pages and CDI beans. You can refactor page names, or even navigate to completely different pages without affecting other pages or CDI beans.

h. Optionally click the **PageFlow** button to switch faces-config's source code view to the page-flow view mode. You may visually observe navigation rules you create. Switch back to the source code view by clicking the **Source** button.

Note: Unfortunately, unlike other Java IDEs, NetBeans does not support visual editing of JSF pages or flows. For example, in JDeveloper this entire practice would have been completely visual, with pages and flows design done with the drag-and-drop feature and visual page editing displaying a live preview of the output of your JSF pages as you edit them.

9. Test the JSF Search and List pages:
   a. Compile the **ProductWeb** project using **Clean and Build**.
   b. Right-click the **ProductWeb** project and select **Deploy.**
   c. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm`
   d. Click the **JSF Product Search** link.

      Note: The URL in the browser address bar points to the **`faces/Search.xhtml`** page.
   e. Enter `Co%` into the product name field.
   f. Click the **Find** button.
   g. Observe the product list.

      Note: The URL in the browser address bar still points to the **`faces/Search.xhtml`** page. This is because JSF forms always submit requests recursively back to the same page. However, after this page action handler completes its logic, it can instruct `FacesServlet` to perform "navigation" — and FacesServlet will let the page that is designated as the navigation target render the response.
   h. Click the **Product Search** link to navigate back to the Search page.

      Note: The URL in the browser address bar now points to the **`faces/List.xhtml`** page. This is because the List page form submitted the request recursively back to itself and then requested "navigation" to the Search page, so it the repose for the search page that you now see.

      Conclusion: In JSF navigation, you always see the URL of the page from which you are navigating rather than the page that rendered the response. Alternatively, you may `<redirect/>` markup inside navigation rule definition, or add an attribute `faces-redirect="true"` to implicit navigation rules. This will disable normal JSF lifecycle handling for this navigation action, but would allow you to match the rendered page and its URL.
   i. Now test how this application handles a case when a non-existent product name is supplied. For example, enter `acme` into the product name field.
   j. Click the **Find** button.

      Note: The response was rendered by the Search page; no navigation to the List occurred in this case. This is because the `showList` operation of the `ProductManager` class returned null instead of list when no products were found. Also notice the Warning FacesMessage.

k. Remove the value from product name field, leaving this field empty.

l. Click the **Find** button.

m. Notice an Error FacesMessage.

Note: When you are navigating from List back to the Search page, the product name field appears empty. This is because this field is bound to a property **name** of the **ProductManager** CDI bean, which is scoped to the request, so a new instance of ProductManager is created for very request that you make. Your next task will be to change the scope of the ProductManager bean and observe how it will affect the way values are retained by the JSF application.

10. Make the `ProductManager` CDI bean `SessionScoped`:

a. Select the **ProductWeb** project in NetBeans.

b. Expand **Source Packages > demos.model** node.

c. Open the **`ProductManager`** class.

d. Add an `implements Serializable` interface clause to the `ProductManager` class definition.

e. Add an import: `java.io.Serializable`

f. Replace the `RequestScoped` annotation with `SessionScoped`.

g. Add an import: `javax.enterprise.context.SessionScoped`

11. Test the JSF Search and List pages again:

a. Compile the **ProductWeb** project using **Clean and Build**.

b. Right-click the **ProductWeb** project and select **Deploy.**

c. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm`

d. Click the **JSF Product Search** link.

e. Enter `Co%` into the product name field.

f. Click the **Find** button.

g. Click the **Product Search** link to navigate back to the Search page.

Note: The Product Name field now "remembers" the previously used value, because it is now retained for the duration of the session by the ProductManager CDI bean.

12. Modify the `List.xhtml` page to allow it to select products from the list and navigate to the `Edit.xhtml` page:

a. Select the **ProductWeb** project in NetBeans.

b. Expand the **Web Pages** node.

c. Open the **`List.xhtml`** file.

d. Replace the **outputText** component that displays product **id** with a **commandLink** component. This command link should use product **id** as the value to be displayed on the link, and its action property should be bound to the **showEdit** operation of the ProductManager CDI bean.

```
<h:commandLink value="#{p.id}" action="#{pm.showEdit}">
</h:commandLink>
```

e. Inside this **commandLink** component, add a **f:param** component called **p_id** with product **id** set as its value.

```
<f:param name="p_id" value="#{p.id}"/>
```

13. Add components to the `Edit.xhtml` file to display the product update form:

a. Open the **Edit.xhtml** file.

b. Change the List page **title** located inside the **h:head** section to **Edit Product.**

```
<title>Edit Product</title>
```

c. Change the text inside the **header** element to **Edit Product.**

```
<header class='header'>Edit Product</header>
```

d. Change the text inside the **footer** element to **Change product and click Update.**

```
<footer class='footer'>Change Product and click Update</footer>
```

e. Modify the navigation element in the list page. Replace the existing link to the Home page inside the `nav` element with the JSF `commandLink` component that navigates back to the `Search.xhtml` page, using the **"search"** navigation rule name. This navigation component must not trigger form validation.

```
<nav class='nav'>
    <h:commandLink value="Product Search"
                   action="search" immediate="true"/>
</nav>
```

f. Inside the **section** element, after the **messages** component, add a `panelGird` component with two columns.

```
<h:panelGrid columns="2">
</h:panelGrid>
```

g. Inside the **panelGrid,** add an **outputLabel** component to display an **"ID:"** prompt for the input item that will represent product `id`.

```
<h:outputLabel for="id" value="ID:"/>
```

h. After the **outputLabel** component, add an **inputText** component. Bind the value of this input item to the **product id** property. Make this a **readonly** item.

```
<h:inputText id="id" value="#{pm.product.id}" readonly="true"/>
```

i. Add another **outputLabel** component to display a **"Name:"** prompt for the input item that will represent product name.

```
<h:outputLabel for="name" value="Name:"/>
```

j. After the **outputLabel** component, add an **inputText** component. Bind the value of this input item to the **product name** property. Make this a **required** item.

```
<h:inputText id="name" value="#{pm.product.name}" required="true"
      requiredMessage="Please specify product name"/>
```

k. Add another **outputLabel** component to display a **"Price:"** prompt for the input item that will represent product price.

```
<h:outputLabel for="price" value="Price:"/>
```

l. After the **outputLabel** component, add an **inputText** component. Bind the value of this input item to the **product price** property. Make this a **required** item.

```
<h:inputText id="price" value="#{pm.product.price}"
required="true" requiredMessage="Please specify product price"/>
```

m. Add another **outputLabel** component to display a **"Best Before:"** prompt for the input item that will represent a product's best-before-date.

```
<h:outputLabel for="date" value="Best Before:"/>
```

n. After the **outputLabel** component, add an **inputText** component. Bind the value of this input item to the **product bestBefore** property. Inside this **inputText** component, add an **f:converter** component with converter ID **LocalDateConverter**. This component will manage conversions between String and LocalDate types.

```
<h:inputText id="date" value="#{pm.product.bestBefore}">
    <f:converter converterId="LocalDateConverter"/>
</h:inputText>
```

o. Add a **commandButton** component to represent the **Update** button. Bind this component's **actionListener** property to the **handleUpdate** operation of the `ProductManager` bean.

```
<h:commandButton value="Update"
actionListener="#{pm.handleUpdate}"/>
```

14. Test the JSF Search and List pages:
    a. Compile the **ProductWeb** project using **Clean and Build**.
    b. Right-click the **ProductWeb** project and select **Deploy**.
    c. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm`
    d. Click the **JSF Product Search** link.
    e. Enter `Co%` into the product name field.
    f. Click the **Find** button.
    g. Observe the product list. Now product ID items in this list are clickable.

h. Click any product ID.

i. You will get to the Edit page where you can change any of the product deals and click the Update button. Test a correct update first: set price to be 3.99 and click **Update**. You should see an information message confirming that the product was successfully updated.

j. Test a missing required values case: Remove the value from the price item and click **Update**. You should see an error message informing you that the value of price is required.

k. Test an incorrect value update: Change price to be less than 1 and Name to be shorter than two characters. Click **Update**. See the error messages produced by the Bean Validation constraint placed on a Product entity.

l. Change name and price back to the correct values. **Click** Update. Observe the information message.

m. Optionally, (if you have time) you may test an Optimistic Lock exception handling, by updating the same product that you have selected in this form, using SQL Console or any other client, such as the ProductApp-client.

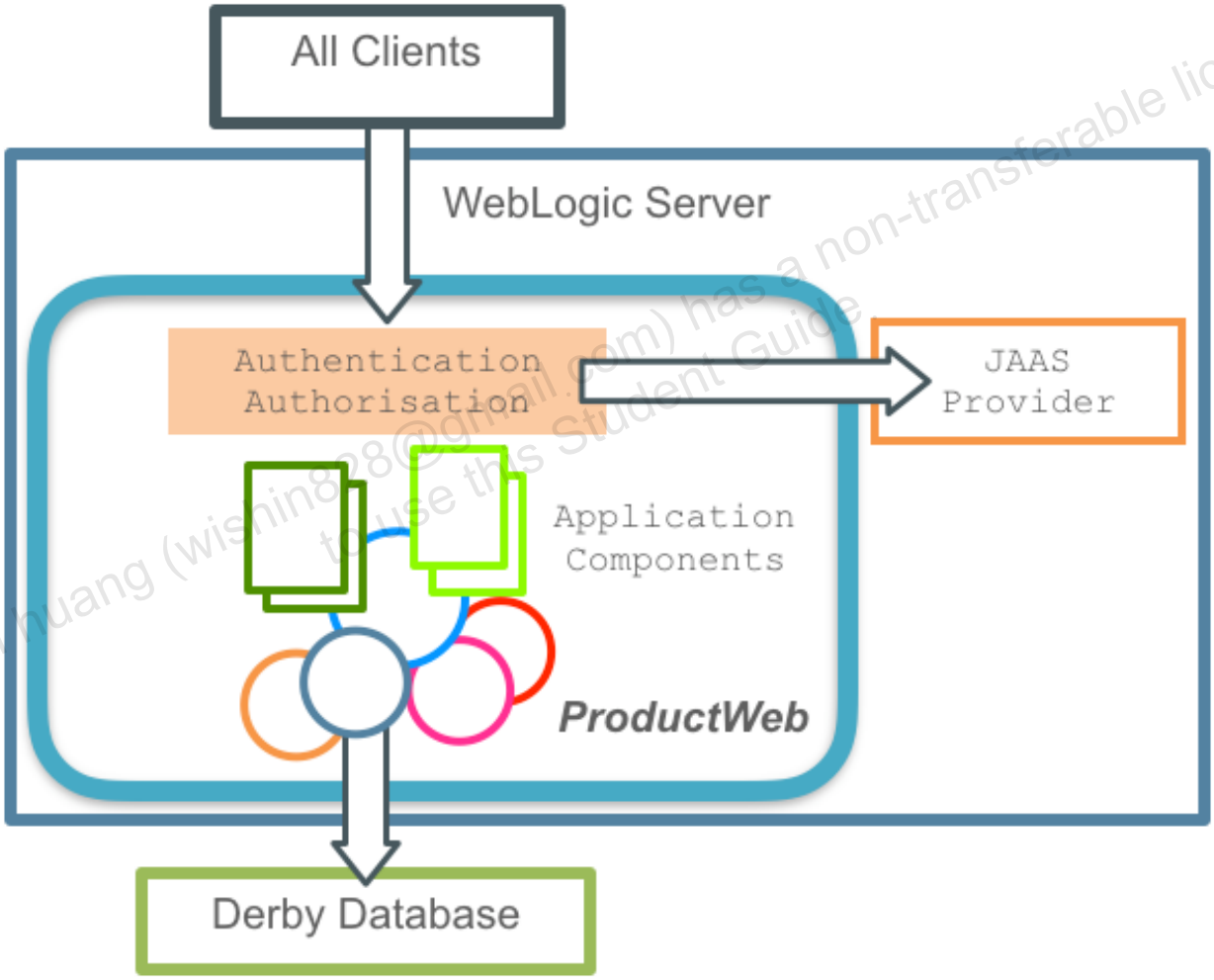
Note: During practice 11 you were instructed to pass parameters between JSF pages. For example, in practice 11-1 step 3.b you were told to write CDI bean that expects product id to be passed as a simple URL parameter. The main benefit of this approach is that it creates a simple URL reference to a target page, that can be easily bookmarked.
More "JSF-Centric" way would be to use JSF View parameter instead. However, this approach would require to also provide a converter, to convert product id value into product object that target page is going to use.
In the practice 11-2 steps 12.d and 12.e you were instructed to describe parameter `p_id` defined using `<f:param>` element.
Alternately, you could have passed parameter using the following code:
`<h:commandLink value="#{p_id}" action="#{pm.showEdit(p.id)}">.`
This means that the supporting CDI bean showEdit method would be defined with parameter that would represent the product id, instead of fetching this value using ExternalContext object.

Practices for Lesson 12: Securing Java EE Applications

Practices for Lesson 12: Overview

Overview

In these practices, you add form-based authentication to your Java EE application, define declarative security with application roles and security constraints, map application roles to security groups that you define using WebLogic's built-in security server as your JAAS provider, and add programmatic security handling to your application.



Practice 12-1: Adding Authentication and Authorization Logic

Overview

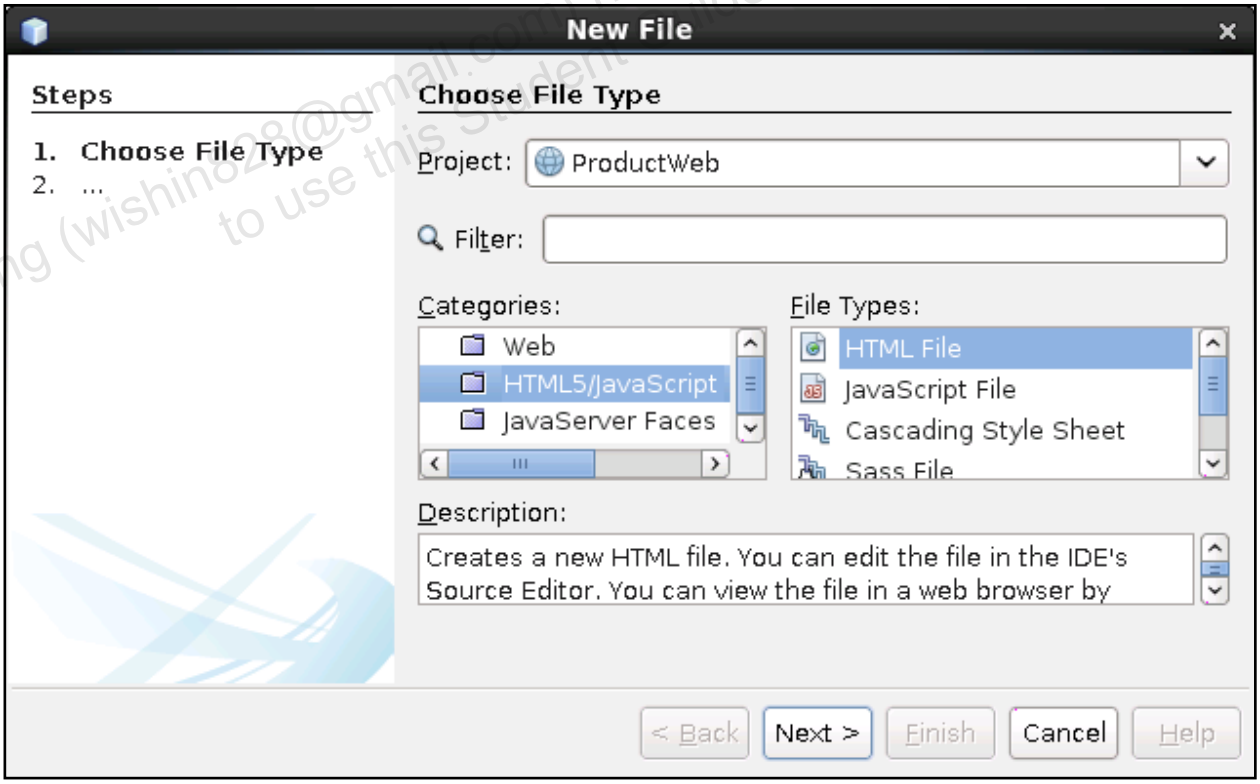
In this practice, you create `Login` and `LoginError` HTML pages to handle form authentication. You will register these pages with the `web.xml` file in a later practice. You modify your application welcome page to make it security-aware, provide a link to the `Login` page, handle logout, and dynamically change what it displays depending on whether the user is authenticated or not.

Assumptions

You have successfully completed all previous practices.

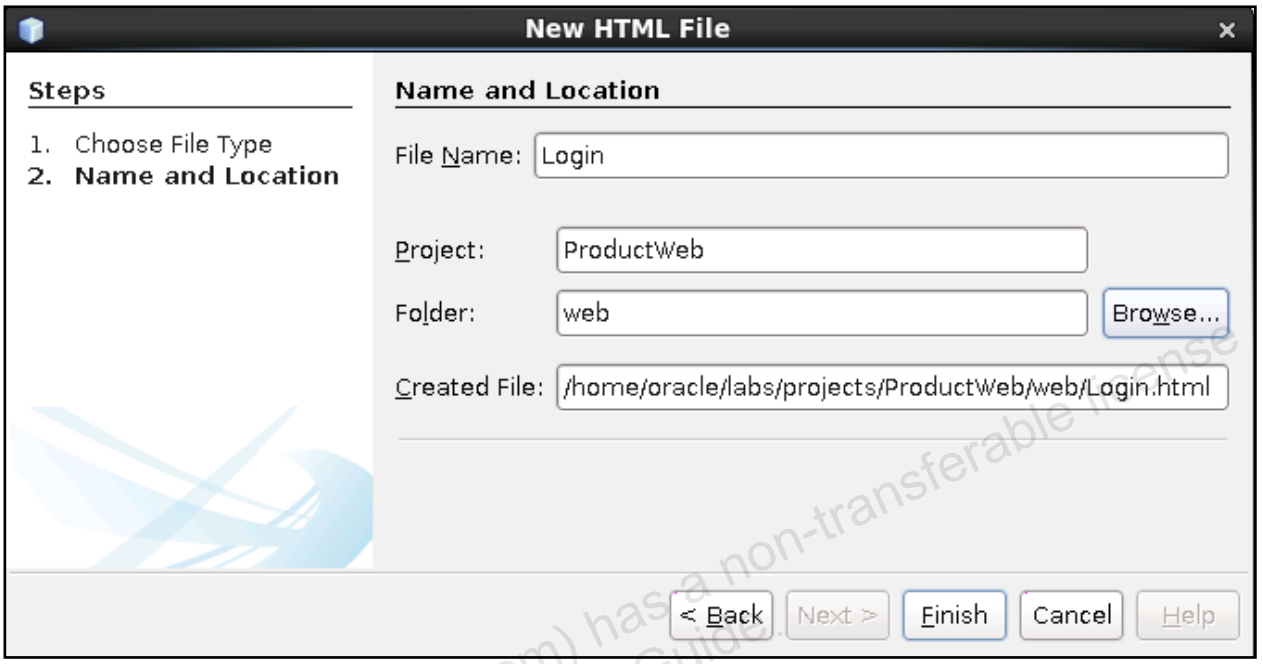
Tasks

1. Create the `Login` HTML page:
   a. Select the **ProductWeb** project.
   b. Select **File > New File**.
   c. Select **HTML5/JavaScript** from the list of Categories and **HTML File** from the list of File Types.



   d. Click **Next**.

e. In the New HTML File dialog box, set the following properties:
   - File Name: `Login`
   - Folder: `web`



f. Click **Finish**.

2. Design the `Login` page:
   a. Modify the `Login.html` code to define the page structure. Replace the entire contents of the `Login.html` file with the contents of the `template.html` file located in the `/home/oracle/labs/resources` folder.
   b. Change the `Login` page's title, header, navigation, and footer elements.
      - Replace the text PAGE TITLE with: `Login`
      - Replace the text PAGE HEADER with: `Login`
      - Replace the text PAGE FOOTER with: `Login with your username and password.`
   c. Replace the text PAGE NAVIGATION with a link back to the home page (`index.html`):
      ```
      <a href="/pm">Home</a>
      ```
   d. Replace the content area of the `Login` page with an HTML form. This form should submit itself to a servlet mapped to a `j_security_check` URL. You should use HTTP method POST to submit this form.
      - Replace the text PAGE CONTENT with:
        ```
        <form action="j_security_check" method="POST">
        </form>
        ```

e. Inside this form element, add an HTML divider containing a text input item with a label to represent a field for a username.

```
<div class="field">
     <label for="user">Username:</label>
    <input type="text" id="user" name="j_username" required>
</div>
```

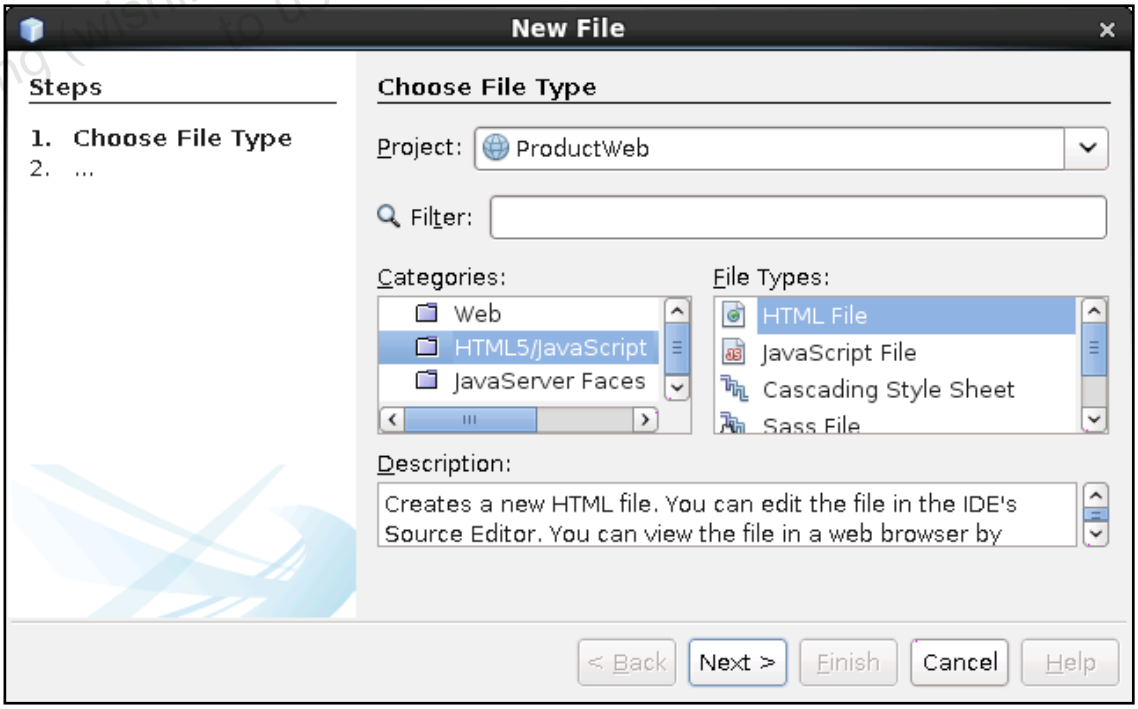
f. Add an HTML divider containing a password input item with a label to represent a field for a password.

```
<div class="field">
     <label for="pass">Password:</label>
    <input type="password" id="pass" name="j_password"
required>
</div>
```

g. Add another divider containing a submit button for this form.

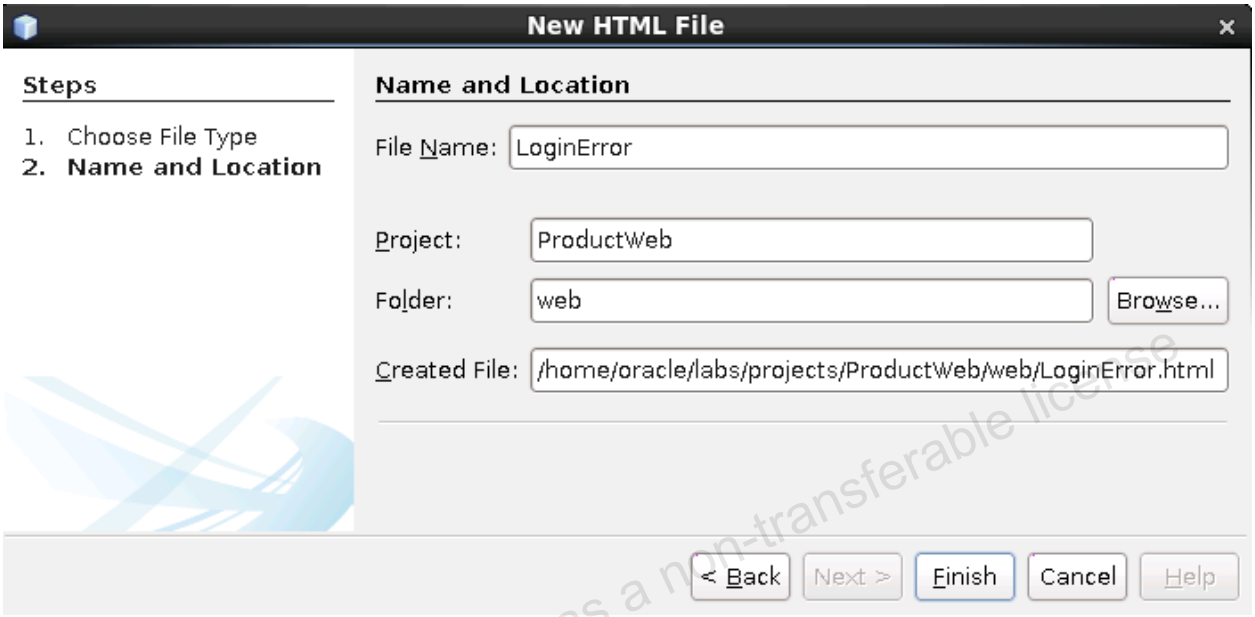
```
<div class="field">
  <input type="submit" value="Login">
</div>
```

3. Create the `LoginError` HTML page:
   a. Select the **ProductWeb** project.
   b. Select **File > New File**.
   c. Select **HTML5/JavaScript** from the list of Categories and **HTML File** from the list of File Types.



   d. Click **Next**.

e. In the New HTML File dialog box, set the following properties:
   - File Name: `LoginError`
   - Folder: `web`



f. Click **Finish**.

4. Design the `LoginError` page:
   a. Modify the `Login.html` code to define the page structure. Replace the entire contents of the `LoginError.html` file with text from the `template.html` file located in the `/home/oracle/labs/resources` folder.
   b. Change the `LoginError` page title, header, navigation, and footer elements.
      - Replace the text PAGE TITLE with: `Login Error`
      - Replace the text PAGE HEADER with: `Login Error`
      - Replace the text PAGE FOOTER with: `Login Error.`
   c. Replace the text PAGE NAVIGATION with a link back to the home page (`index.html`) and a link back to the login page (`Login.html`):

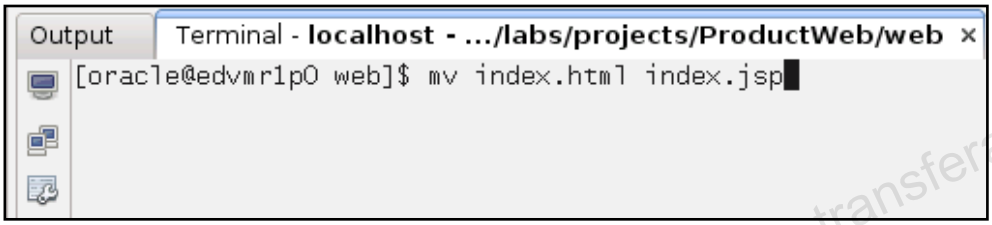
```
<div><a href="/pm">Home</a></div>
<div><a href="/pm/Login.html">Try to login again</a></div>
```

   d. Replace the content area of the `LoginError` page with an HTML divider with a visual class error containing a message indicating that the login attempt has failed.
      - Replace the text PAGE CONTENT with:

```
<div class="error">Your login attempt has failed.</div>
```

5. Modify the `index.html` page to dynamically handle authenticated and nonauthenticated requests. This requires you to change the `index.html` page (which is just a static HTML file at the moment) to an `index.jsp` page. This allows you to add the required dynamic processing to the page.
   a. Select the **`index.html`** file in the NetBeans project.
   b. Select **Tools > Open in Terminal**. A **Terminal** panel appears in the booth part of the NetBeans window.
   c. Inside the terminal panel, type the following command:

      ```
      mv index.html index.jsp
      ```

   d. Press **Enter**.
   e. Close the terminal panel by typing `exit` and pressing **Enter**.
   f. Press **Enter**.

6. Modify the `Index.jsp` page to perform the following actions:
   - Hide and show navigation links depending on whenever the incoming request is placed by the authenticated user or not.
   - Add logout handling.
   - Display the username of the logged-in user.

   a. Open the **`index.jsp`** file in the ProductWeb project.
   b. At the very beginning of the page, before the `<!DOCTYPE html>` declaration, add JSP **`page`** and **`taglib`** declarations.

      Hint: You can copy these two lines of code from any other JSP page in your project (for example, `ProductList.jsp`).

      ```
      <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
      <%@page contentType="text/html" pageEncoding="UTF-8"%>
      ```

c. After these two lines, but before the `<!DOCTYPE html>` declaration, add a JSTL **`if`** condition to check if a request contains the **`logout`** parameter. You are going to introduce this parameter in a later step of this practice. If this is the case programmatically log out this user.

```
<c:if test="${param.logout != null}">
    ${pageContext.request.logout()}
    ${pageContext.session.invalidate()}
</c:if>
```

Note: SessionScope is not automatically destroyed when you log out your user. If you want that to be the case, you should perform it as a separate step.

d. Enable the display of all navigation links only when this page is accessed by a properly authenticated user. To achieve this, surround all content within the **`nav`** element with JSTL **`choose / when / otherwise`** elements. Add a test to the `when` element that checks if the `userPrincipal` object exists. In addition, if the `userPrincipal` object is not null, set a new variable called **`user`** and assign the username to it. The **`otherwise`** element should contain a link pointing to the **`Login`** page. Finally, at the end of all the existing content inside the **`when`** element, add a link that triggers the **`logout`** action.

```
<nav class='nav'>
<c:choose>
<c:when test="${pageContext.request.userPrincipal != null}">
<c:set var="user">${pageContext.request.userPrincipal.name}</c:set>
  ALL EXISTING CONTENT OF THE NAV ELEMENT SHOULD REMAIN HERE
  <div><a href="/pm/index.jsp?logout">Logout</a></div>
</c:when>
<c:otherwise>
  <div><a href="/pm/Login.html">Login</a></div>
</c:otherwise>
</c:choose>
</nav>
```

e. Change the text inside the **`div`** element located within the **`section`** element, to display the username that you have assigned to a **`user`** variable.

```
<section class='content'>
    <div>Welcome ${user}</div>
</section>
```

Note: The section of the index.jsp file that contains links pointing to Search.xhml and ProductSerach.html are now only accessible by authenticated users. However, they should really be restricted to members of the employee and customer roles only. You can test this discrepancy by login as a user who is not a member of either of these roles, for example user weblogic. You would be able to see these links, but if you try to actually access corresponding pages you will get an unauthorized error. Optionally, if you have

time, you may write an extra condition around these links to display them only to members of appropriate roles.

Practice 12-2: Configuring Java EE Web Module Security

Overview

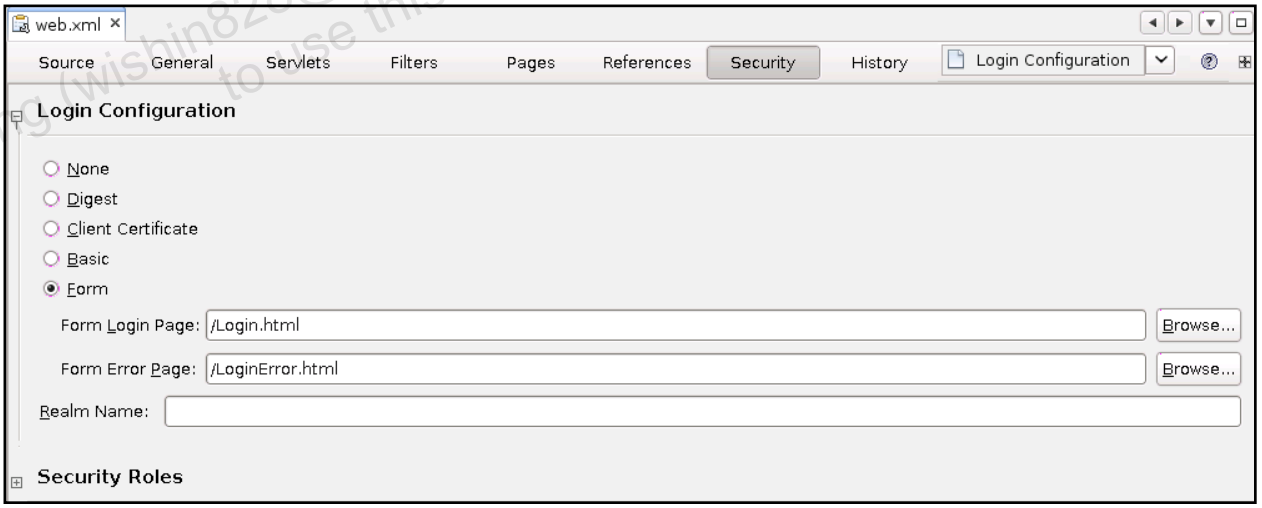
In this practice, you create a security configuration for the ProductWeb module, define your login module configuration, add application security roles, and create security constrains to protect application resources.

Assumptions

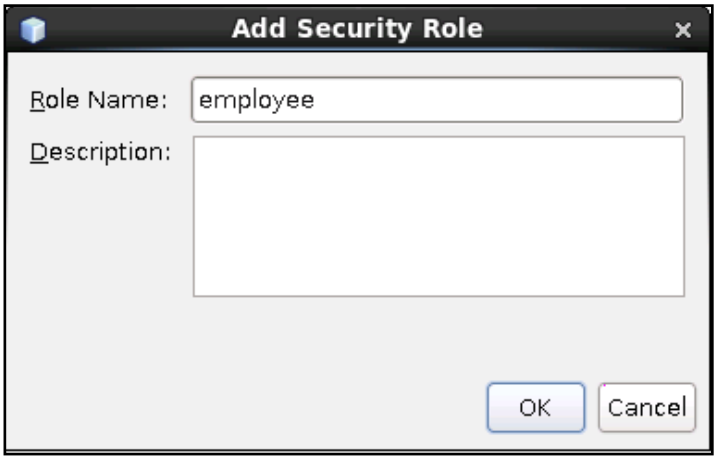
You have successfully completed all previous practices.

Tasks

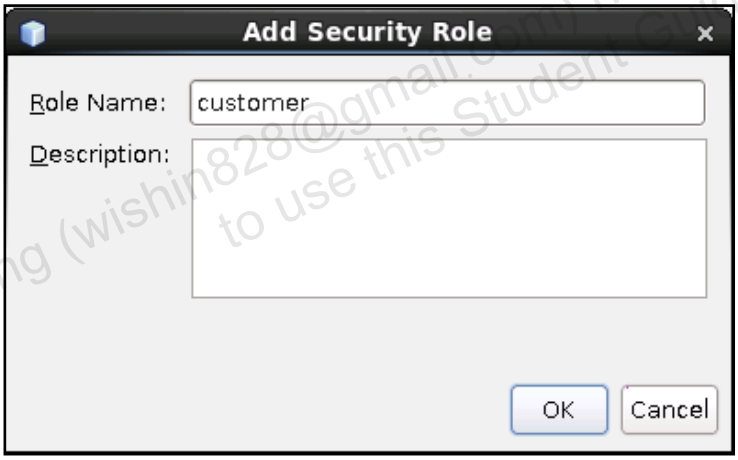
1. Create a Java EE Login Module configuration by registering the `Login` page with the `web.xml` file as a form-based authentication page:
   a. Open the **`web.xml`** file located under **Web Pages > WEB-INF** folder.
   b. Click the **Security** button on top of the `web.xml` source page view.
   c. Expand the **Login Configuration** section.
   d. Select the **Form** option button.
   e. Set the **Form Login Page** property as `/Login.html` You could select it using the Browse button.
   f. Set the **Form Error Page** property as `/LoginError.html` You could select it using the Browse button.



2. Create Application Security Roles:
   a. Continue using the same **Security** screen of the **`web.xml`** file.
   b. Click the **Add** button in the **Security Roles** section.
   c. In the Add Security Role dialog box, enter `employee` for the **Role Name** property.

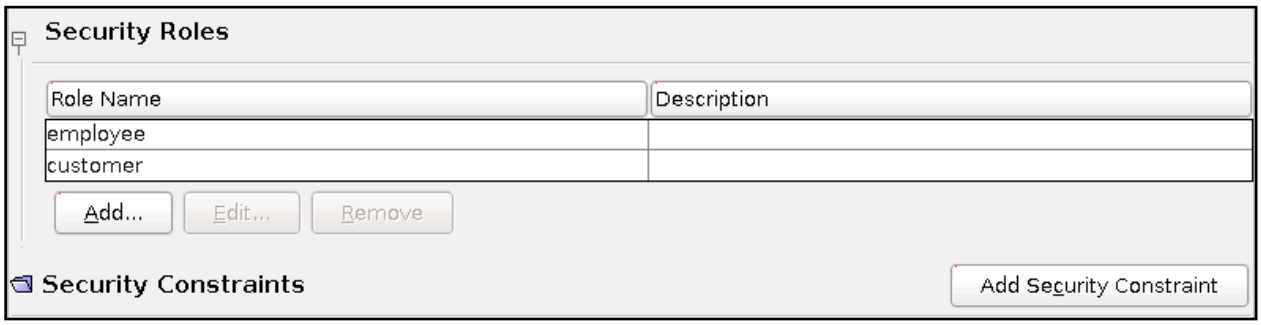


   d. Click **OK**.
   e. Click the **Add** button in the **Security Roles** section again.
   f. In the Add Security Role dialog box, enter `customer` for the **Role Name** property.

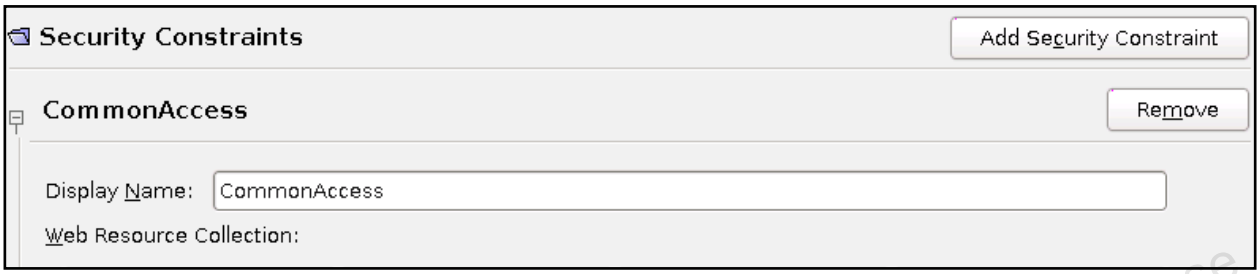


   g. Click **OK**.

This is what the result should look like:



3. Create a Security Constraint that describes access to resources restricted to anyone who is a member of an employee or customer role.
   a. Continue using the same **Security** screen of the **`web.xml`** file.
   b. Click the **Add Security Constraint** button.
   c. Set the **Display Property** to `CommonAccess`.



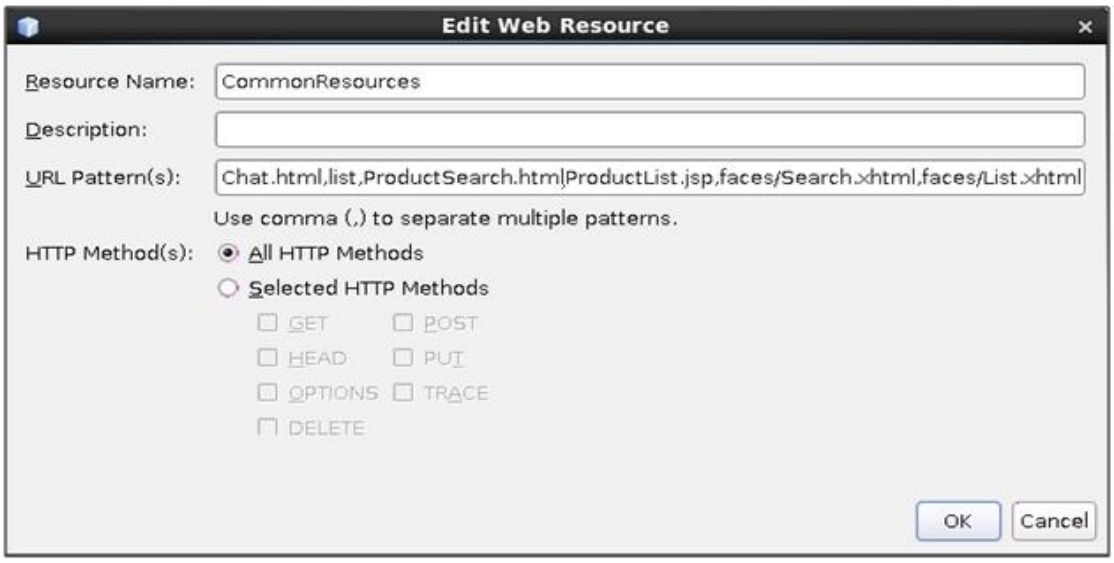
   Note: The purpose of this security constant is to represent any pages that require authentication, regardless of whether the user is an employee or customer.

   d. Click the **Add** button under the **Web Resource Collection** section.
   e. In the Add Web Resource dialog box, set the following properties:
      - Resource Name: `CommonResources`
      - URL Pattern(s):

```
Chat.html
list
ProductSearch.html
ProductList.jsp
faces/Search.xhtml
faces/List.xhtml
```

   (This is a comma-separated list of URLs written as one line.)

   f. Make sure you restrict **All HTTP Methods**.

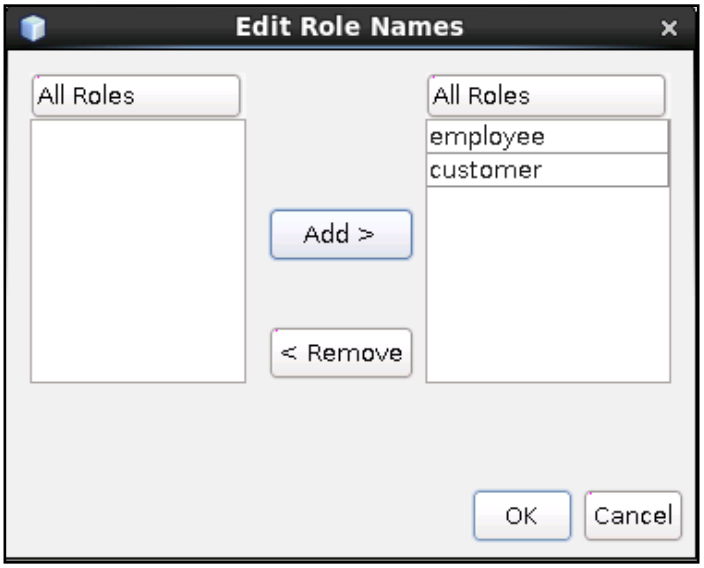


   g. Click **OK**.

h. Select the **Enable Authentication Constraint** check box.

i. Click the **Edit** button on the right side of the **Role Name(s)** field.

j. In the **Edit Role Names** dialog box, select **All Roles** and click the **Add >** button.



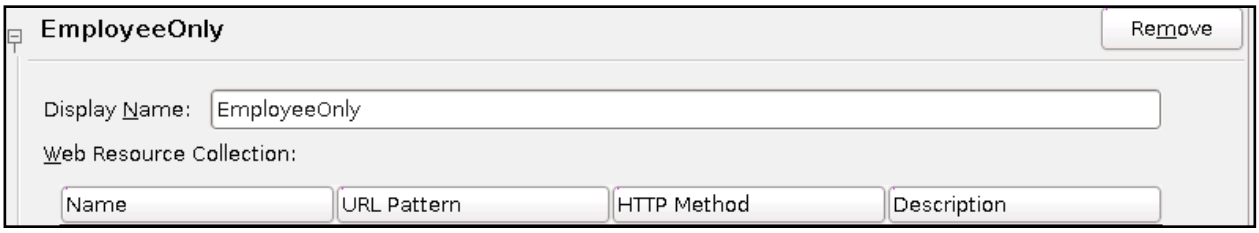
k. Click **OK**.

Note: Your first security constraint, called `CommonAccess`, is now protecting `Chat.html` and also all pages related to product search and list functionalities. This constraint will allow any member of employee or customer roles to access these pages. It also means that your users must be authenticated before they try to access these resources.

4. Create Security Constraints that describe access to resources restricted to anyone who is a member of an employee role only.

a. Click the **Add Security Constraint** button again.

b. Set the **Display Property** to `EmployeeOnly`.

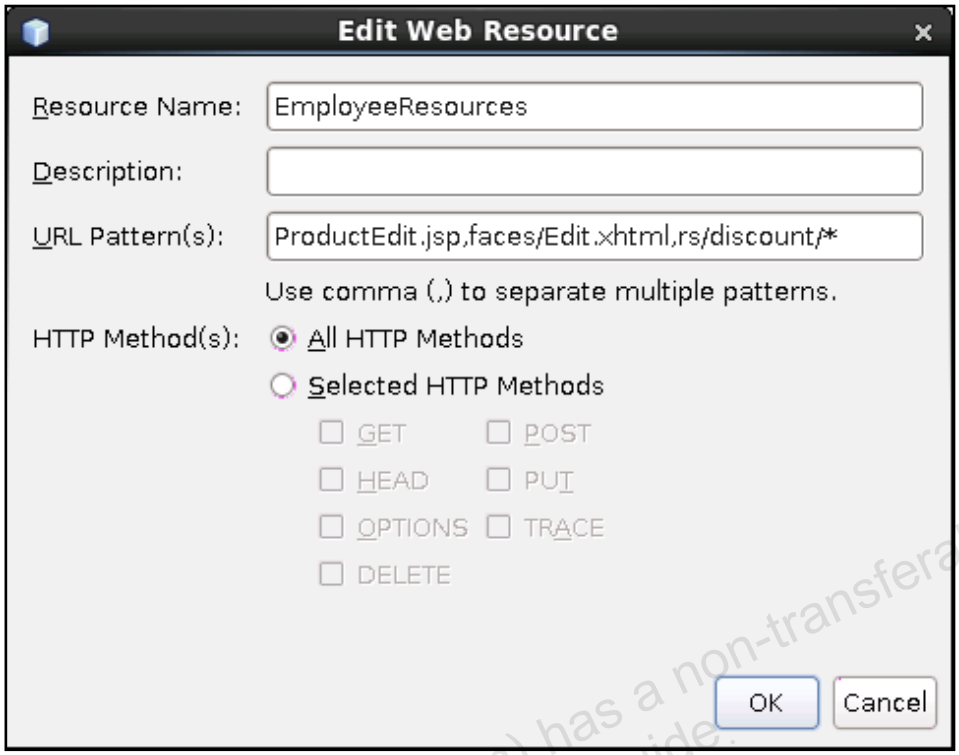


c. Click the **Add** button under the **Web Resource Collection** section.

d. In the Add Web Resource dialog box, set the following properties:

- Resource Name: `EmployeeResources`
- URL Pattern(s): `ProductEdit.jsp,faces/Edit.xhtml,rs/discount/*`
  (It is a comma-separated list of URLs.)

- Make sure you restrict **All HTTP Methods**.

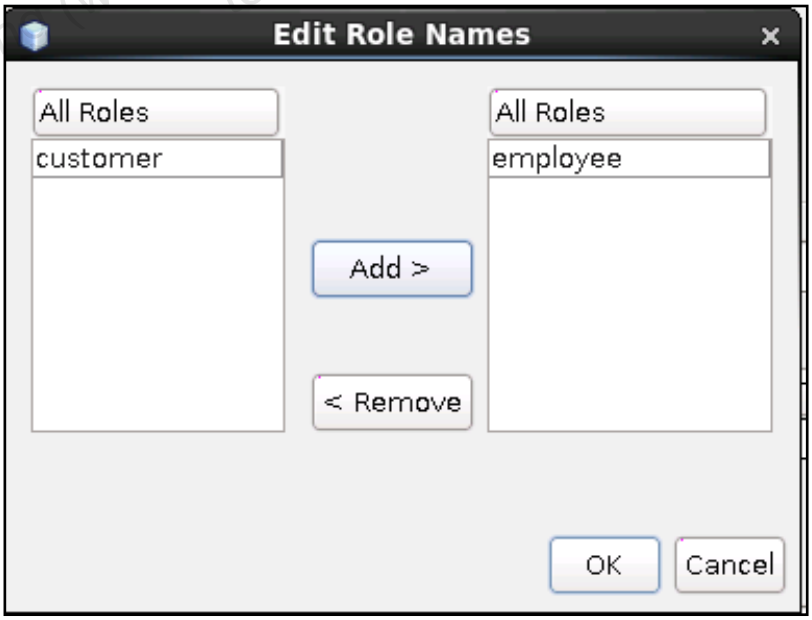


e. Click **OK**.

f. Select the **Enable Authentication Constraint** check box.

g. Click the **Edit** button on the right side of the **Role Name(s)** field.

h. In the Edit Role Names dialog box, select the **employee** role only and click the **Add >** button.



i. Click **OK**.

Note: Your second security constraint called `EmployeeOnly` is now protecting your product update pages and the Discount REST service. It will allow only members of the employee role to access these URLs.

Note: Other pages (`index.jsp` and `Login.html`) are not protected with any security contains. Therefore, anyone can access these pages without providing any authentication at all.

Practice 12-3: Configuring WebLogic Security and Mapping Security Roles

Overview

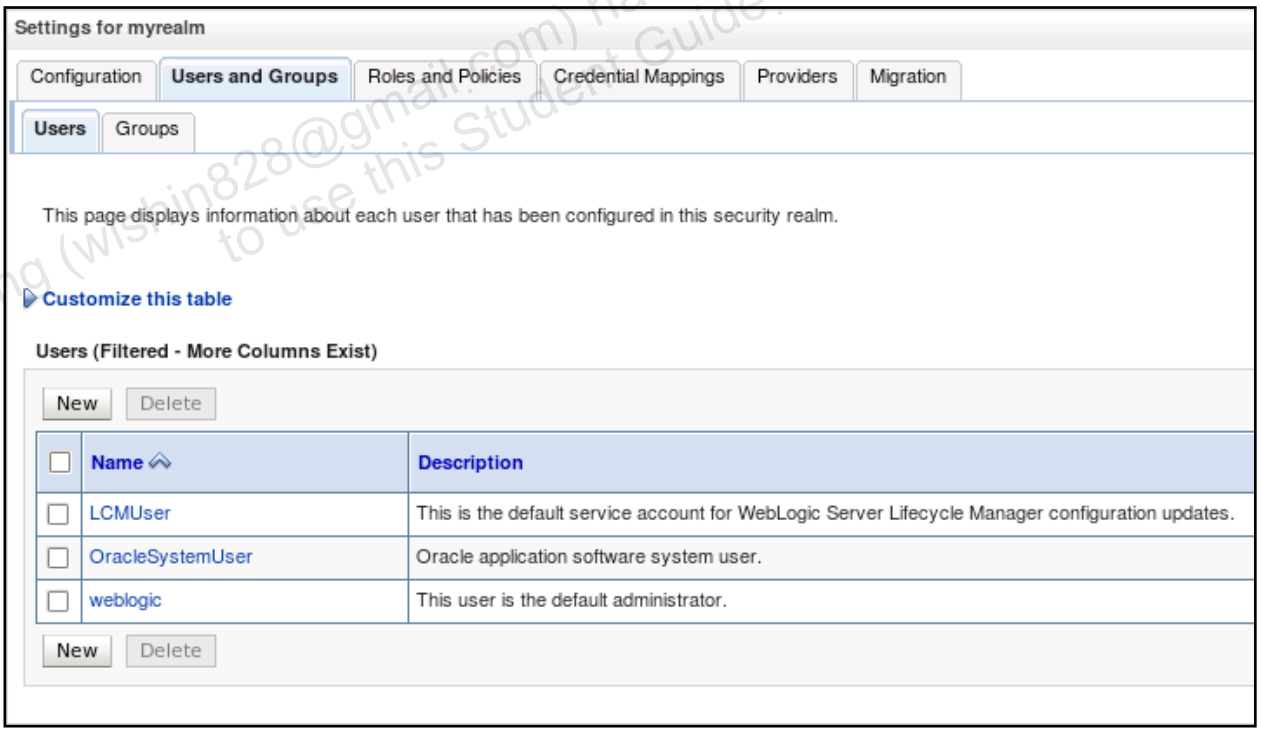
In this practice, you configure users and groups within the WebLogic built-in security provider. You then map these groups to your application roles.

Assumptions

You have successfully completed all previous practices.

Tasks

1. Create two users, `jbloggs` and `jdoe`, both with `welcome1` passwords:
   a. Open WebLogic Console in a web browser: `http://localhost:7001/console`
   b. Log in with username `weblogic` and password `welcome1`.
   c. In the **Domain Structure** panel, click the **Security Realms** link.
   d. Click the **myrealm** link.
   e. Click the **Users and Groups** tab. This will open the **Users** subtab.



   f. Click the **New** button.

g. In the Create a New User dialog box, set the following properties:

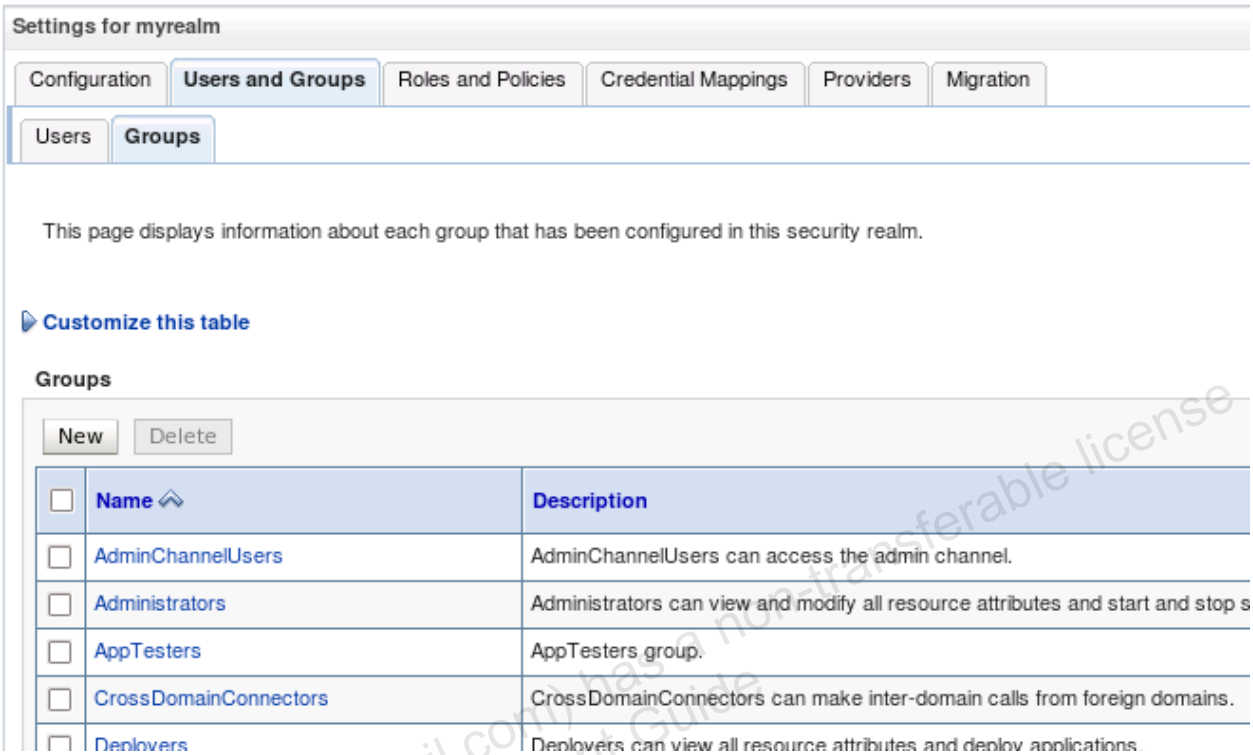
- Name: `jbloggs`
- Password: `welcome1`
- Confirm Password: `welcome1`

h. Click **OK**.

i. Create another user (repeat the process described above).

- Name: `jdoe`
- Password: `welcome1`

2. Create two groups: `client` and `clerk`
   a. Click the **Groups** subtab, located inside the **Users and Groups** tab.



   b. Click the **New** button.

c. In the **Create a New Group** dialog box, set the **Name** property to: `client`

d. Click **OK**.

e. Create another group (repeat the process described above).

- Name: `clerk`

f. Click **OK**.

3. Assign users to groups.

a. Click the **Users** subtab.

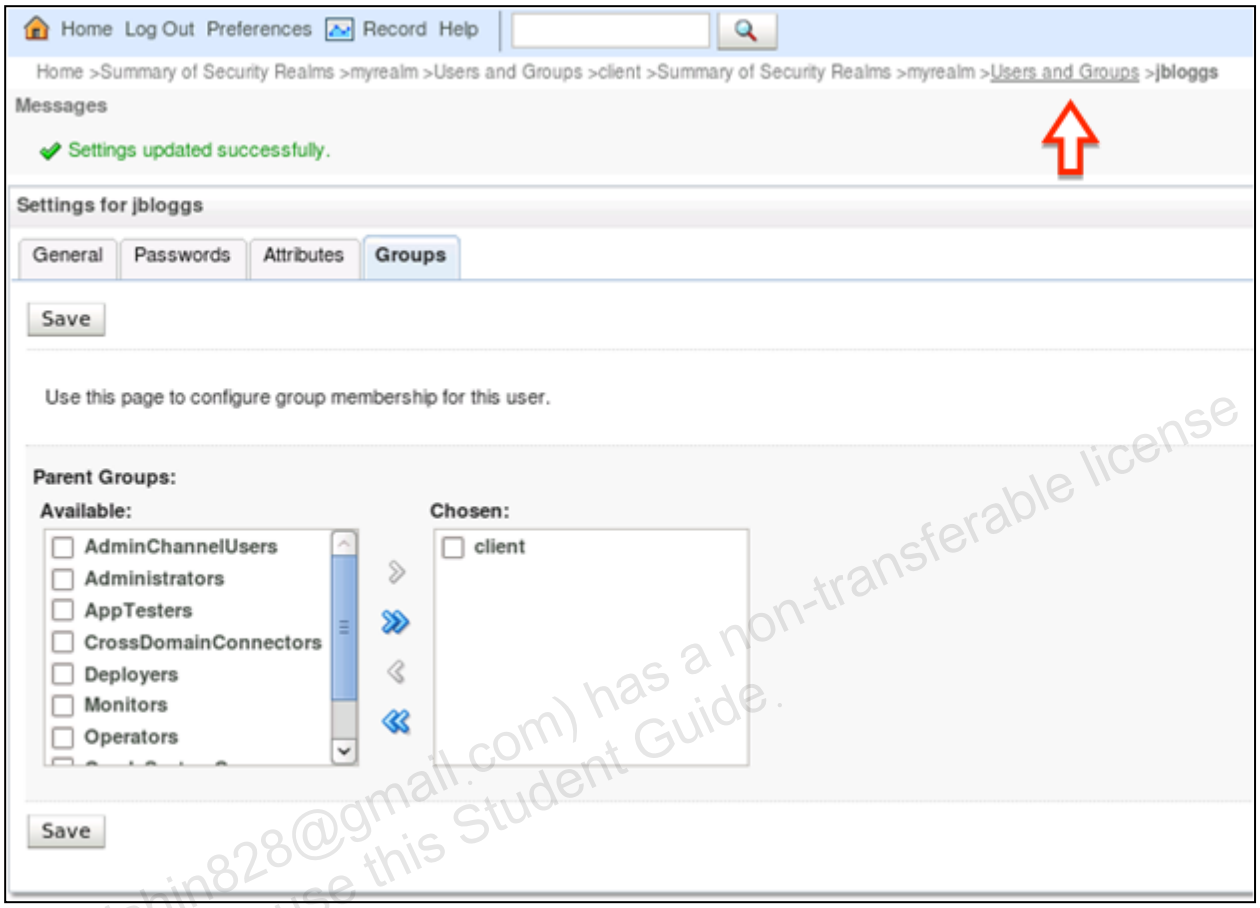
b. Click **jbloggs**.

c. Click the **Groups** tab in the **Settings for jbloggs** dialog box.

d. Select the group **client** from the list of Available groups and add it to the list of Chosen groups.

e. Click the **Save** button.

f. Return back to the list of users by clicking the **Users and Group** link in the bread-crumbs on top of the page.



g. Make user **`jdoe`** a member of the **clerk** group. (Repeat the process described above.)

4. Map the groups **client** and **clerk** to the roles **customer** and **employee**:
   a. Select the **ProductWeb** project in NetBeans.
   b. Open the **`weblogic.xml`** file located under **Web Pages > WEB-INF** folder.
   c. Inside the **`weblogc-web-app`** element, add two **security role assignment** elements to map the role `customer` to the principal name `client` and the role employee to the principal name `clerk`.

```
<security-role-assignment>
  <role-name>customer</role-name>
  <principal-name>client</principal-name>
</security-role-assignment>
<security-role-assignment>
  <role-name>employee</role-name>
  <principal-name>clerk</principal-name>
</security-role-assignment>
```

Practice 12-4: Adding Programmatic Security and Testing the Application

Overview

In this practice, you modify elements in the `ProductList.jsp` and `List.xhtml` pages to disable navigation to corresponding Edit pages if the user is not a member of the employee role.

You also modify WebSocket ChatServer to use authenticated usernames in the chat.

Assumptions

You have successfully completed all previous practices.

Tasks

1. Modify `ProductList.jsp` to disable navigation to the `ProductEdit.jsp` page if your user is not a member of the `employee` role:
   a. Open the **`ProductList.jsp`** code.
   b. Inside the `forEach` iterator that displays the list of products, locate the line of code that produces a reference to `ProductEdit.jsp`.
   c. Surround this line with a `choose` / `when` / `otherwise` statement that tests if the current request is associated with an authenticated user who is a member of the employee role. If this is the case, display the link to the `ProductEdit` page as usual; otherwise, simply display product ID with any link around it.

```
<c:choose>
<c:when test="${pageContext.request.isUserInRole('employee')}">
    <a href="ProductEdit.jsp?p_id=${p.id}">${p.id}</a>
</c:when>
<c:otherwise>
    ${p.id}
</c:otherwise>
</c:choose>
```

   Note: You could also add a check if `userPrincipal` is not null. However, your declarative security constraint is already configured to prevent access to this page by nonauthenticated invokers.

2. Modify the **`List.xhtml`** JSF page to disable navigation to the `Edit.xhtml` page if your user is not a member of the employee role.
   a. Open the **`List.xhtml`** code.
   b. Inside **`dataTable`** component, locate the **`commandLink`** component that displays the link to navigate to the Edit page.

c. Add an attribute to this `commandLink` that disables it when the user is not a member of the `employee` role.

```
<h:commandLink value="#{p.id}"
               action="#{pm.showEdit}"
               disabled="#{!request.isUserInRole('employee')}">
    <f:param name="p_id" value="#{p.id}"/>
</h:commandLink>
```

3. Modify WebSocket ChatServer to use authenticated usernames in the chat.
   a. Open **ChatServer** class located in **demos.web.socket** package.
   b. Inside the **onOpen** operation, add code to extract user **name** from the **UserPrincipal** object. You can obtain **UserPrincipal** object from the **session** object you received as an argument. Change the welcome message and broadcast message to include this user name.

```
@OnOpen
public void onOpen(Session session) {
    sessions.add(session);
    String user = session.getUserPrincipal().getName();
    session.getAsyncRemote().sendText("Welcome "+user+"!");
    broadcastMessage(user+" has joined the chat");
}
```

   c. Inside the **onClose** operation, add code to extract user **name** from the **UserPrincipal** object. You can obtain the **UserPrincipal** object from the **session** object that you received as an argument. Change the broadcast message to include this username.

```
@OnClose
public void onClose(Session session, CloseReason reason) {
    sessions.remove(session);
    String user = session.getUserPrincipal().getName();
    broadcastMessage(user+" has left the chat");
}
```

d. Modify the onMessage operation. Add a **Session** parameter, extract the user **name** from the **UserPrincipal** object. You can obtain the **UserPrincipal** object from the **session** object you received as an argument. Change the broadcast message to include this username.

```
@OnMessage
public void onMessage(Session session, String message) {
    if(message == null || message.length() == 0){
        throw new RuntimeException("No actual message received");
    }
    String user = session.getUserPrincipal().getName();
    broadcastMessage(user+": "+message);
}
```

4. Test the ProductWeb application:
   a. Compile the **ProductWeb** project using **Clean and Build**.
   b. Right-click the **ProductWeb** project and select **Deploy.**
   c. After this project has been deployed, open the browser and navigate to the following URL: `http://localhost:7001/pm`
   d. Notice that the welcome page is now an `index.jsp` and it dynamically altered its appearance to display a Login link.
   e. Try an unauthorized access to any protected pages of this application. Simply type into the browser address bar a protected URL. For example:

      `http://localhost:7001/pm/ProductList.jsp`

      The server will prevent you from seeing this page and will display a login page instead.

5. Test application access as a member of the **`customer`** role:
   a. Log in as user `jbloggs` with password `welcome1`.

      Note: Now you page is displayed. However, because you did not supply any product search conditions, it did not display any products.
   b. Click a **ProductSearch** link.
   c. Set the Product name to be `%` and click **Find**.

      Note: Now you should see the list of products. However, because you have logged in as a user who is not a member of the `employee` role, you do not see the clickable navigation link to the `ProductEdit.jsp` page.
   d. Try an unauthorized access to `ProductEdit.jsp`. Although the link to that page is not displayed, you can still attempt to access it by typing the value that would have been on this link into the browser address bar:

      `http://localhost:7001/pm/ProductEdit.jsp?p_id=2`

      Note: You are now getting an HTTP 403 -- Forbidden response. The reason you are getting this error is, of course, that you are logged in as a user who is not a manager of

the `employee` role. A little earlier you tried to perform an unauthorized access to the application and got a login page back. However, this time you are getting the error—because you were not logged in at all on the previous occasion and now you are. Therefore, the reason for the error is not a lack of authentication, but lack in authorization permissions.

Note: If you want a custom error page for this error, you can map your `ErrorHandler` Servlet to handle the 403 error code. Modify the `web.xml` file's `processRequest` operation to add a handling case for this error as well. Consider this an optional extra task to try if you have time.

e. Navigate back to the `index.jsp` page by inserting the web module context root URL into a browser address bar: `http://localhost:7001/pm`

f. Click the **JSF Product Search** link.

g. Set the Product name to be `%` and click **Find**.

Note: Now you should see the list of products. However, because you have logged in as a user who is not a member of the `employee` role, you do not see the clickable navigation link to the edit page.

6. Log out from the application:

a. Click the **Product Search** link.

b. Click the **Home** link.

c. Click the **Logout** link.

7. Test application access as a member of the **`employee`** role:

a. Click the Login link.

b. Log in as user `jdoe` with password `welcome1.`

c. Click the **JSF Product Search** link.

d. Set the Product name to be `%` and click **Find**.

Note: Now you should see the list of products with the ID property displayed as a link to the edit page.

e. Click any product ID to navigate to the product edit form.

Note: Now you should see the product edit form.

f. Navigate back to the Home page by clicking the **ProductSearch** link and then the **Home** link.

g. On the Home page, click the **Product Search** link to test your access to a Servlet/JSP part of the application.

h. Set the Product name to be `%` and click **Find**.

Note: Now you should see the list of products with the ID property displayed as a link to the edit page.

i. Click any product ID to navigate to the product edit form.

Note: Now you should see product edit form. You may also check the invocation of Discount REST service by clicking the **Check Discount** button.

j. Navigate back to the Home page, by clicking the **ProductSearch** link and then **the Home** link.

8. Test WebSocket Chat Server's handling of logged in usernames.

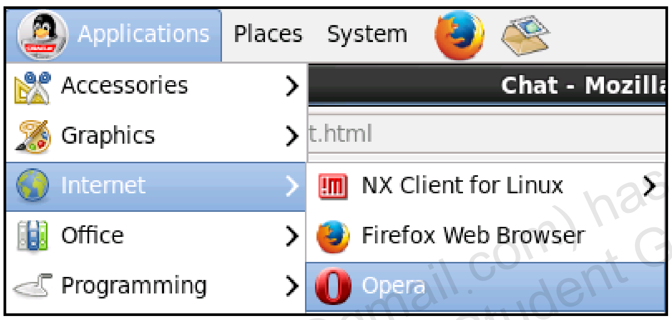
   a. Click the **Open Chat Window** button.

   b. In the Chat window, click **Join**.

      Note: Now you should see messages from the Chat Server welcoming you as user `jdoe`.

   c. Try access WebSocket Chat Server chat application using a different browser. This would allow you to log in as two different users at the same time.

      Hint: To launch Opera browser, select **Applications > Internet > Opera.**



   d. Using the browser window you just launched, navigate to your application Home page: `http://localhost:7001/pm`

   e. Log in as user **`jbloggs`** with password **`welcome1`**.

   f. Click the **Open Chat Window** button.

   g. Put two browser Chat windows side-by-side.

   h. Inside the new Chat window click **Join.** You may have to rejoin chat in another window as well if your Socket session has expired.

   i. Exchange some chat messages.