# 6G80G IBM WebSphere Commerce V8 Programming Essentials Exercises

## Contents


**Exercise 1. Customizing Storefront Pages ..... 5**

Estimated time ..... 5

What this exercise is about ..... 5

What you should be able to do ..... 5

Introduction ..... 6

Requirements ..... 6

Exercise Instructions ..... 7

Part 1: Display the AuroraESite storefront ..... 7

Part 2: Find the JSP files that produce a view ..... 9

Part 3: Advanced Storefront Customization ..... 13

Record store ids & catalog id ..... 13

Map a new view to a JSP file in the Struts configuration file ..... 15

Update the WebSphere Commerce registry with new view information ..... 17

Create a new property file ..... 19

Create a JSP file for use in WebSphere Commerce ..... 21

Create and load access policies to support new views ..... 23

Update a property file to add new store text ..... 25

Test the new view in the WebSphere Commerce Test Environment ..... 26

Add a link to the header ..... 27

**What you did in this exercise? ..... 30**

**Exercise 2. Customizing Store Business Logic ..... 31**

Estimated time ..... 31

What this exercise is about ..... 31

What you should be able to do ..... 31

Introduction ..... 32

Requirements ..... 32

Exercise instructions ..... 33

Part 1: Create new task command interfaces ..... 33

Part 2: Create new task command implementations....38
Part 3: Create a Controller Command Interface....43
Part 4: Create a Controller Command Implementation Class....44
Part 5: Register the new Controller Command....49
Part 6: Load the Access Control Policy....51
Part 7: Modify the JSP to process the Controller command....53
Part 8: Test on the WebSphere Commerce Test Server....57
What you did in this exercise....59
**Exercise 3. Creating an Enterprise JavaBeans in WebSphere Commerce....60**
Estimated time....60
What this exercise is about....60
What you should be able to do....60
Introduction....61
Requirements....61
Exercise instructions....62
Part 1: Create the XPAPERCATALOG table....63
Part 2: Create the XPaperCatalog Entity Bean....65
Part 3: Configure the EJB properties....71
Part 5: Map the database table to the EJB....89
Part 6: Generate the Access Bean for the EJB....99
Part 7: Test the Entity bean....103
What you did in this exercise....109
**Exercise 4. Extending an SOI service with UserData....110**
Estimated time....110
What this exercise is about....110
What you should be able to do....110
Introduction....110
Requirements....110
Exercise Instruction....111
Part 1: Preparing the database for new engraving attributes....111
Part 2: Customizing the business logic....113
Step 3: Updating the order shopping flow to include your customized code....114

Step 4: Customizing the Aurora store to show engraving attributes ..... 116
Step 5: Testing your customization ..... 121
What you did in this exercise ..... 125
**Exercise 5: Creating and customizing REST services ..... 126**
Estimated time ..... 126
What this exercise is about ..... 126
What you should be able to do ..... 126
Introduction ..... 126
Requirements ..... 127
Exercise Instructions ..... 128
Part 1: Preparing the data bean and controller command ..... 128
Part 2: Generating Configuration-Based Mapping Files ..... 131
Part 3: Creating custom REST resource handler to perform mapping ..... 136
Part 4: Verifying new REST service call by using the Poster browser plug-in ..... 140
Part 5: Customizing an existing configuration-based data bean mapping to return more data ..... 147
Part 6: Verifying customized REST service call using the Poster browser plug-in ..... 149
Part 7: Customizing the store widget to use the additional data ..... 152
Part 8: Verifying the store widget customization in the storefront ..... 155
Part 9: Annotating your REST resource handler API to use the Swagger user interface 156
Part 9: Verify your custom REST resource handler API through the Swagger user interface ..... 160
What you did in this exercise ..... 164
**Exercise 6: WebSphere Commerce Build and Deployment ..... 165**
Estimated time ..... 165
What this exercise is about ..... 165
What you should be able to do ..... 165
Introduction ..... 165
Requirements ..... 165
Exercise instructions ..... 166
Part 1: Perform a Build to create a deployment package by using the WCBD tool ..... 166

Part 2: Perform a Deployment with the WCBD tool ..... 170
What you did in this exercise ..... 173

# Exercise 1. Customizing Storefront Pages

## Estimated time

02:00 hours

## What this exercise is about

The objective of this exercise is to demonstrate the development of a new view that can be used within the AuroraESite store. It also demonstrates many of the steps necessary for modifying the behavior of an existing view.

## What you should be able to do

After completing this exercise, you will be able to:

- Find the JSP file that produces a view.
- Inspect the mappings in the Struts configuration file.
- Update a property file to modify store text.
- Review property-level changes in a store.
- Map a new view to a JSP file in the Struts configuration file.
- Update the WebSphere Commerce registry with new view information.
- Construct a new property file.
- Create a JSP file for use in WebSphere Commerce.
- Create and load access policies to support new views.

## Introduction

The exercise demonstrates the necessary steps for creating and accessing a new view, customizing existing views. Unlike basic web applications, a page in WebSphere Commerce must function within an intricate framework of pages, rules, and policies. The first change that you make is rudimentary: changing the title bar in a multi-lingual environment by changing property files.

Then, you build a new view. This view functions as a form for customers to order a paper version of the company's catalog. You then modify an existing page to link to your new view. This linking is accomplished by adding new programmatic commands in a page that uses the Struts framework and the multi-lingual capabilities of WebSphere Commerce.

The exercise is an excellent introduction to the steps necessary to add a view, and its associated property files. The exercise also deals with access control policies.

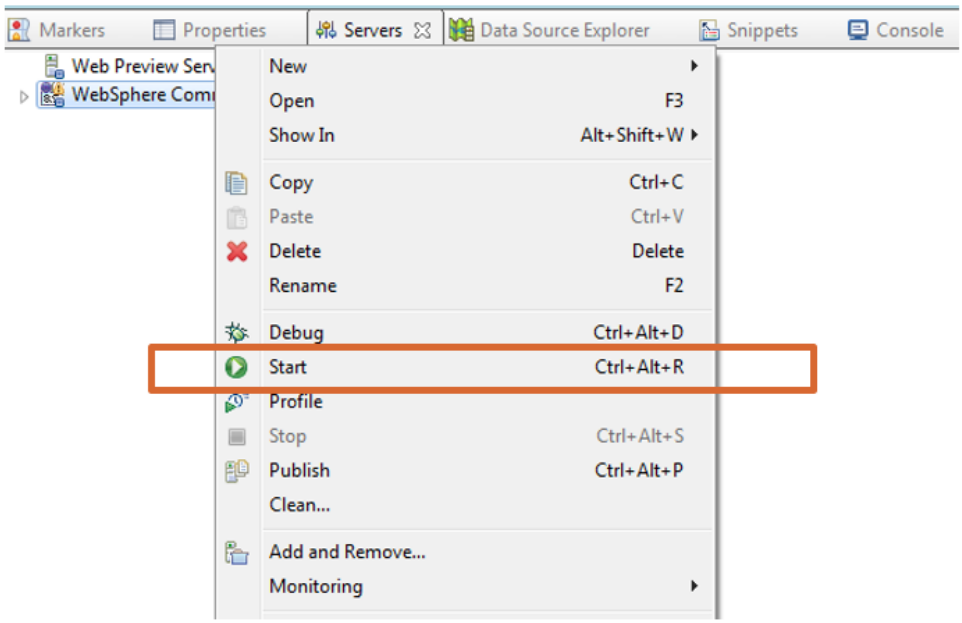
## Requirements

- WebSphere Commerce Developer V8.0.0 Enterprise
- Extended sites store archive published
- AuroraESite store created

# Exercise Instructions

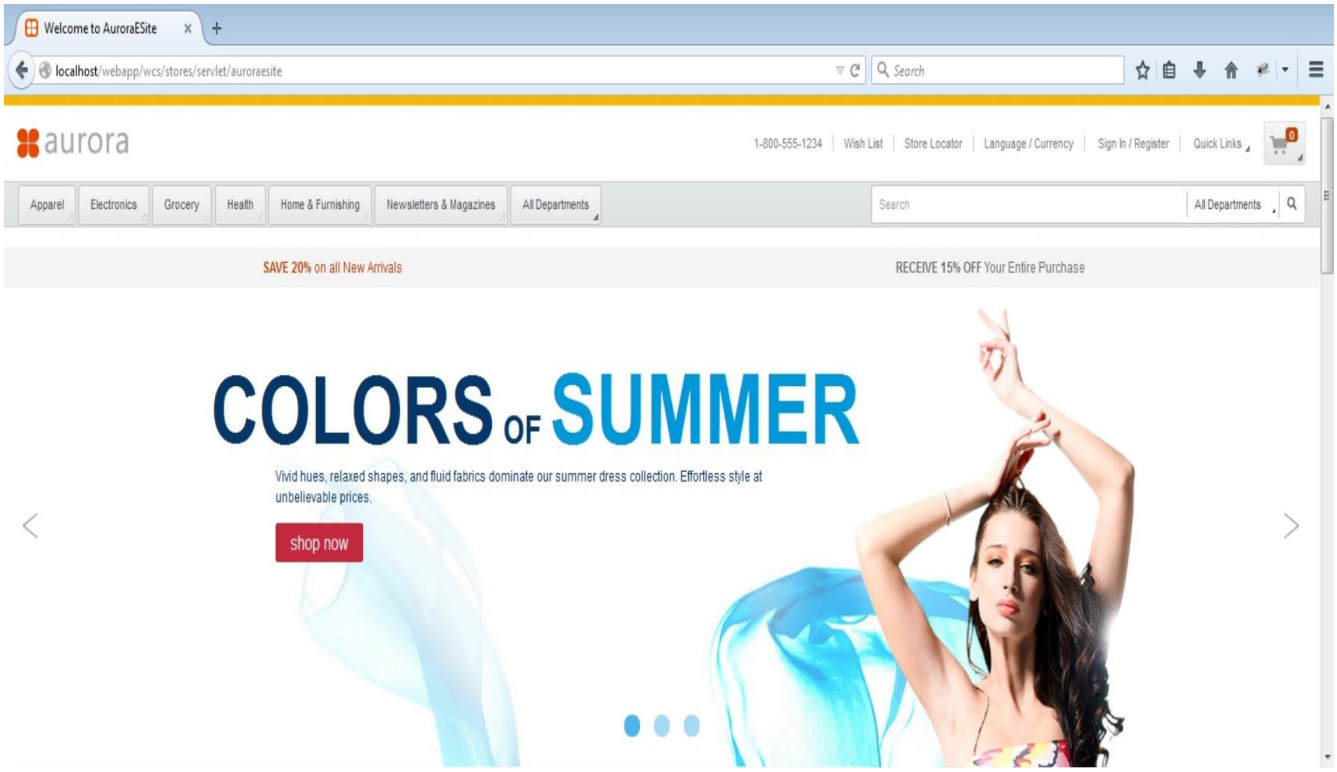
## Part 1: Display the AuroraESite storefront

1. Locate the WebSphere Commerce Developer shortcut on the Desktop and open it to start it.
2. Start WebSphere Commerce Developer.
3. Open the Java EE perspective if it is not already open.
4. Start the WebSphere Commerce Test Server, if it is not already started. The operation is pictured in the following screen capture.




5. Open a browser window and access the following URL:

   http://localhost/webapp/wcs/stores/servlet/auroraesite


Welcome to AuroraESite
localhost/webapp/wcs/stores/servlet/auroraesite
Search
aurora
1-800-555-1234
Wish List
Store Locator
Language / Currency
Sign In / Register
Quick Links
Apparel
Electronics
Grocery
Health
Home & Furnishing
Newsletters & Magazines
All Departments
Search
All Departments
SAVE 20% on all New Arrivals
RECEIVE 15% OFF Your Entire Purchase
COLORS OF SUMMER
Vivid hues, relaxed shapes, and fluid fabrics dominate our summer dress collection. Effortless style at unbelievable prices.
shop now

## Part 2: Find the JSP files that produce a view

1. In the browser, right-click and click **View Source** to view the source of the page. In the beginning of the source, you will find the name of the JSP:

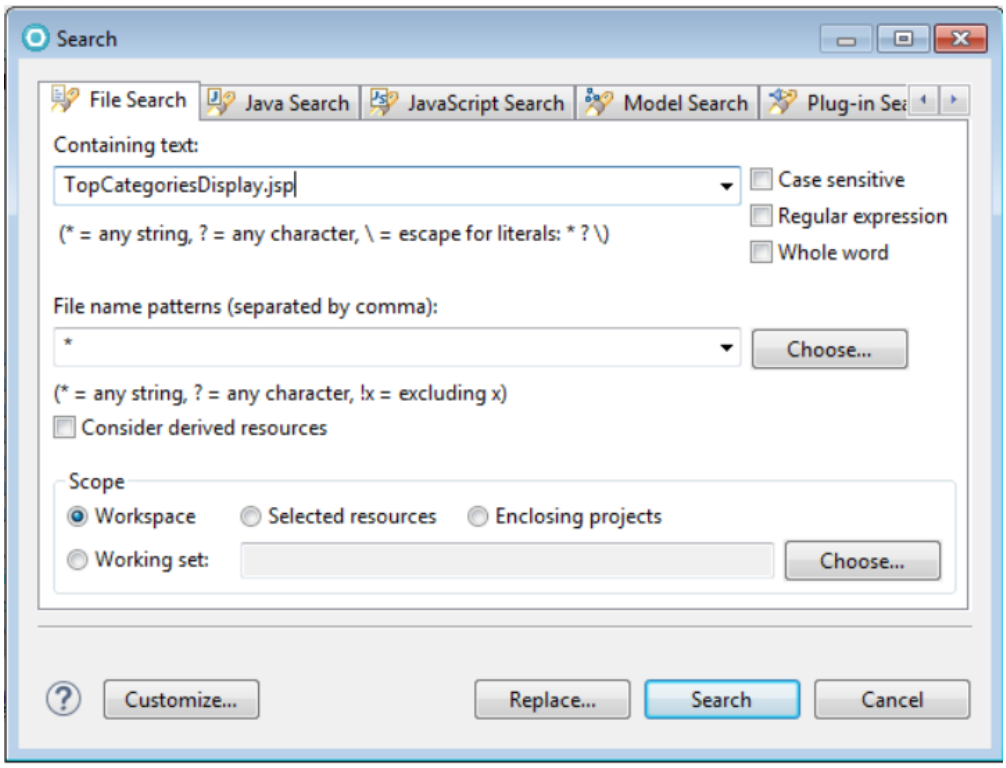
```
<!doctype HTML>

<!-- BEGIN TopCategoriesDisplay.jsp -->

<html xmlns:wairole="http://www.w3.org/2005/01/wai-rdf/GUIRoleTaxonomy#"

xmlns:waistate="http://www.w3.org/2005/07/aaa" lang="en" xml:lang="en">
    <head>
```

   Based on this comment, the source of the page is found in the **TopCategoriesDisplay.jsp**, among others.

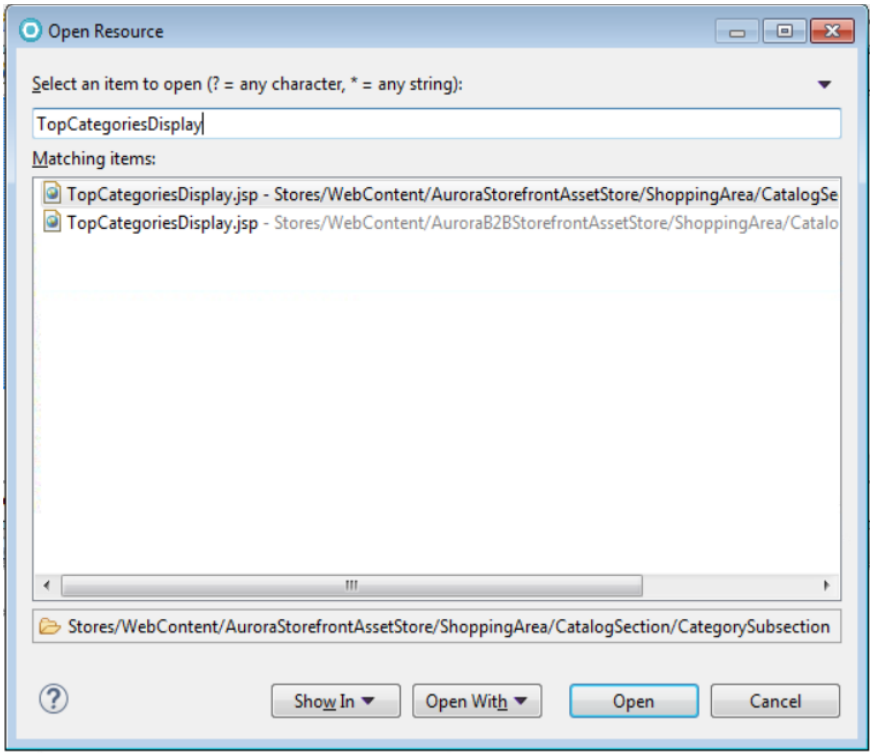
2. In the Enterprise Explorer view, in the **AuroraStorefrontAssetStore** folder, navigate to the **TopCategoriesDisplay.jsp**.
   a) Expand **Stores > Web Content > AuroraStorefrontAssetStore > ShoppingArea > CatalogSection,** and finally **CategorySubsection**.
   b) Double-click **TopCategoriesDisplay.jsp** to open it in the JSP Editor.
   c) Click the **Source** tab.

---

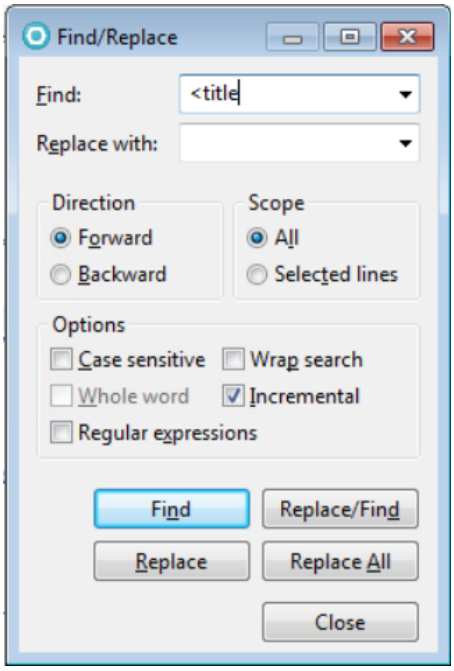
**Note:** It is also possible to find this file by using the search features of WebSphere Commerce Developer. Click **Search > File** from the toolkit window, and enter **TopCategoriesDisplay.jsp** as the file name pattern in the File Search tab.




Alternatively, you can use the Open Resource dialog to open the file. You can open the using the keyboard shortcut Ctrl + Shift + R or Navigate > Open Resource in the menu.



d) Press **Ctrl+F** to search, and enter "**<title**" as the search text. Click **Find** until the title element is displayed, then click **Close**.



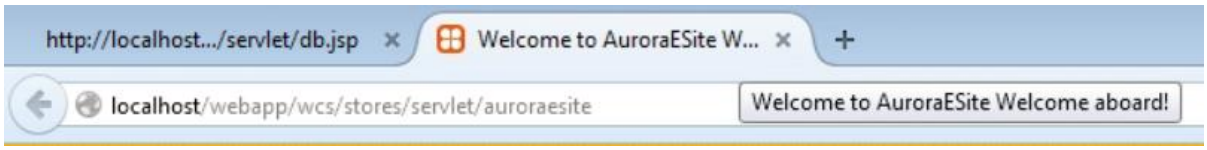

e) Modify the title element to add some text after the **<title>** tag. For example, you could add "Welcome aboard!":

```
<title><c:out value="${pageTitle}"/> Welcome aboard!</title>
```

f) Press **Ctrl+S** to save the file.

3. Test the change in a browser. Open a browser and navigate to the following URL:

http://localhost/webapp/wcs/stores/servlet/auroraesite




This exercise is a simple demonstration of the ability to modify a page in WebSphere Commerce, but the principle is always the same: find the page that needs to be modified and insert the necessary elements with HTML, JSP, JSTL, or WebSphere Commerce specific syntax.

## Part 3: Advanced Storefront Customization

In this part of this exercise, you create a link on the **TopCategoriesDisplay.jsp**. This task leads users to a new page from which the users can order a paper version of the store catalog.

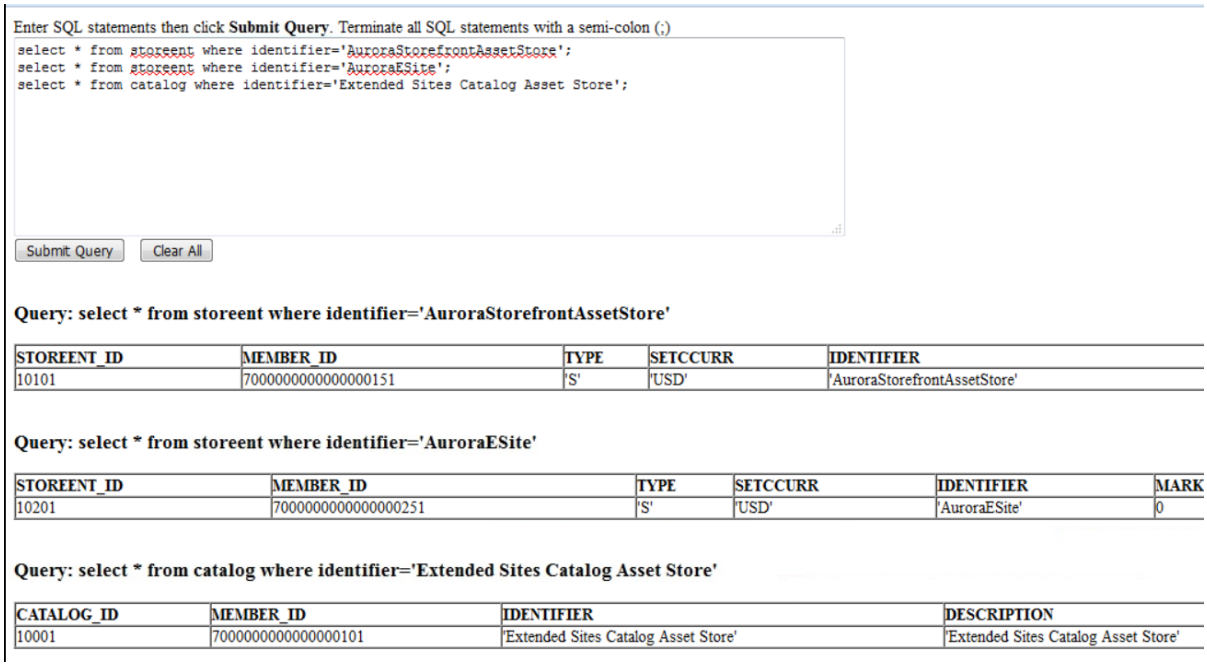
For now, this new JSP file holds sample data, and does not have any functionality. You build this functionality in later exercises.

## Record store ids & catalog id

1. If the WebSphere Commerce server is not running, then start it. Once the server is started, open a web browser to the following URL:
   http://localhost/webapp/wcs/admin/servlet/db.jsp

Enter the following SQLs, select **Submit Query**, and note the **storeent_id** & **catalog_id**.

```
select * from storeent where identifier =
'AuroraStorefrontAssetStore';

select * from storeent where identifier = 'AuroraESite';

select * from catalog where identifier='Extended Sites Catalog Asset
Store';
```



Enter SQL statements then click **Submit Query**. Terminate all SQL statements with a semi-colon (;)

```
select * from storeent where identifier='AuroraStorefrontAssetStore';
select * from storeent where identifier='AuroraESite';
select * from catalog where identifier='Extended Sites Catalog Asset Store';
```

Submit Query | Clear All

**Query: select * from storeent where identifier='AuroraStorefrontAssetStore'**

| STOREENT_ID | MEMBER_ID | TYPE | SETCCURR | IDENTIFIER |
|---|---|---|---|---|
| 10101 | 7000000000000000151 | 'S' | 'USD' | 'AuroraStorefrontAssetStore' |

**Query: select * from storeent where identifier='AuroraESite'**

| STOREENT_ID | MEMBER_ID | TYPE | SETCCURR | IDENTIFIER | MARK |
|---|---|---|---|---|---|
| 10201 | 7000000000000000251 | 'S' | 'USD' | 'AuroraESite' | 0 |

**Query: select * from catalog where identifier='Extended Sites Catalog Asset Store'**

| CATALOG_ID | MEMBER_ID | IDENTIFIER | DESCRIPTION |
|---|---|---|---|
| 10001 | 7000000000000000101 | 'Extended Sites Catalog Asset Store' | 'Extended Sites Catalog Asset Store' |

**Note:** If you are using a different environment, then the store id and catalog id values may be different.

## Map a new view to a JSP file in the Struts configuration file

1. Add a mapping to the Struts configuration file.
   a) Launch WebSphere Commerce Developer if not already open.
   b) In the Enterprise Explorer view, navigate to **Stores > WebContent > WEB-INF**.
   c) Right-click the **struts-config-ext.xml** file and then, click **Open With > XML Editor.**

2. Create a new Global Forward to route to the new view.

   a) Add the following line just before the **</global-forwards>** element. This creates a new struts entry for **OrderPaperCatalogView/10101** mapped to **/OrderPaperCatalog.jsp**.

   **Note: 10101** is the id of **AuroraStorefrontAssetStore** store. It may be different in your environment, you can verify the value by querying the **STOREENT** table.

```
<forward
className="com.ibm.commerce.struts.ECActionForward"
name="OrderPaperCatalogView/10101"
path="/OrderPaperCatalog.jsp"/>
```

   **Note:** This JSP does not exist yet; it is created in a later section.

   b) Next, after the **<action-mappings>** element, add an action path mapping for our paper catalog request to map the request to the struts BaseAction class.

```
<action path="/OrderPaperCatalogView"
type="com.ibm.commerce.struts.BaseAction"/>
```

The changes should resemble the following screenshot:

```
890
891 <forward className="com.ibm.commerce.struts.ECActionForward" name="OrderPaperCatalogView/10101" path="/OrderPaperCatalog.jsp"/>
892
893 </global-forwards>
894 <action-mappings type="com.ibm.commerce.struts.ECActionMapping">
895
896 <action path="/OrderPaperCatalogView" type="com.ibm.commerce.struts.BaseAction"/>
```

c) Press **Ctrl+S** to save your work.

## Update the WebSphere Commerce registry with new view information

For the changes done in the previous section in **struts-config-ext.xml** to reflect in the store, you need to either restart the server or refresh the Struts registry using WebSphere Commerce Administration Console. In this section, we will refresh the registry.

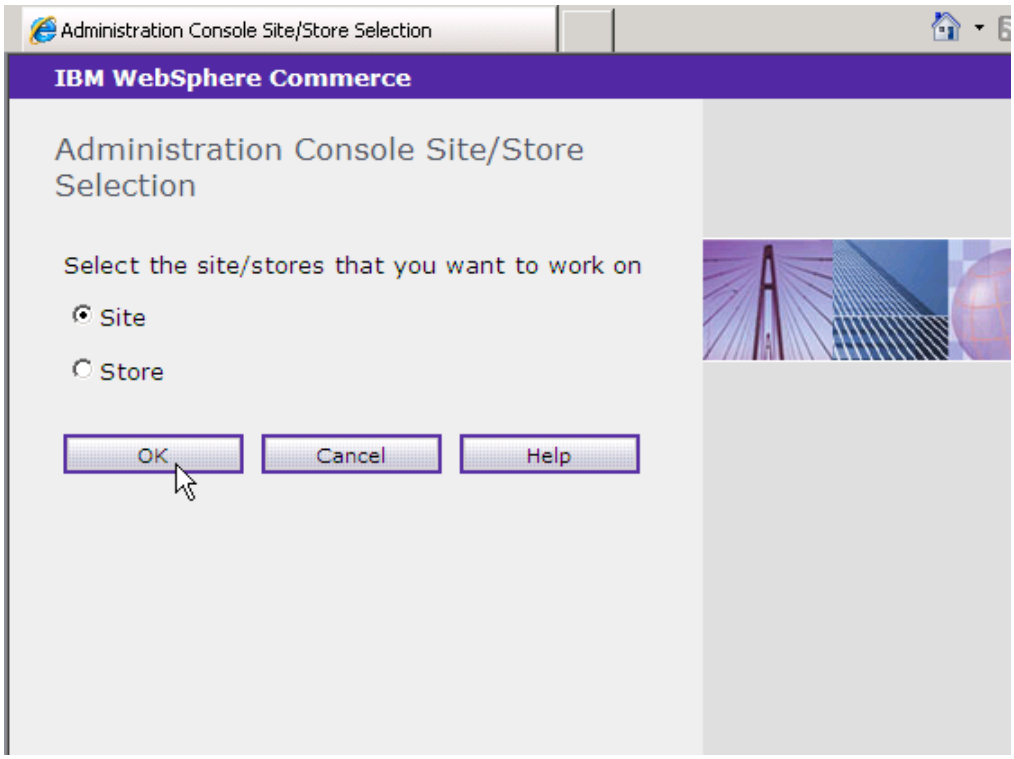
1. Open Internet Explorer and select Favorites.

**Note:** Only Internet Explorer is supported for Administration Console.

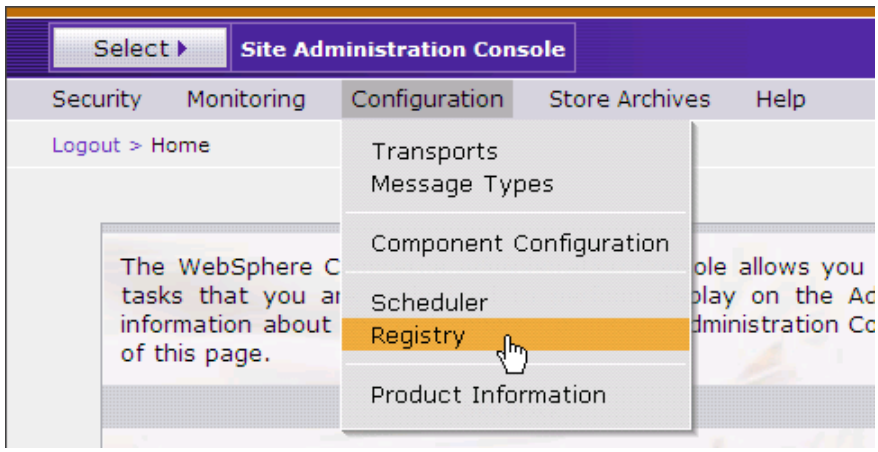
2. Select the Administration Console link. Or you can use the following link:

https://localhost:8002/webapp/wcs/admin/servlet/ToolsLogon?XMLFile=adminconsole.AdminConsoleLogon

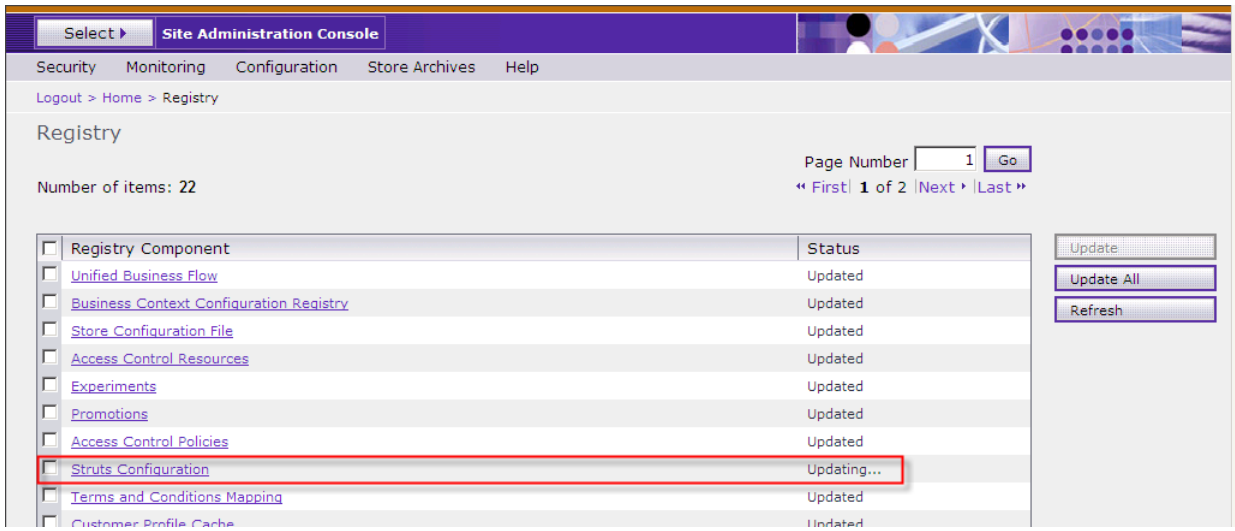
3. If prompted regarding a Certificate Error, select “Continue to this website (not recommended)”.
4. Log in to the Commerce Administration Console by using user ID **wcsadmin** and password **wcsadmin**.
5. Click **Site** on the **Administration Console Site/Store Selection** page and click OK.




6. From the **Configuration** menu, click **Registry**.




7. A list of registry components for all sites is displayed.
8. Select the **Struts Configuration** registry component check box and click **Update**. The Registry window reloads, listing the status for the selected components as **Updating…**



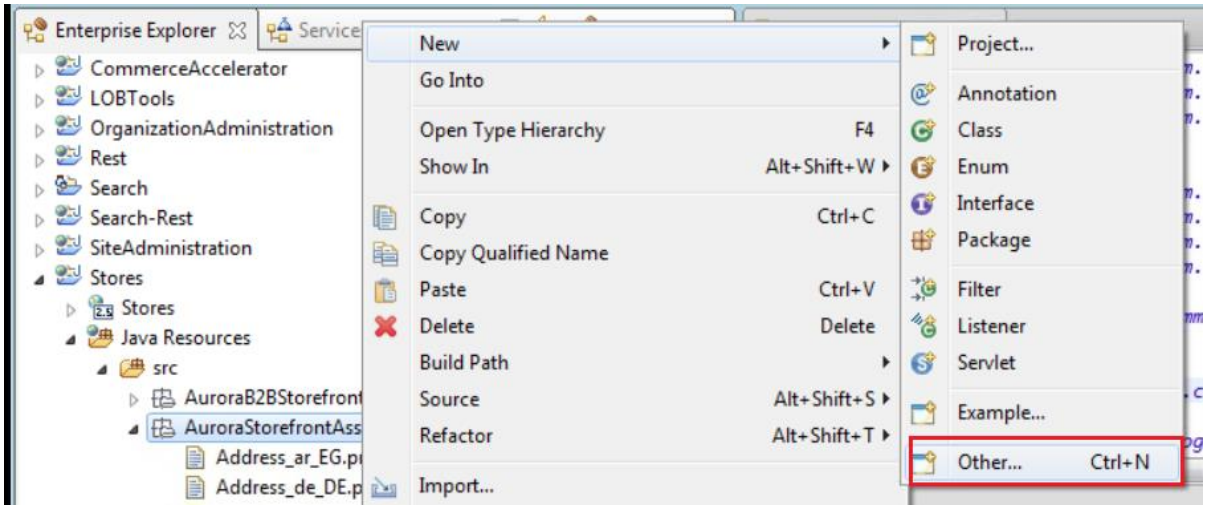

9. Click **Refresh** to reload the Registry window. When the update is complete, the status column for the Struts Configuration entry reads **Updated**.
10. Log out of the WebSphere Commerce Administration Console.

## Create a new property file

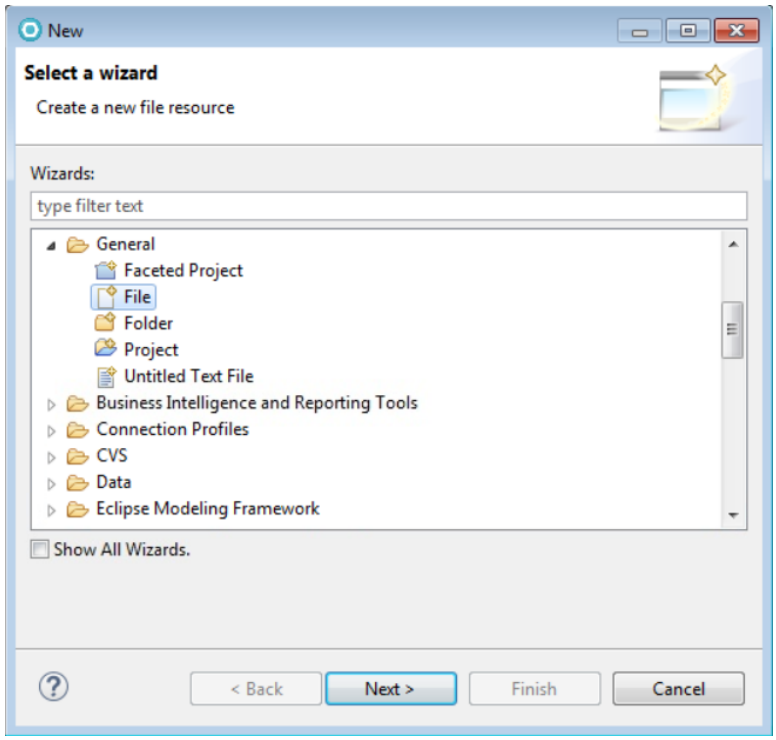
1. Add a properties file.

   a) In the Java EE perspective, Enterprise Explorer view, navigate to the **Stores > Java Resources > src > AuroraStorefrontAssetStore** folder.

   b) With the **AuroraStorefrontAssetStore** folder highlighted, right-click and then click **New > Other** to open the **New** dialog.




   c) Expand the **General** folder and select **File**. Click **Next**.



d) In the File Name field, enter **OrderCatalogNLS_en_US.properties**

e) Click **Finish**.

f) A properties file editor opens for the new file; type the following text:

```
# -- ORDER CATALOG SECTION -- #

OrderCatalogTitle = Order a catalog

OrderCatalogHeader = Order a catalog for home delivery!

OrderCatalogMessage = Please enter your mailing address for
quick home delivery
```

g) Save by using **Ctrl+S**.

2. Create the default properties file.

a) Right-click the new **OrderCatalogNLS_en_US.properties** and click **Copy**.

b) Right-click the file again and click **Paste**.

c) In the **Name Conflict** dialog, change the name to: **OrderCatalogNLS.properties**. This creates the default properties file that acts as a fallback if a certain property key is not found in the locale specific file.

---

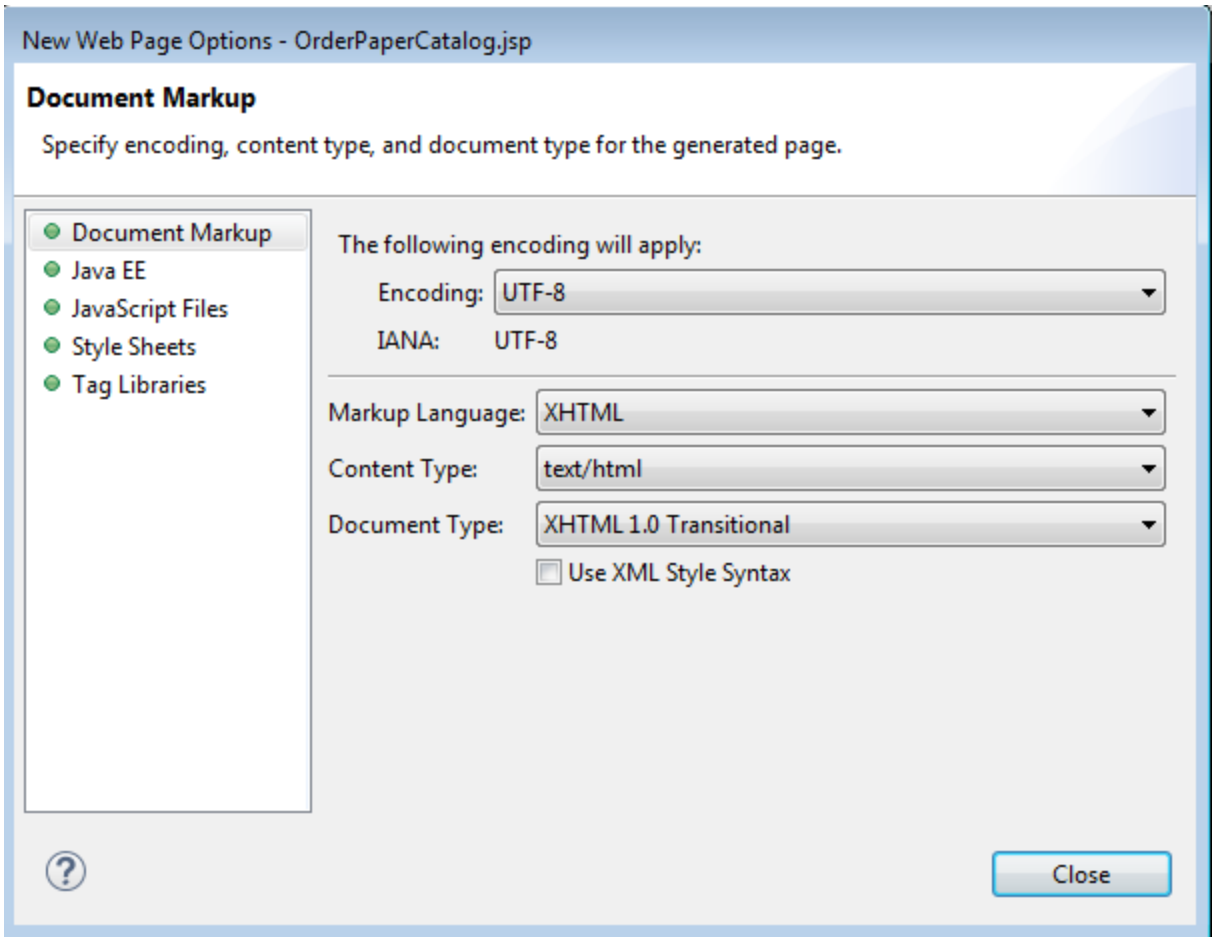
**Note**: In the properties folder, you see that there is a file for each locale that is supported, such as fr_FR, es_MX, en_US, as well as the default (no-locale) version.

---

d) Click **OK**.

## Create a JSP file for use in WebSphere Commerce

1. Create a JSP file, **OrderPaperCatalog.jsp** for the **AuroraStorefrontAssetStore** store.
    a) In the Enterprise Explorer view, navigate to the **Stores > WebContent > AuroraStorefrontAssetStore** directory.
    b) Right-click the **AuroraStorefrontAssetStore** folder. From the pop-up menu, select **New > Web Page** to create the JSP page in this folder. The web page wizard is launched.
    c) Specify values for the new file, as follows:
        i. In the **File name** field, type **OrderPaperCatalog.jsp**
        ii. From the template dialog, select **Basic Templates > JSP.**
    d) Click **Options**.
    e) From the **Encoding** list, select **UTF-8**.
    f) Under **Document Markup**, from the **Markup Language** list, select **XHTML**.
    g) From the **Content Type** list, select **text/html**.
    h) From the **Document Type** list, select **XHTML 1.0 Transitional**.




i) Click **Close**.

j) On the **New Web Page** dialog, click **Finish**. The **OrderPaperCatalog.jsp** file opens for editing.

k) Click the **Design, Source, and Split** tabs for different views of the file.

Using the **Source** pane, cut and paste the contents from the **OrderPaperCatalog.jsp** file that was provided with the exercise. This file is in the **C:\exercises\customizing-storefront-pages** folder.

l) Save your work and continue.

## Create and load access policies to support new views

1. Stop the WebSphere Commerce test environment. If it is started, stop the test environment by right-clicking **WebSphere Commerce Test Server** in the **Server** view of the Java EE perspective and selecting **Stop**.

---

**Note:** If **Apache Derby** is used as the development database, then it is required to stop the WC server as only one connection can be made to the database at a time. If we don't stop the server, then **acpload** will be unable to connect to the database. If you use **DB2**, then there is no need to stop the server before running **acpload**.

---

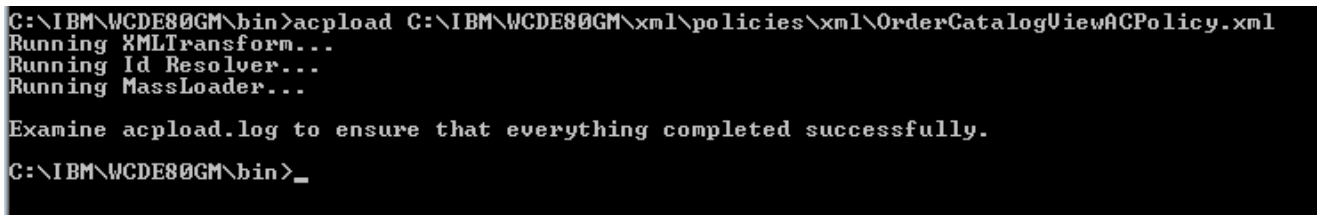
2. Run the **acpload** utility on the **OrderCatalogViewACPolicy.xml** file that is supplied with the exercises, found in **C:\exercises\customizing-storefront-pages**.

    a) Copy **OrderCatalogViewACPolicy.xml** to the **C:\IBM\WCDE80GM\xml\policies\xml** folder.

    b) Start a command prompt and navigate to the following directory: **C:\IBM\WCDE80GM\bin**.

    c) From this directory, run the **acpload** command.

```
acpload OrderCatalogViewACPolicy.xml
```




---

**NOTE**: The **acpload** utility is used to load access control policies, which are written in an XML format, into the database. The load utility takes the following format:

```
acpload db_name db_user db_password xmlFile db_schema
```

If you use Derby as the development database, then the database details do not need to be provided to the utility. But if you use DB2, then the database details need to be provided as shown in the above syntax.

Customized access control policies should always be placed in the ***<WC_installdir>*/xml/policies/xml** directory.

---

    d) Navigate to the **C:\IBM\WCDE80GM\logs** directory. Inspect the **acpload.log** and **messages.txt** files to ensure that the access control

policy loaded successfully. The messages.txt file might not exist if the load completed successfully. Also, check whether the policy files were created successfully in the **C:\IBM\WCDE80GM\xml\policies\xml** directory: **OrderCatalogViewACPolicy_idres.xml** and **OrderCatalogViewACPolicy_xmltrans.xml**. These two files are created as part of a successful **idresgen** utility process. The error files in this directory, if any, indicate that there was a problem.

## Update a property file to add new store text

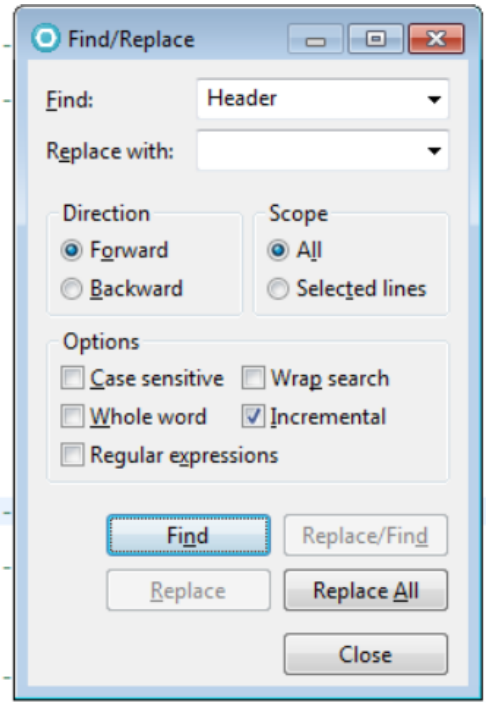
1. Navigate to **Stores > Java Resources: src > AuroraStorefrontAssetStore**.

2. Double-click the file **storetext_v2_en_US.properties**. This file is used if the user's locale preference in the store is US English or if it is the store's default language.

3. Double-click the **storetext_v2.properties**. This file acts as a fallback if a certain properties key is not found in the locale specific properties file.

4. Modify the **storetext_v2.properties** file to add a property.
   a. Press **Ctrl+F** and type **Header**. Click **Find**. This action pulls the file up to the section about Header Component section.
   b. Add a property to this set with the name **ORDER_PAPER_CATALOG** with the text: **ORDER A CATALOG**.

```
42
43 #----------------------------------------------------------
44 # Header Component
45 #----------------------------------------------------------
46 HEADER_SHOPPING_LIST = Wish List
47 HEADER_STORE_LOCATOR = Store Locator
48 HEADER_SIGN_IN = Sign In
49 HEADER_SIGN_IN_REGISTER = Sign In / Register
50 HEADER_MY_ACCOUNT = My Account
51 HEADER_SIGN_OUT = Sign Out
52 HEADER_SIGN_OUT_USERNAME = Sign Out, {0}
53 HEADER_QUICK_LINKS = Quick Links
54 BCT_HOME = Home
55 DIVIDING_BAR = |
56 COLON_SYMBOL = :
57 ATTRNAMEKEY = {0}:
58 ORDER_PAPER_CATALOG = ORDER A CATALOG
59
60 #----------------------------------------------------------
61 # ShopCart Component
62 #----------------------------------------------------------
63 MINI_SHOPCART_SUBTOTAL = Subtotal
64 MINI_SHOPCART_SHOPPING_CART = Shopping Cart
65
66 #----------------------------------------------------------
```




   c. Press **Ctrl+S** to save your work.

5. Add the same property to **storetext_v2_en_US.properties**.

## Test the new view in the WebSphere Commerce Test Environment

1. Start the test environment by right-clicking WebSphere Commerce Test Server in the **Server**s view of the Java EE perspective and selecting Start.
2. In the web browser, enter the following web address:

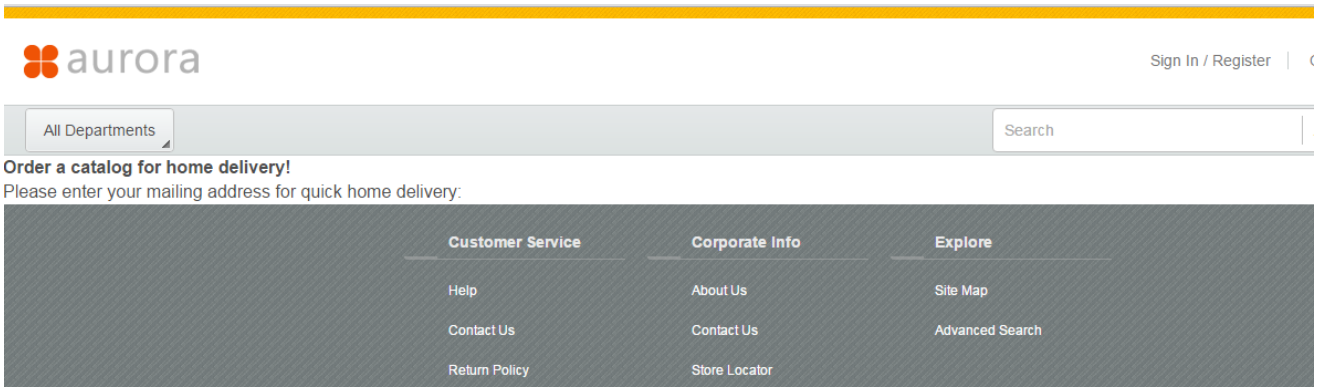
   http://localhost/webapp/wcs/stores/servlet/OrderPaperCatalogView?storeId=10201&catalogId=10001

   A new view is opened.

---

**Note: 10201** is the id of **AuroraESite** store. It may be different in your environment, you can verify the value by querying the **STOREENT** table. **10001** is the id of the **Extended Sites Catalog Asset Store** master catalog. It may different in your environment, you can verify the value by querying the **CATALOG** table.

---



## Add a link to the header

Now that the new JSP is functional, you can add a link to the header. By adding the link, you can access the page without the need for manually entering the web address.

You recall that the TopCategoriesDisplay.jsp was made up of smaller fragments. One such fragment is the header. This header contains the links for accessing the home page, Shopping Cart, and so on. You can add an entry to this list, which accesses the new JSP page.

1. Open **/Stores/WebContent/AuroraStorefrontAssetStore/Widgets/Header/Header_Data.jspf**.
2. At the end of the file, add a variable to construct the URL to view by using the following syntax:

```
<wcf:url var="OrderPaperCatalogURL" value="OrderPaperCatalogView">
  <wcf:param name="langId" value="${langId}" />
  <wcf:param name="storeId" value="${storeId}" />
  <wcf:param name="catalogId" value="${catalogId}" />
</wcf:url>
```

   This code establishes a programmable URL with variable language, store, and catalog ID numbers.

   **Note:** The code can be also copied from **C:\exercises\customizing-storefront-pages\CodeSnippet1.txt**.

3. Open **Stores/WebContent/AuroraStorefrontAssetStore /Widgets/Header/Header_UI.jspf**. Press **Ctrl + F** and find where the code <ul id=*"quickLinksBar"*> is used. Below this code, add a link to the header, by using the variable you declared previously and the formatted message you created in an earlier step:

```
<li>

<a href="<c:out value="${OrderPaperCatalogURL}"/>"
class="panelLinkSelected" id="WC_HeaderDisplay_Link_catalog">

<fmt:message key="ORDER_PAPER_CATALOG" bundle="${storeText}" />

</a>

</li>
```

The code should resemble the following screenshot:

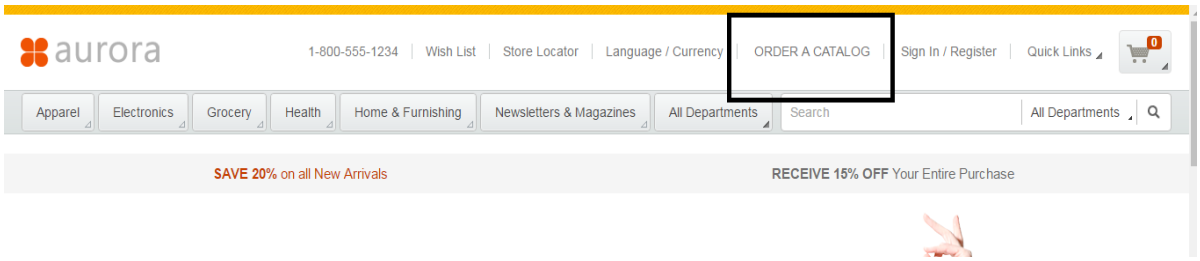
```
</div>
<ul id="quickLinksBar">
<li>
    <a href="<c:out value="${OrderPaperCatalogURL}"/>" class="panelLinkSelected" id="WC_HeaderDisplay_Link_catalog">
        <fmt:message key="ORDER_PAPER_CATALOG" bundle="${storeText}" />
    </a>
</li>
    <c:if test="${!env_globalLoginViewDisabled eq true}">
```

**Note**: The code can be copied from **C:\exercises\customizing-storefront-pages\CodeSnippet2.txt**.

4. Save the file.
5. If you updated the JSPF files using Windows File Explorer, refresh the **Stores** project in RAD, so that the changes are picked up and built.
6. Publish your changes to the server.
   a) Select the server from the **Servers** view.
   b) Right-click the server and click **Publish**.
7. Navigate to

   http://localhost/webapp/wcs/stores/servlet/auroraesite

   You should notice a new link in the header:




8. Click the link and you are directed to your new page.

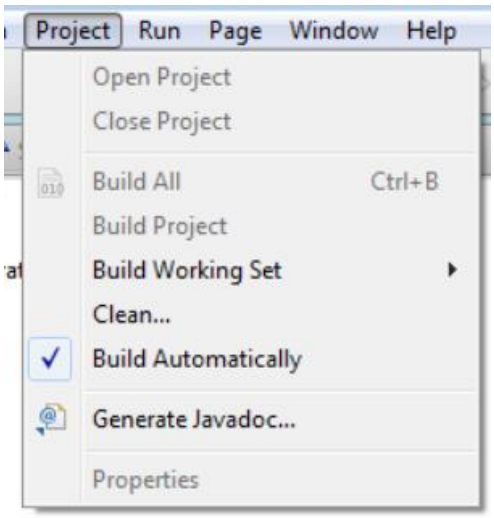
**Troubleshooting:** If the text in the header appears as ???ORDER_PAPER_CATALOG???, then follow these troubleshooting steps.

In WebSphere Commerce Developer, select the **WebSphere Commerce Test Server**, right-click it, and then click **Restart**.

If the problem persists after you restart the server, the project might need to be refreshed.

In WebSphere Commerce Developer, select the **Stores** project, right-click, and click **Refresh**. Click **Project > Clean...** In the Clean window, select the **Clean projects that are selected below** option. Select **Stores** from the list. Select **Build only the selected projects** and click **OK**.

Also, ensure that **Project > Build Automatically** is selected. This will ensure that projects are built automatically whenever any code changes are made.




The build takes several minutes to complete. Once completed, select **the WebSphere Commerce Test Server,** right-click, and click **Publish.**

Publishing to the server also takes several minutes to complete. Once it is completed, select the WebSphere Commerce Test Server, right-click, and click **Restart**.

Refresh the webpage. The same troubleshooting steps can be used in future steps if the changes in the properties files are not displayed on the page.

---

# What you did in this exercise?

This exercise demonstrated the steps necessary to create a view in WebSphere Commerce. A new order paper catalog view was created, along with the required access control policy and property bundle.

In a realistic setting, a developer might extend the page, by using REST Services to display a list of new products or sales items.

# Exercise 2. Customizing Store Business Logic

## Estimated time

02:00 hours

## What this exercise is about

The objective of this exercise is to demonstrate the creation of WebSphere Commerce commands.

## What you should be able to do

After completing this exercise, you will be able to:

- Create new task command interfaces.
- Create new task command implementations.
- Create a controller command interface.
- Create a controller command implementation class.
- Register the new controller command.
- Load access control policy for controller command.
- Modify the JSP file to call the controller command.

## Introduction

The exercise demonstrates the creation of new controller and task commands, as well as the steps necessary for a developer to register and call the logic in the new commands.

In the previous exercise, you built a JSP page for ordering a paper-version of the company catalog for home delivery. In this exercise, you build the corresponding business logic.

The business logic is composed of two tasks, which are implemented as task commands. One to email the catalog desk to place a new request for a catalog, and one to place an indicator in the customer's profile as a reminder to ship new catalogs once they become available. These two tasks are called from a single Controller Command.

It is important to note that you do not implement functional Business Logic yet. The task commands function as placeholders for more complex logic that is not yet built. The purpose of this exercise is to introduce you to the principles of customizing Business Logic.

As well, this exercise presupposes that you make some simple changes to your new JSP. You create a "Submit" button to submit the new catalog order.

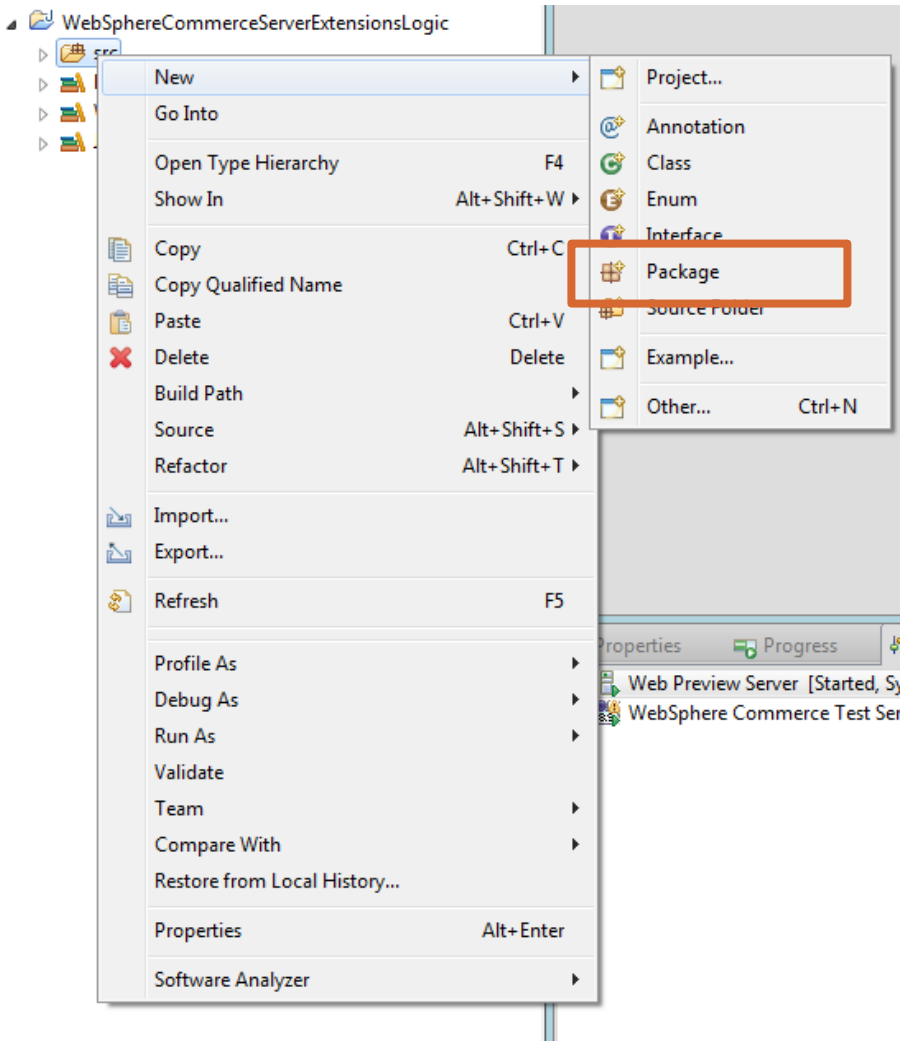
## Requirements

- WebSphere Commerce Developer V8 Enterprise
- Extended sites store archive published
- AuroraESite store created
- Completion of previous exercise, *Customizing Storefront Pages*

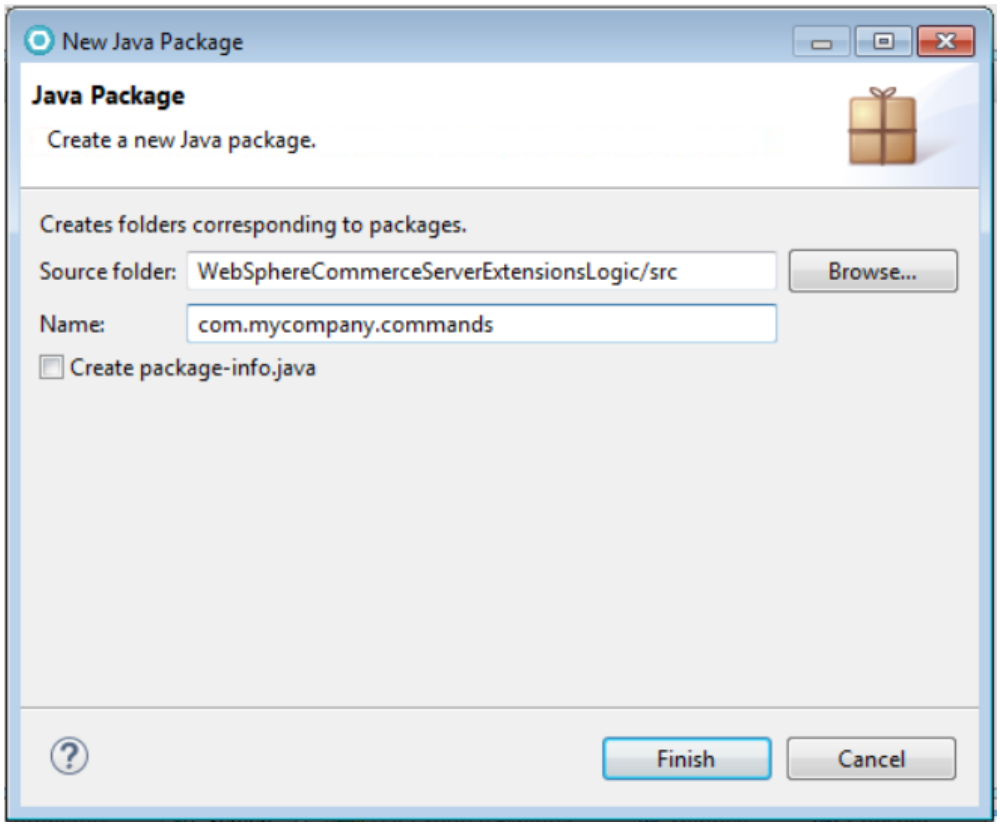
# Exercise instructions

## Part 1: Create new task command interfaces

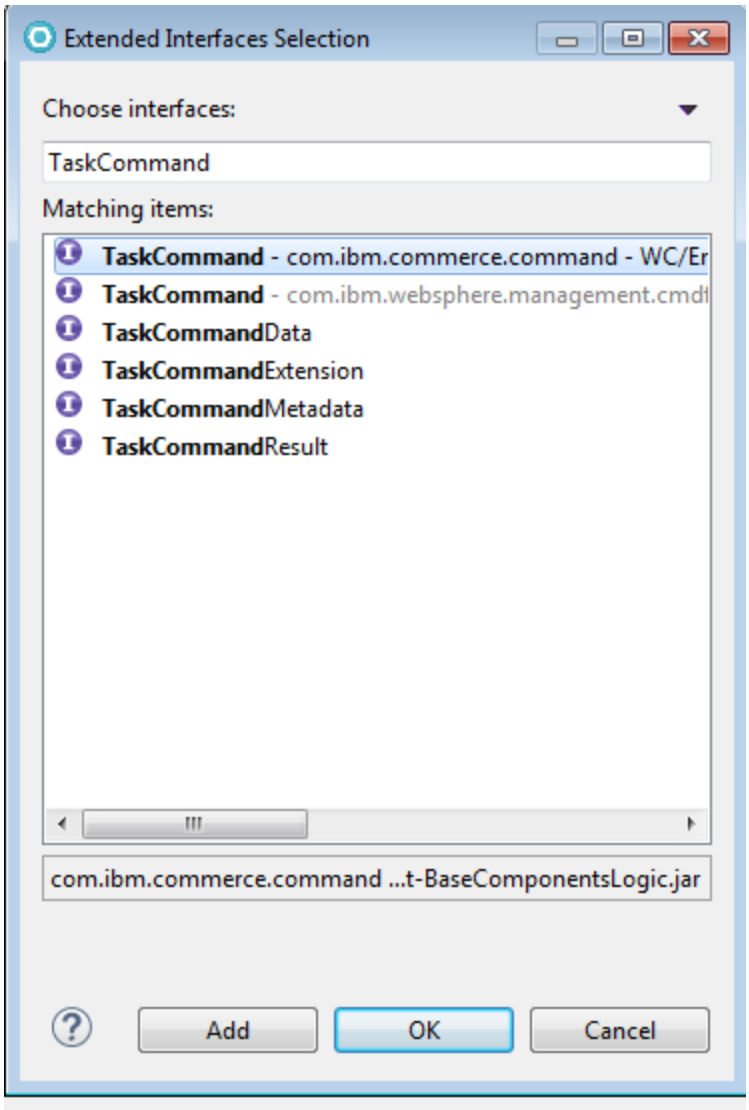
1. Start WebSphere Commerce Developer. Make sure that you are in the Java EE perspective.
2. In the Enterprise Explorer View, navigate to **WebSphereCommerceServerExtensionsLogic** project.
3. Create a package, **com.mycompany.commands**, and a new interface:
   a) Navigate to the **src** directory, right-click, and click **New > Package**.



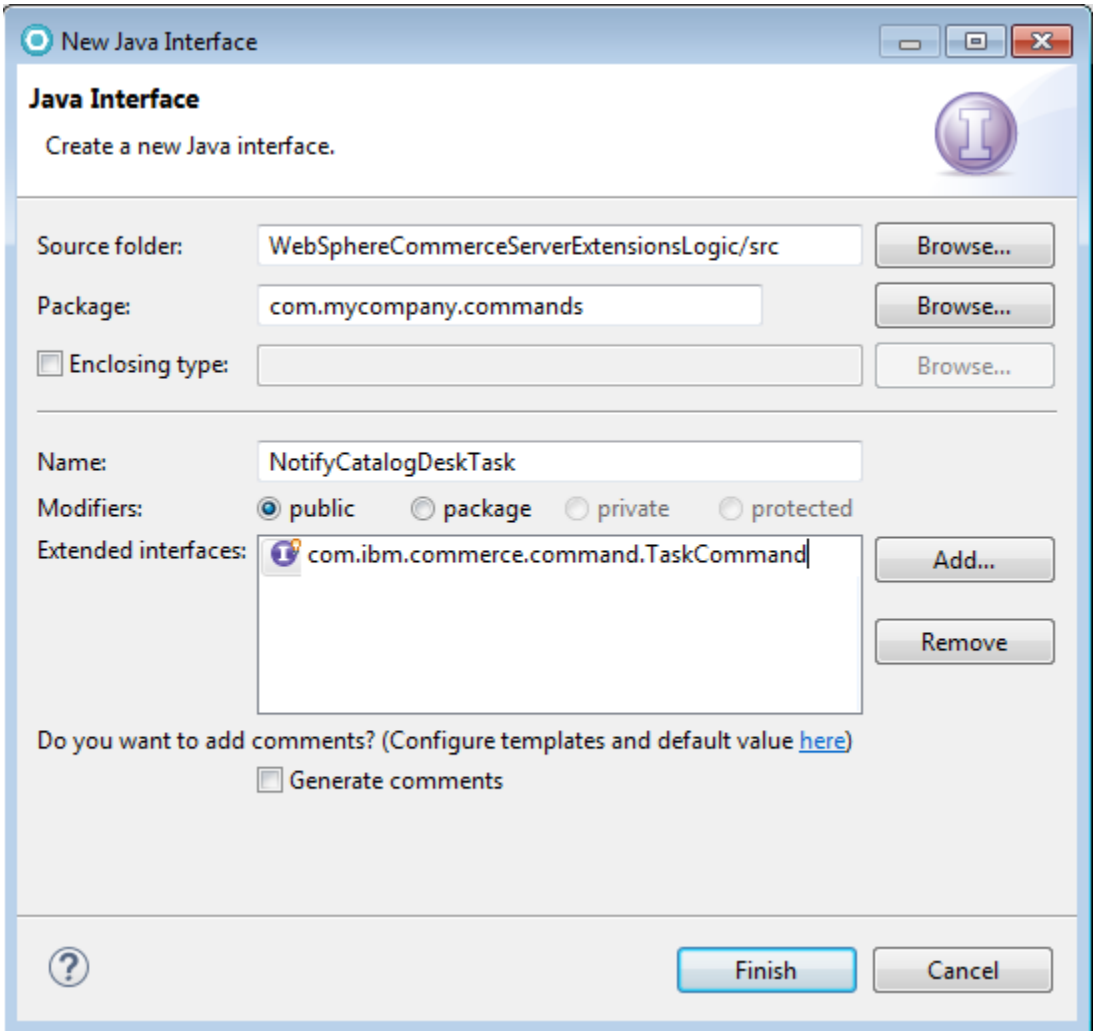

   b) In the resulting dialog box, enter a package name of **com.mycompany.commands** and click **Finish**.




4. Create an interface named **NotifyCatalogDeskTask** that extends **TaskCommand**.
   a) Right-click the newly created **com.mycompany.commands** package and click **New > Interface**.
   b) In the resulting dialog box, supply a name of **NotifyCatalogDeskTask**.
   c) Click **Add** to extend this new interface from an existing interface.
   d) Begin typing **TaskCommand** in the **Extended Interfaces Selection** dialog box. As you type, the editor automatically searches and filters the results for you. Select **TaskCommand - com.ibm.commerce.command** from the list and click **OK**.




e) Verify that your new interface resembles the following and then click **Finish**.




5. The interface will, by default, call an implementation class. Supply the name of this class by adding the following line to the interface:

```
static final String defaultCommandClassName =
"com.mycompany.commands.NotifyCatalogDeskTaskImpl";
```

6. Save the interface.
7. Repeat the preceding steps to create another task command interface that is named **AddUserToCatalogMailingListTask** that extends the **TaskCommand** interface.
8. Add the default implementation class in the interface:

```
static final String defaultCommandClassName =
"com.mycompany.commands.AddUserToCatalogMailingListTaskImpl";
```

9. Add interface methods to each of the new interfaces.

- Add the following interface methods to **NotifyCatalogDeskTask**.

```
public java.lang.String getCustomerName();
public java.lang.String getMailingAddress();
public java.lang.String getCatalogId();

public void setCustomerName(java.lang.String customerName);
public void setCatalogId(java.lang.String catalogId);
public void setMailingAddress(java.lang.String
mailingAddress);
```

- Add the following interface methods to **AddUserToCatalogMailingListTask**.

```
public java.lang.String getCustomerId();
public java.lang.Boolean getIsMailingListMember();
public void setCustomerId(java.lang.String customerId);
public void setIsMailingListMember(java.lang.Boolean
isMember);
```

---

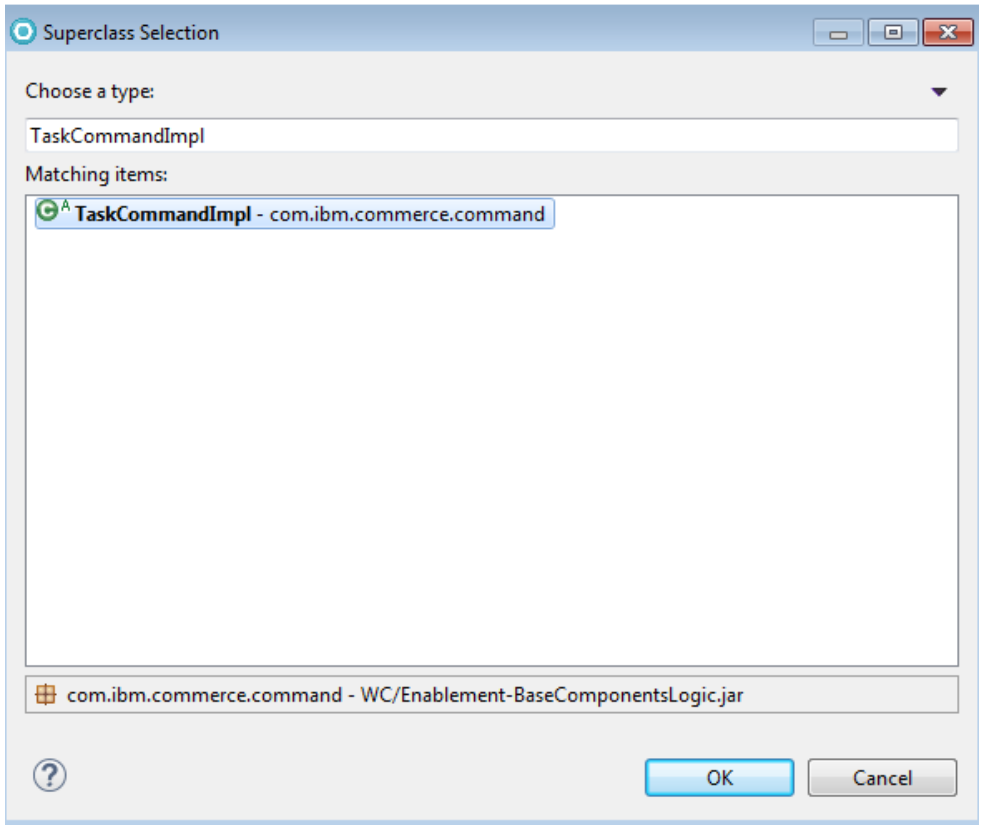
**Note:** Completed copies of the files that were being created are available in: **C:\exercises\customize-business-logic**. The files can either be copied to the workspace or opened in a text editor to review the contents. If you copy the files into the workspace using Windows File Explorer, make sure you refresh the **WebSphereCommerceServerExtensionsLogic** project in RAD, so that the changes are detected and compiled.

---

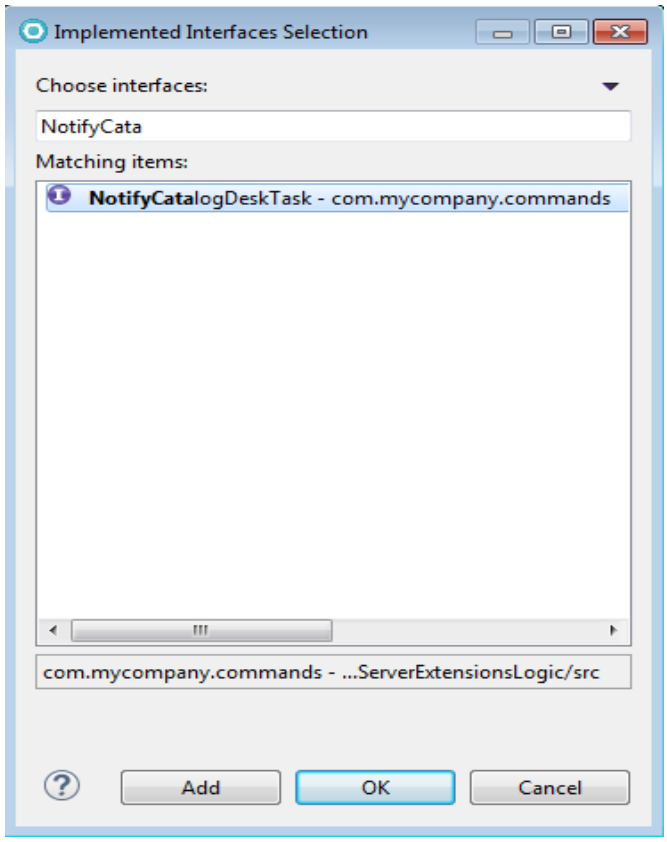
## Part 2: Create new task command implementations

In the previous section, you created two new interfaces for task commands. In this section, you build the implementation classes for those interfaces.

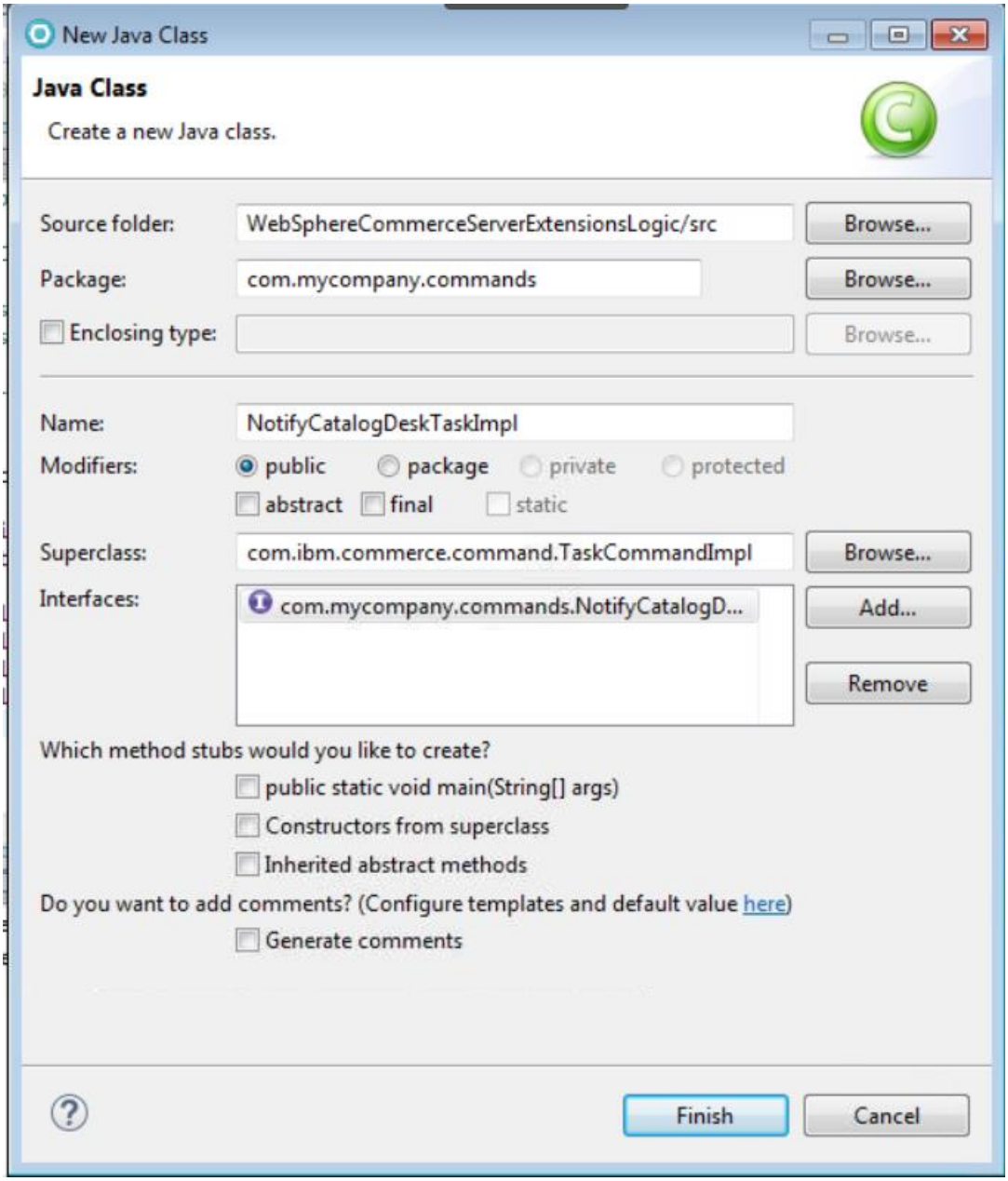
1. In the Enterprise Explorer view, right-click the **com.mycompany.commands** package and click **New > Class**.
2. Set the name of the class to **NotifyCatalogDeskTaskImpl**.
3. Click **Browse** next to the **Superclass** field to locate the class that is named **TaskCommandImpl**. Select it and click **OK**.




4. Click **Add** next to the **Interfaces** field to locate the corresponding interface that you created in the previous section.




5. Click **OK**.
6. Clear the **Inherited abstract methods** check box is cleared. The resulting **New Java Class** dialog resembles the following image:




7. Click **Finish**.
8. Repeat the preceding steps to create another Task command implementation that is named **AddUserToCatalogMailingListTaskImpl** that implements its related interface and extends from **TaskCommandImpl**.

---

**Troubleshooting:** Each of these new classes show errors because they do not yet implement the appropriate methods from the interfaces. The errors are corrected in the following steps.

---

9. Add fields, with getters and setters, to the **NotifyCatalogDeskTaskImpl** for the Customer's Name, Mailing Address, and the Catalog ID.

   a) Open the **NotifyCatalogDeskTaskImpl** class if not already open.

      - Add the following code inside the class definition:

```
private String customerName;
private String mailingAddress;
private String catalogId;
```

   b) Right-click the code, and click **Source > Generate Getters and Setters**.

   c) Click **Select All** and **OK**.

   d) Save the class but keep the editor open. Notice that the error is removed from this class.

10. Add fields, with getters and setters, to the **AddUserToCatalogMailingListTaskImpl** for the customer's ID and a boolean to represent whether the customer is a member of the mailing list. By default, a customer is not part of the mailing list.

   a) Open the AddUserToCatalogMailingListTaskImpl class if not already open.

      - Add the following code inside the class definition:

```
private String customerId;
private Boolean isMailingListMember = new Boolean(false);
```

   b) Right-click the code and click **Source > Generate Getters and Setters**.

   c) Click **Select All** and **OK**. The other error disappears.

11. In each of these task command implementations, add sample code for the respective **performExecute()** methods.

   a) In each task command implementation, right-click, and click **Source > Override/Implement Methods**.

      - Select **performExecute()** from the list, and click **OK**. An override method stub is added to the class definitions.

   b) Add the following code to **NotifyCatalogDeskTaskImpl.performExecute()**:

```
public void performExecute() throws ECException {

   super.performExecute();
```

```
        setCustomerName("Dr. John Smith");
        setMailingAddress("123 Evergreen St., Wheeling, WV");
        setCatalogId("Spring 2010");
    }
```

c) Right click the code, select **Source > Organize Imports**.

d) Add the following code to **AddUserToCatalogMailingListTaskImpl.performExecute()**:

```
public void performExecute() throws ECException {
    super.performExecute();
    setCustomerId("112233");
    setIsMailingListMember(Boolean.valueOf(true));
}
```

12. Save the two new interfaces and two new implementation classes.

---

**Note:** Obviously, the classes are not functional or practical classes. The methods in these Task commands are further modified, but for now, they serve as a demonstration of the relationship between controller and task commands, and how those changes are captured at the storefront.

---

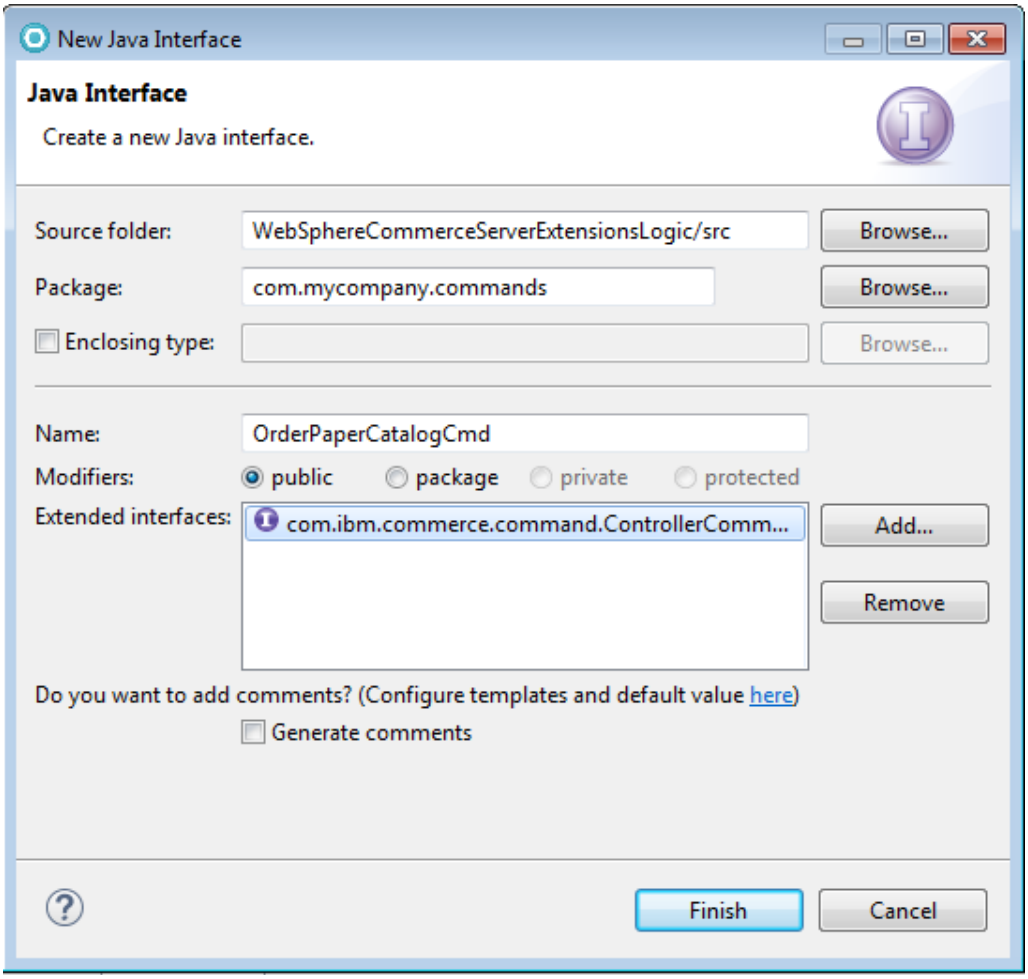
**Note:** Completed copies of the files that were being created are available in: **C:\exercises\customize-business-logic**. The files can either be copied to the workspace or opened in a text editor to review the contents. If you copy the files into the workspace using Windows File Explorer, make sure you refresh the **WebSphereCommerceServerExtensionsLogic** project in RAD, so that the changes are detected and compiled.

---

## Part 3: Create a Controller Command Interface

Create an interface that is named **OrderPaperCatalogCmd** that extends **ControllerCommand**.

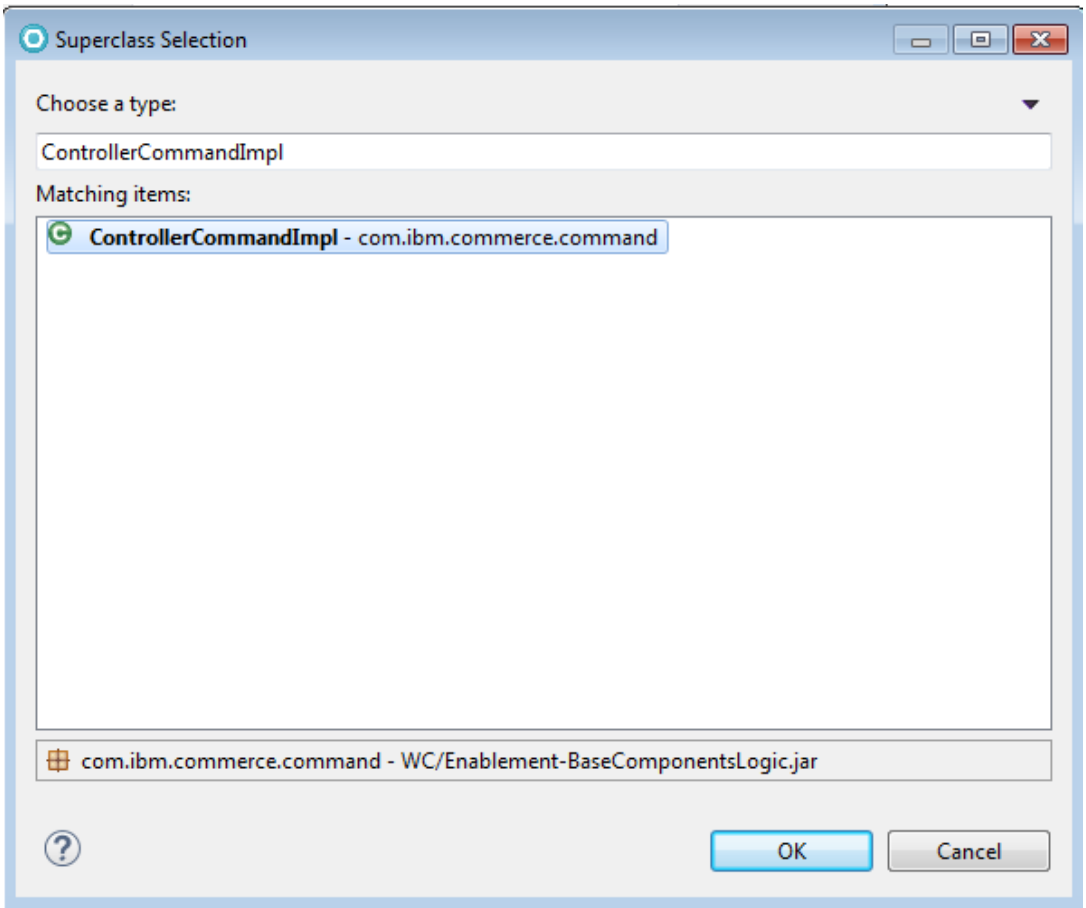
1. Right-click the **com.mycompany.commands** package and click **New > Interface**. In the resulting dialog box, enter **OrderPaperCatalogCmd** for the interface name.
2. Click **Add** to add the interface that this command extends which is **ControllerCommand**.
3. In the resulting dialog box, begin to type the name **ControllerCommand** in the **Extended Interfaces Selection** field. Notice that the list of matching types gets more specific as you type. When you can see the **ControllerCommand** interface in the list, select it and click **OK**.
4. Verify that your **New Java Interface** dialog has the following settings, and click **Finish**.



## Part 4: Create a Controller Command Implementation Class

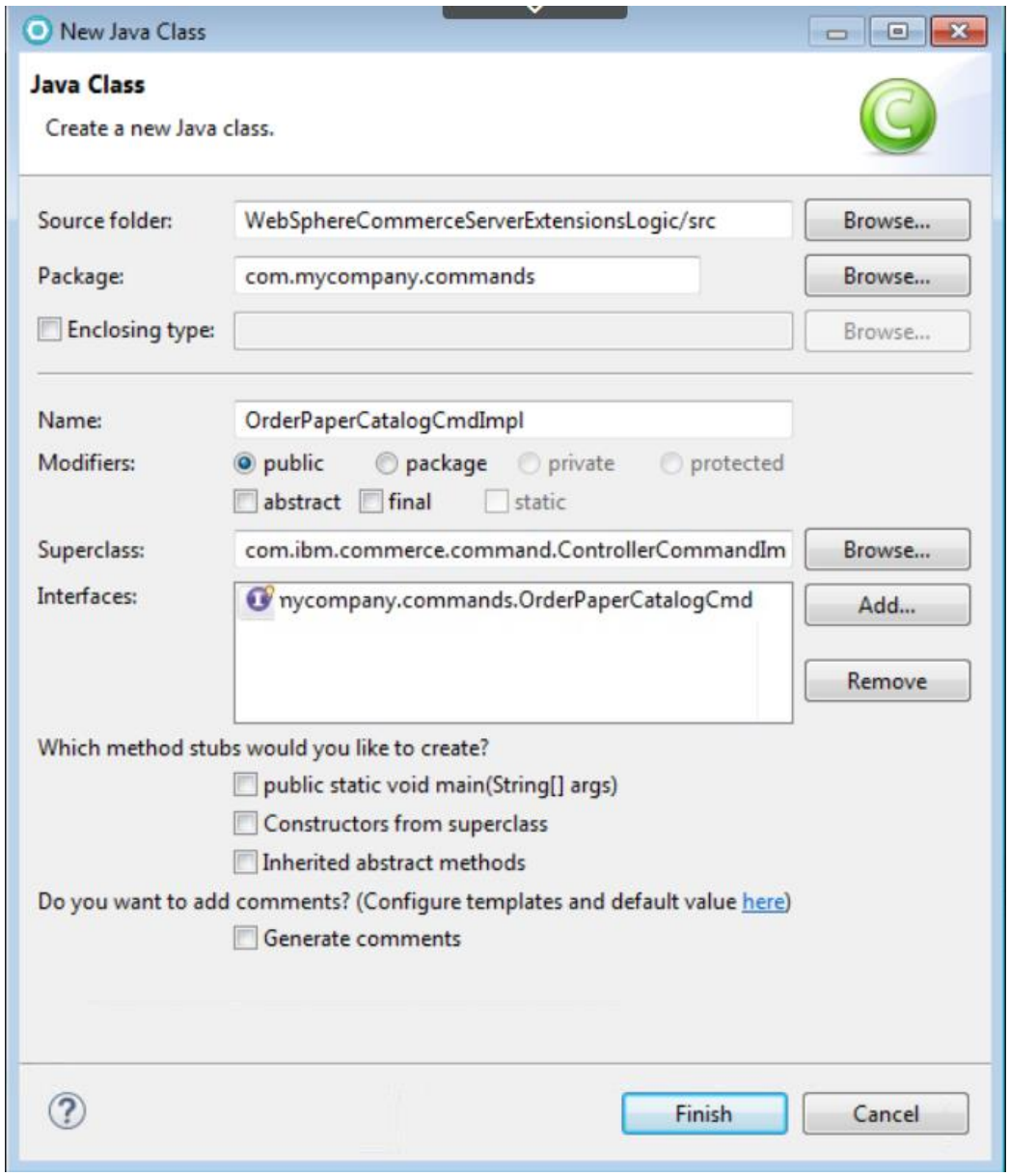
Create an implementation class, **OrderPaperCatalogCmdImpl**, to implement the new **OrderPaperCatalogCmd** interface.

1. Right-click the **com.mycompany.commands** package and click **New > Class**. In the resulting dialog box, enter a name of **OrderPaperCatalogCmdImpl**.
2. Click **Browse** beside the Superclass text field.
3. In the resulting dialog box, begin to type **ControllerCommandImpl** in the Superclass **Selection** field. As you type, the list of matches changes. When you see the **ControllerCommandImpl** type in the list, select it, and click **OK**.




4. Click **Add** beside the **Interfaces** field
5. In the resulting dialog box, begin to type **OrderPaperCatalogCmd** in the Implemented **Interfaces Selection** field. As the list of matching interfaces becomes more specific, select the **OrderPaperCatalogCmd** interface, and click **OK**.

6. Verify that your new class has the following parameters, and then click **Finish**. The new class opens in an editor.




7. Verify that the new class is similar to the screen capture here.

```
1  package com.mycompany.commands;
2
3  import com.ibm.commerce.command.ControllerCommandImpl;
4
5  public class OrderPaperCatalogCmdImpl extends ControllerCommandImpl implements
6          OrderPaperCatalogCmd {
7
8  }
9
```

8. Save the file, but keep the editor open. You continue to edit this class.
9. Right-click this implementation class and click **Source > Override/Implement Methods**.
10. Select **AbstractECTargetableCommand.performExecute()** and click **OK**.
11. Add the following code to the body of the **OrderPaperCatalogCmdImpl** class. This code overrides the **performExecute** method of the superclass.

```
public void performExecute() throws ECException {
// rspProp will hold the responses from the tasks
TypedProperty rspProp = new TypedProperty();
NotifyCatalogDeskTask notifyDeskCmd = null;
AddUserToCatalogMailingListTask addToMailingListCmd = null;
String mailingListMessage = "Please join our mailing list";

super.performExecute();

try {
// STEP ONE: Run the NotifyDesk task command
notifyDeskCmd = (NotifyCatalogDeskTask) CommandFactory.createCommand
("com.mycompany.commands.NotifyCatalogDeskTask", getStoreId());

// this is required for all commands
notifyDeskCmd.setCommandContext(getCommandContext());

// invoke the task command's performExecute method
notifyDeskCmd.execute();

// retrieve output parameters from the task command,
// then add them to response properties
rspProp.put("taskShippingMessage", "The " +
notifyDeskCmd.getCatalogId() + " catalog will be shipped");
```

```
rspProp.put("taskShippingAddressee", "To: " +
notifyDeskCmd.getCustomerName() + " at: " +
notifyDeskCmd.getMailingAddress());

} catch (ECException ex) {
throw (ECException) ex;
}
try {
// STEP TWO: Run the AddToMailingList task command
addToMailingListCmd = (AddUserToCatalogMailingListTask)
CommandFactory.createCommand
("com.mycompany.commands.AddUserToCatalogMailingListTask",
getStoreId());
addToMailingListCmd.setCommandContext(getCommandContext());
addToMailingListCmd.execute();

if(addToMailingListCmd.getIsMailingListMember()) {
mailingListMessage = "Welcome to our mailing list!";
}
rspProp.put("taskAddToMailingList", mailingListMessage);
} catch (ECException ex) {
throw (ECException) ex;
}
rspProp.put(ECConstants.EC_VIEWTASKNAME, "OrderPaperCatalogView");
setResponseProperties(rspProp);
}
```

**Note:** The code for this class is supplied in a text file that is in **C:\exercises\customize-business-logic\OrderPaperCatalogCmdImpl.java** if you would prefer to cut and paste. If you copy the files into the workspace using Windows File Explorer, make sure you refresh the **WebSphereCommerceServerExtensionsLogic** project in RAD, so that the changes are detected and compiled.

**Note:** In a more robust version of the task commands, it would be possible to send an email to the Catalog Publications desk with the customer's order, and have the customer select which catalog they would prefer to receive. It would also be possible to retrieve the customer's information from their profile and pass that to the responsible parties and to the view. Because of time constraints, and the complexity that is involved in such tasks, they are not implemented now.

**Note:** In the **performExecute()** method of the controller command implementation, did you call the task commands by their interface or implementation classes?

**A:** The tasks are called and cast to their interfaces. The methods that are called are the interface method stubs created in a previous step.

What is the benefit of calling them in this way?

**A:** By calling them by their interfaces, the developer takes advantage of the power of redirection: any implementation class might be called if it uses that particular interface. This option allows for other classes to replace the default implementation with minimal code change. This is a typical Java strategy.

How are the implementations invoked and executed?

**A:** In each of the interfaces, a **defaultCommandClassName** was listed. If the implementation classes are ever replaced, this default class name needs to be changed. If implementation classes change regularly, it is a better strategy to log the class name in the CMDREG instead of hardcoding it in Java.

12. Fix your imports, if necessary. You can type them, use **Organize Imports** from the right-click menu, or copy them from the supplied source file, **C:\exercises\customize-business-logic\OrderPaperCatalogCmdImpl.java**. There should not be any errors present.

    Once you resolve your imports, your imports resemble the following list:

    ```
    import com.ibm.commerce.command.CommandFactory;

    import com.ibm.commerce.command.ControllerCommandImpl;

    import com.ibm.commerce.datatype.TypedProperty;

    import com.ibm.commerce.exception.ECException;

    import com.ibm.commerce.server.ECConstants;
    ```

    The **performExecute** method accomplishes three main tasks. The first two tasks are to call and execute the task commands that make up the body of the controller. In a real sense, the work of the controller command is being delegated to the two tasks.

    Finally, once the tasks are properly executed it is a requirement of the WebSphere Commerce Programming Model that all Controller commands return a view. In the final section, it specifies that the view to be returned is the Order Catalog view that you previously created. Additionally, it sets the command's response properties to be the new **rspProp** object.

13. Save your work and close the editor.

## Part 5: Register the new Controller Command

1. Restart the WebSphere Commerce Test Server.

---

**Note:** You can follow the progress of the test environment by viewing the **startServer.log** file that is in the **<WCDE_HOMEDIR>\wasprofile\logs\server1** folder.

---

2. After the server is started, open a web browser to:

   http://localhost/webapp/wcs/admin/servlet/db.jsp

3. Update the **CMDREG** table to include the entry for the **OrderPaperCatalogCmd**; enter the following query, and click **Submit Query**.

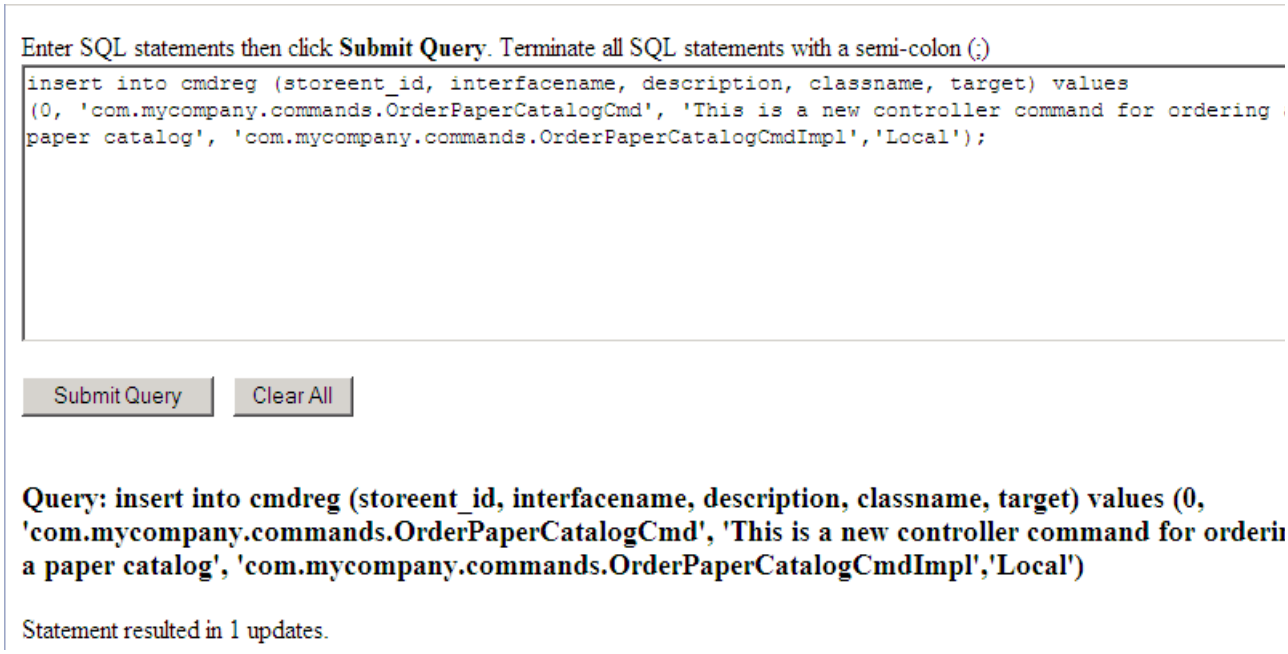
```
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION, CLASSNAME,
TARGET) values (0,'com.mycompany.commands.OrderPaperCatalogCmd', 'This
is a new controller command for ordering a paper catalog',
'com.mycompany.commands.OrderPaperCatalogCmdImpl','Local');
```

   Submitting the query results in one update.

---

**Note:** You can also copy the SQL from: **C:\exercises\customize-business-logic\SQLSnippet.txt**

---



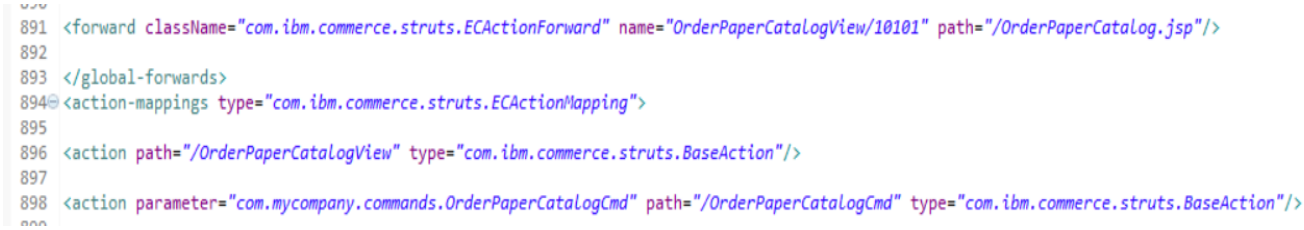
4. In WebSphere Commerce Developer, find the **struts-config-ext.xml** file in the Enterprise Explorer view. Navigate to **Stores > WebContent > WEB INF > struts-config-ext.xml**, right-click it, and then click **Open With > XML Editor**.

5. Add an action mapping for the new **OrderPaperCatalogCmd**.

   a) In the **<action-mappings>** section, add an entry for **OrderPaperCatalogCmd**:

```
<action parameter="com.mycompany.commands.OrderPaperCatalogCmd"
      path="/OrderPaperCatalogCmd"
      type="com.ibm.commerce.struts.BaseAction"/>
```

   The section should resemble the following screenshot:




   b) Press **Ctrl+S** to save and close the editor.

6. Refresh the **Struts Configuration** and **Commerce Commands** registries using WebSphere Commerce Administration Console.

## Part 6: Load the Access Control Policy

In the section, you need to create an access control policy for the new controller command you created in the previous section. A task command does not need a separate access control policy as it is protected by the access control policy of the controller command that calls it.

1. If it is started, stop the test environment by right-clicking WebSphere Commerce Test Server in the Server view of the Java EE perspective, and click **Stop**.

   Note: If Apache Derby is used as the development database, then it is required to stop the WC server as only one connection can be made to the database at one time. If we don't stop the server, then acpload will be unable to connect to the database. If you use DB2, then there is no need to stop the server before running acpload.

2. Copy the **OrderPaperCatalogCmdACPolicy.xml** file found in the **C:\exercises\customize-business-logic** folder to the **C:\IBM\WCDE80GM\xml\policies\xml** folder.

**Note:** All access policy files must reside in C:\IBM\WCDE80GM\xml\policies\xml directory in order for the **acpload** command to access them.

3. Start a command prompt and navigate to the following directory: **C:\IBM\WCDE80GM\bin**.
4. From this directory, run the **acpload** command with the access control policy XML file.

```
acpload OrderPaperCatalogCmdACPolicy.xml
```

**Note**: The **acpload** utility is used to load access control policies, which are written in an XML format, into the WebSphere Commerce database. The load utility takes the following format:

```
acpload db_name db_user db_password xmlFile db_schema
```

If you use Derby as the development database, then the database details do not need to be provided to the utility. But if you use DB2, then the database details need to be provided as shown in the above syntax.

5. Navigate to the **C:\IBM\WCDE80GM\logs** directory. Inspect the **acpload.log** and **messages.txt** files to ensure that the access control policy loaded successfully. The **messages.txt** file might not exist if the load completed successfully. Also, check to see whether policy files were created successfully in the **C:\IBM\WCDE80GM\bin** directory:

   **OrderPaperCatalogCmdACPolicy_idres.xml**

and

**OrderPaperCatalogCmdACPolicy_xmltrans.xml**.

These two files are created as part of a successful **idresgen** utility process. Check the other error files in this directory if there are any issues.

## Part 7: Modify the JSP to process the Controller command

The new business logic is built. When the controller command is executed, it instantiates and executes its two task commands, processes the responses, and returns them, along with the view name, back to the presentation layer.

In order to execute the business logic, you must modify the existing JSP so that it can call the controller command. To accomplish this task, you need to add a button to the JSP. When clicked, this button runs a JavaScript function that you build. The function submits a dummy form to execute the underlying controller command. Finally, you must present the results of the command execution to the JSP page.

### Create a New JavaScript Function

1. In the Enterprise Explorer view, navigate to **Stores > WebContent > AuroraStorefrontAssetStore**.
2. Open **OrderPaperCatalog.jsp** in the editor.
3. Inside the body of the JSP (hint: look for the **<body>** tag), add the following script:

```
<script language="Javascript1.2"> function placeCatalogOrder(form)

{

        form.submit();

}

</script>
```

When called, this function uses POST to submit a form (which you create) to the server.

4. Save the file.

### Add a "Dummy" Form

1. Beneath the script, add the following form:

```
<form method="post" action="OrderPaperCatalogCmd"
name="OrderPaperCatalog_Form" id="OrderPaperCatalog_Form">

      <input type="hidden" name="storeId" value="<c:out
value="${WCParam.storeId}" />"
id="OrderPaperCatalogForm_FormInput_storeId"/>
```

```
      <input type="hidden" name="langId" value="<c:out
value="${WCParam.langId}" />"
id="OrderPaperCatalogForm_FormInput_langId"/>

      <input type="hidden" name="catalogId" value="<c:out
value="${WCParam.catalogId}" />"
id="OrderPaperCatalogForm_FormInput_catalogId"/>

</form>
```

This form indicates the controller command to execute (an action named **OrderPaperCatalogCmd**) when the form is posted. The action name should match the value given for the controller command **OrderPaperCatalogCmd** in **struts-config-ext.xml** in a previous step. The input fields ensure that the correct store, catalog, and language ID are passed to the command.

---

**Note:** The code is available for you to copy and paste in the following file: **C:\exercises\customize-business-logic\JSPSnippet1.txt**. If you copy the code into the JSP using Windows File Explorer, make sure you refresh the **Stores** project in RAD, so that the changes are detected and built.

---

```
<%@ include file="include/LayoutContainerTop.jspf"%>

<script language="Javascript1.2">
    function placeCatalogOrder(form)
    {
        form.submit();
    }
</script>
<form method="post" action="OrderPaperCatalogCmd" name="OrderPaperCatalog_Form" id="Orc
    <input type="hidden" name="storeId" value="<c:out value="${WCParam.storeId}" />" id
    <input type="hidden" name="langId" value="<c:out value="${WCParam.langId}" />" id='
    <input type="hidden" name="catalogId" value="<c:out value="${WCParam.catalogId}" />
</form>
```

2. Save the file.

## Add a Button to the JSP

1. Under the OrderCatalog header and message, add the following code to represent the button:

```
<a href="javascript:placeCatalogOrder(document.OrderPaperCatalog_Form)"
id="OrderPaperCatalogForm_Button"  class="button_secondary" role="button">
<div class="left_border"></div>
```

```
<div class="button_text"><span><fmt:message key="PLACE_ORDER" bundle="${storeText}"
/></span></div>
<div class="right_border"></div></a>
```

The resulting code should look like this:

```
<h2><fmt:message key="OrderCatalogHeader" bundle="${storeText}" />
</h2>
<p><fmt:message key="OrderCatalogMessage" bundle="${storeText}" />
</p>

<a href="javascript:placeCatalogOrder(document.OrderPaperCatalog_Form)" id=
"OrderPaperCatalogForm_Button"  class="button_secondary" role="button">
<div class="left_border"></div>
<div class="button_text"><span><fmt:message key="PLACE_ORDER" bundle="${storeText}" />
</span></div>
<div class="right_border"></div>
```

---

Examine the preceding code. It presupposes that the following are true:

- There is a JavaScript function named **placeCatalogOrder**.
- There is a form that is named **OrderPaperCatalog_Form.**
- A placeholder that points to the **storetext_v2.properties** bundle determines the text on the button.

---

**Note:** The code is available for you to copy and paste from the following file: **C:\exercises\customize-business-logic\JSPSnippet2.txt**. If you copy the code into the JSP using Windows File Explorer, make sure you refresh the **Stores** project in RAD, so that the changes are detected and built.

---

2. Save the file.
3. Navigate to **Stores > Java Resources : src > AuoraStorefrontAssetStore** and open **storetext_v2.properties**.
4. Locate the line that you created in the previous exercise; ORDER_PAPER_CATALOG, and add a line below it:

   PLACE_ORDER = PLACE ORDER

5. Save and close the file.
6. In the same folder, open **storetext_v2_en_US.properties** and make the same modification.
7. Save and close the file.

## Add the Response Fields

1. In the **OrderPaperCatalog.jsp**, output the responses from the task commands (which is stored in the response of the controller command) to the view. Under the code that was added in a previous step to include a new button, include the following code:

```
<p><c:out value="${taskAddToMailingList}"/></p>

<c:out value="${taskShippingMessage}"/>

<c:out value="${taskShippingAddressee}"/>
```

---

**Note:** The code is available for you to copy and paste from the following file: **C:\exercises\customize-business-logic\JSPSnippet3.txt**. If you copy the code into the JSP using Windows File Explorer, make sure you refresh the **Stores** project in RAD, so that the changes are detected and built.

---

2. Save and close the JSP.

---

**Note:** The completed **OrderPaperCatalog.jsp** file can be found in: **C:\exercises\customize-business-logic.** If you copy the JSP using Windows File Explorer, make sure you refresh the **Stores** project in RAD, so that the changes are detected and built.

---

## Part 8: Test on the WebSphere Commerce Test Server

1. Start the test environment.
2. Ensure that all the updated files are deployed to the server. Right-click the WebSphere Commerce Test Server and click **Publish.**
3. Open a new browser window and enter the following URL:

   http://localhost/webapp/wcs/stores/servlet/auroraesite
4. Click **Order a Catalog** from the title bar. A view with your button is displayed.
5. Click **Place Order**. The same view is displayed, but with the responses from the controller command.


---

**Note:** What happened when you clicked the **Place Order** button?

- Clicking the button that is called **placeCatalogOrder** JavaScript function that took a dummy form as its argument.
- The dummy form contained the name of the controller command and the default store, catalog, and language IDs.
- When posted, the business logic facade called the controller command interface. This interface was cross-referenced in the command registry (CMDREG) to an implementation class: **OrderPaperCatalogCmdImpl**.
- The command implementation was instantiated and executed.
- In its execution, the command implementation called two task interfaces: **NotifyCatalogDeskTask** and **AddUserToCatalogMailingListTask**, the implementations for which were hardcoded in the interface definitions, not in the command registry, as the parent controller was.
- The task command implementations were instantiated and executed in turn. The Notify task sets dummy values for the customer and catalog. The mailing list task command sets a dummy value for the customer ID and sets the mailing list Boolean value to true.
- The response values from the tasks were returned to the controller command.
- The controller command sets the view name for return. It is the same view, but it is repopulated with response data.

- If the view is working properly, the true value from the mailing list task command issues the Welcome message and the dummy customer values from the Notify task is reflected on the JSP.

---

## What you did in this exercise

In this exercise, you created a command interface and implementation, performed the steps necessary to register it in the database, which is mapped to a Struts Action, and provided an access control policy.

## End of exercise

# Exercise 3. Creating an Enterprise JavaBeans in WebSphere Commerce

## Estimated time

02:30 hours

## What this exercise is about

This exercise covers the creation of a new Enterprise Java Beans and Access beans in WebSphere Commerce.

## What you should be able to do

After completing this exercise, you will be able to:

- Create a WebSphere Commerce table.
- Create an Entity bean.
- Configure EJB properties.
- Modify EJB Methods and Finders.
- Map the database table to the EJB.
- Generate the Access bean for the EJB.
- Test the Entity bean.

## Introduction

Your company is shipping paper catalogs to customers manually, and are now employing an automated system for customers to order and have specific catalogs that are shipped to them.

In previous exercises, you built the user interface and commands responsible, but the business logic in the controller command and associated tasks was dummy code. In this exercise, you build code to be able to access the WebSphere Commerce database schema.

The set of available paper catalogs are stored in a new database table, called XPAPERCATALOG. This table needs to be created. This exercise presumes that a new EJB is needed to maintain the records that are in the new XPAPERCATALOG table. In WebSphere Commerce, an access bean wraps an EJB to relieve some of the complexity of working with EJBs, and to provide an interface more like one would expect with Java classes.

## Requirements

- WebSphere Commerce Developer V8.0.0 Enterprise
- Extended sites store archive published
- AuroraESite store created
- Completion of the previous exercises, *Customizing Storefront Pages & Customize Business Logic*

## Exercise instructions

In the previous exercise, you created a controller command that executed two separate task commands. One of the task commands, **AddUserToCatalogMailingListTask**, was intended to set a field that would flag the customer as a member of the company catalog mailing list. The other task command, **NotifyCatalogDeskTask**, was intended to notify an external ordering desk to send a paper catalog to a specific customer.

Although in the previous exercise you used dummy data that is hardcoded into the implementation classes of the task commands, it is reasonable to assume that the data of the task commands is written to and retrieved from database tables.

This exercise covers customizing WebSphere Commerce persistence layer to write and retrieve values from a WebSphere Commerce database table. In this exercise, you create the database table, EJB, and Access bean that is required to work with the data. The data contained in the table will represent the set of available catalogs for mailing.

## Part 1: Create the XPAPERCATALOG table

1. Start WebSphere Commerce Test Server.

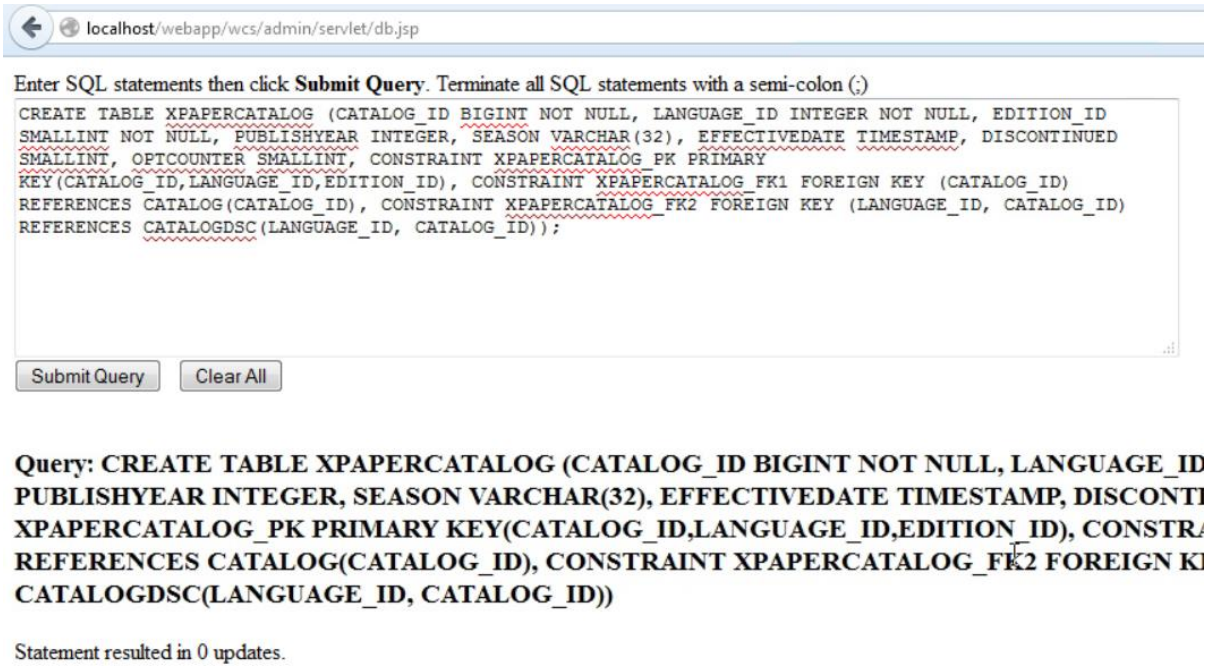
2. Open a web browser window and enter the following URL: http://localhost/webapp/wcs/admin/servlet/db.jsp

3. Enter the following SQL in the text area, to create the XPAPERCATALOG table.

```
CREATE TABLE XPAPERCATALOG (CATALOG_ID BIGINT NOT NULL,
LANGUAGE_ID INTEGER NOT NULL, EDITION_ID SMALLINT NOT NULL,
PUBLISHYEAR INTEGER, SEASON VARCHAR(32), EFFECTIVEDATE
TIMESTAMP, DISCONTINUED SMALLINT, OPTCOUNTER SMALLINT,
CONSTRAINT XPAPERCATALOG_PK PRIMARY
KEY(CATALOG_ID,LANGUAGE_ID,EDITION_ID), CONSTRAINT
XPAPERCATALOG_FK1 FOREIGN KEY (CATALOG_ID) REFERENCES
CATALOG(CATALOG_ID), CONSTRAINT XPAPERCATALOG_FK2 FOREIGN KEY
(LANGUAGE_ID, CATALOG_ID) REFERENCES CATALOGDSC(LANGUAGE_ID,
CATALOG_ID));
```

**Note:** The SQL is also available in **C:\exercises\create-ejb\createPaperCatalogTable.sql**.

Make sure the SQL is executed without any errors.

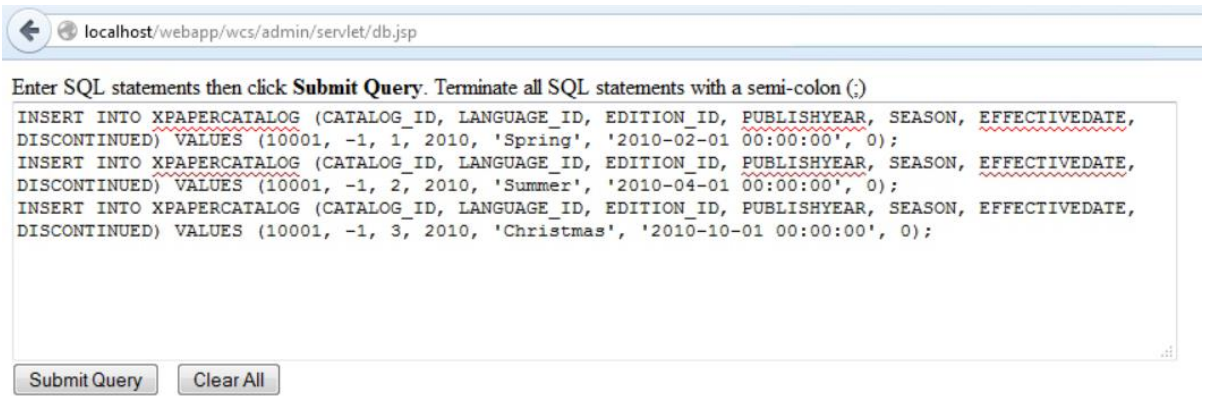



4. Enter the following SQLs to populate the new table with sample data.

```
INSERT INTO XPAPERCATALOG (CATALOG_ID, LANGUAGE_ID, EDITION_ID,
PUBLISHYEAR, SEASON, EFFECTIVEDATE, DISCONTINUED) VALUES (10001,
-1, 1, 2010, 'Spring', '2010-02-01 00:00:00', 0);

INSERT INTO XPAPERCATALOG (CATALOG_ID, LANGUAGE_ID, EDITION_ID,
PUBLISHYEAR, SEASON, EFFECTIVEDATE, DISCONTINUED) VALUES (10001,
-1, 2, 2010, 'Summer', '2010-04-01 00:00:00', 0);

INSERT INTO XPAPERCATALOG (CATALOG_ID, LANGUAGE_ID, EDITION_ID,
PUBLISHYEAR, SEASON, EFFECTIVEDATE, DISCONTINUED) VALUES (10001,
-1, 3, 2010, 'Christmas', '2010-10-01 00:00:00', 0);
```

**Note:** The SQLs are also available in **C:\exercises\create-ejb\populateTable.sql**.

Verify that the SQLs execute without errors.




**Note:** The **CATALOG_ID** (10001) used in the SQLs above is the id of the **Extended Sites Catalog Asset Store** master catalog. The value may be different in your environment. You can confirm the value by querying the **CATALOG** table.

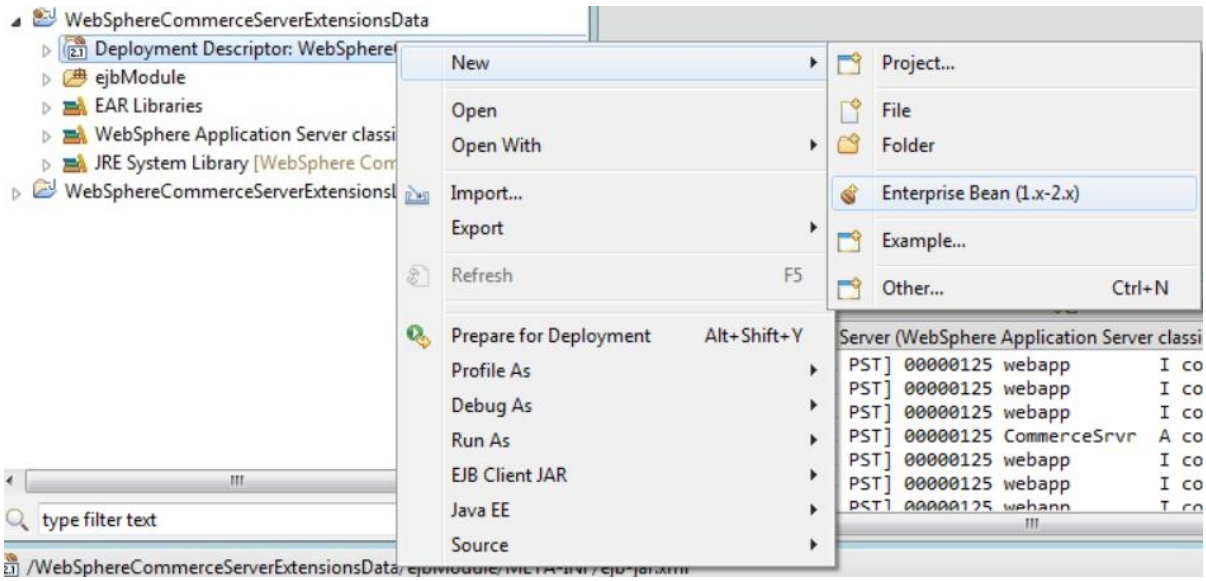
## Part 2: Create the XPaperCatalog Entity Bean

The following steps create the **XPaperCatalog** Entity bean.

1. In WebSphere Commerce Developer, open the Java EE perspective.

2. In the Enterprise Explorer view, navigate to **WebSphereCommerceServerExtensionsData > Deployment Descriptor**.

3. Right click **Deployment Descriptor**, and click **New > Enterprise Bean (1.x-2.x)**.




4. In the **Create an Enterprise Bean** dialog, enter the following information:

    a. Select the **Entity bean with container-managed persistence (CMP) fields** radio.

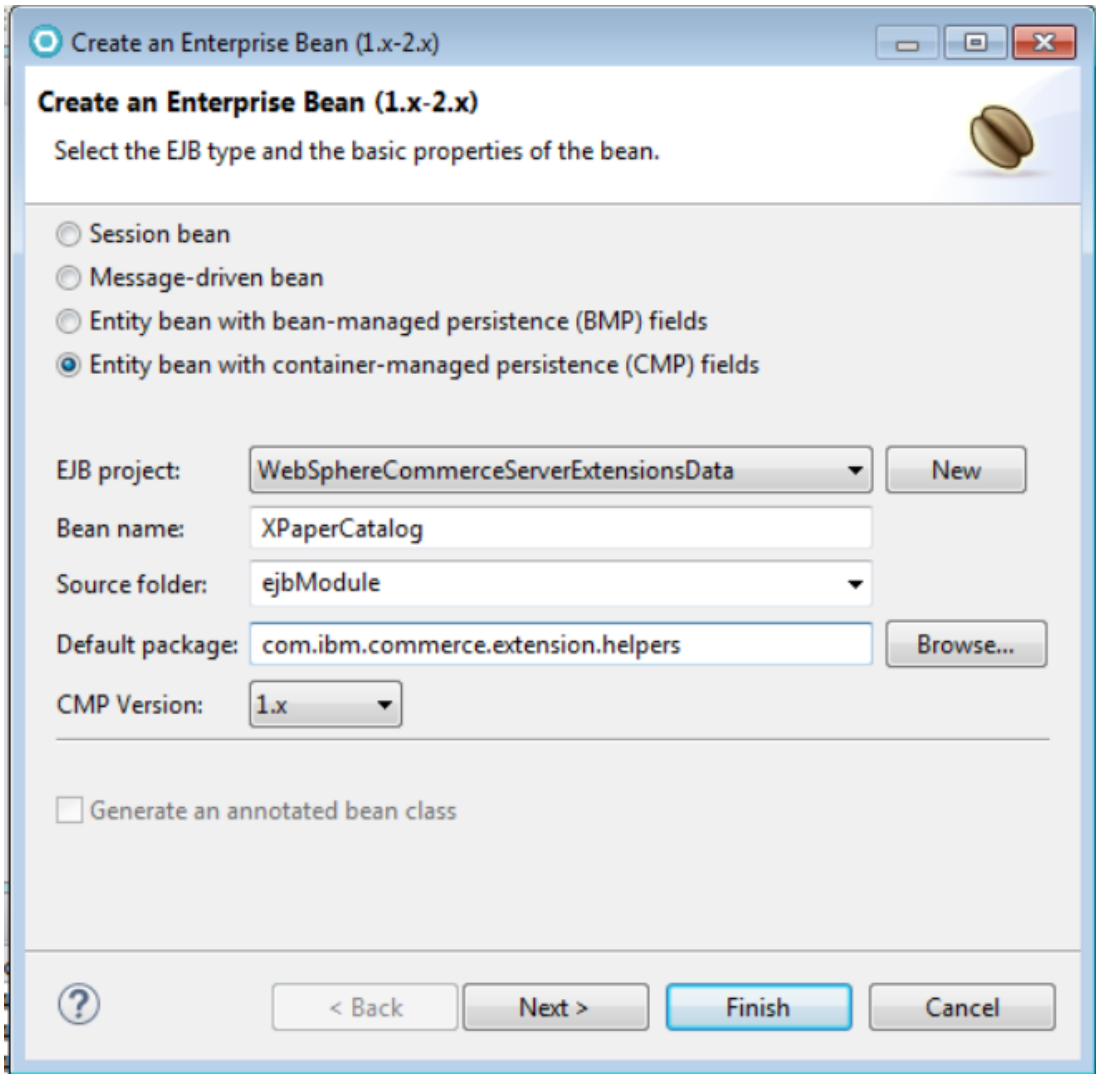
    b. Verify that the EJB project is **WebSphereCommerceServerExtensionsData**.

    c. Enter **XPaperCatalog** for bean name.

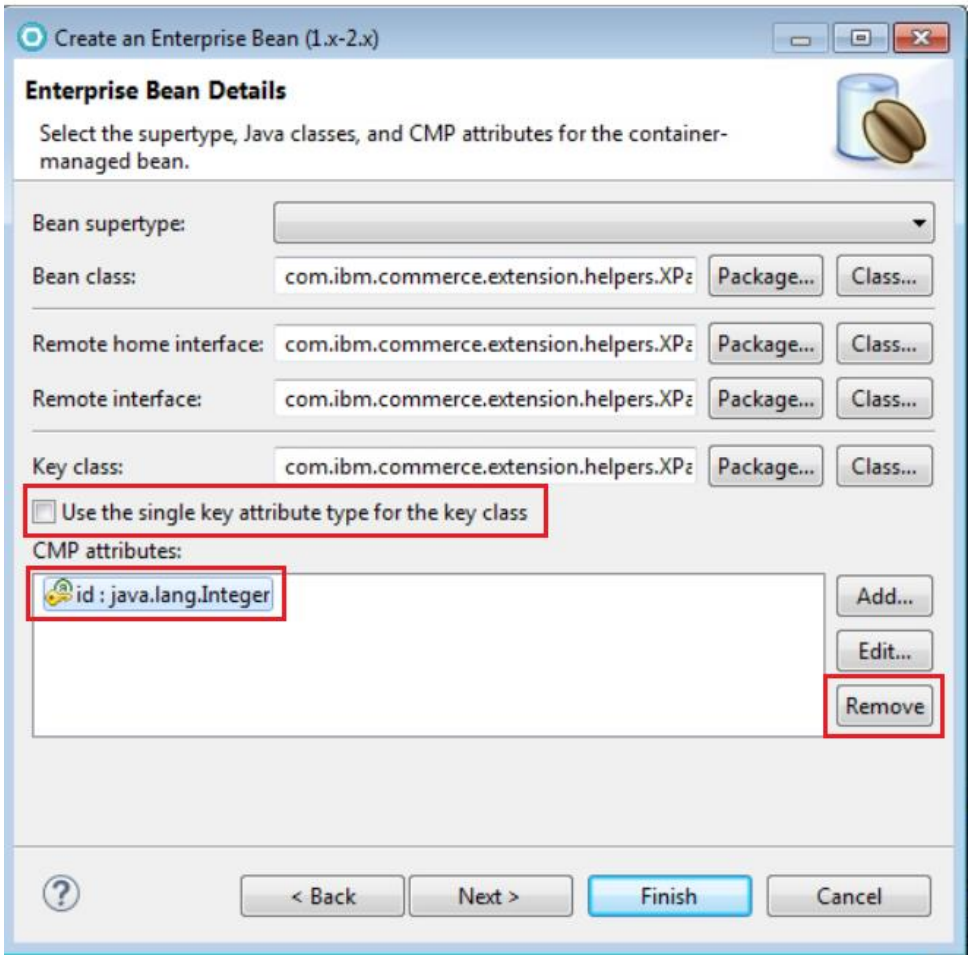
    d. Enter default package as **com.ibm.commerce.extension.helpers**.

    e. Select **1.x** for CMP version.

    f. Click **Next**.

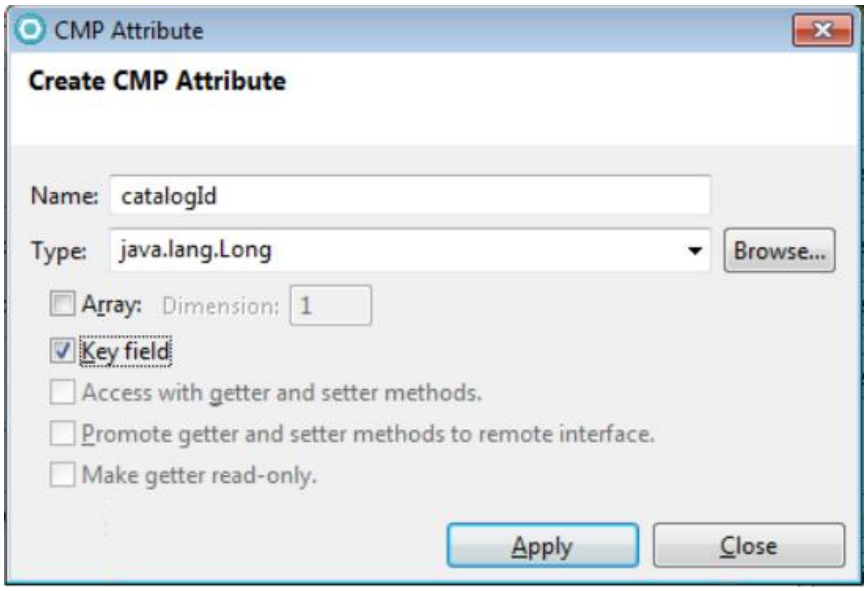



5. In the **Enterprise Bean Details** dialog, do the following:
    a. Clear the **Use the single key attribute type for the key class** check box.
    b. Remove the **id** attribute from the CMP attributes.

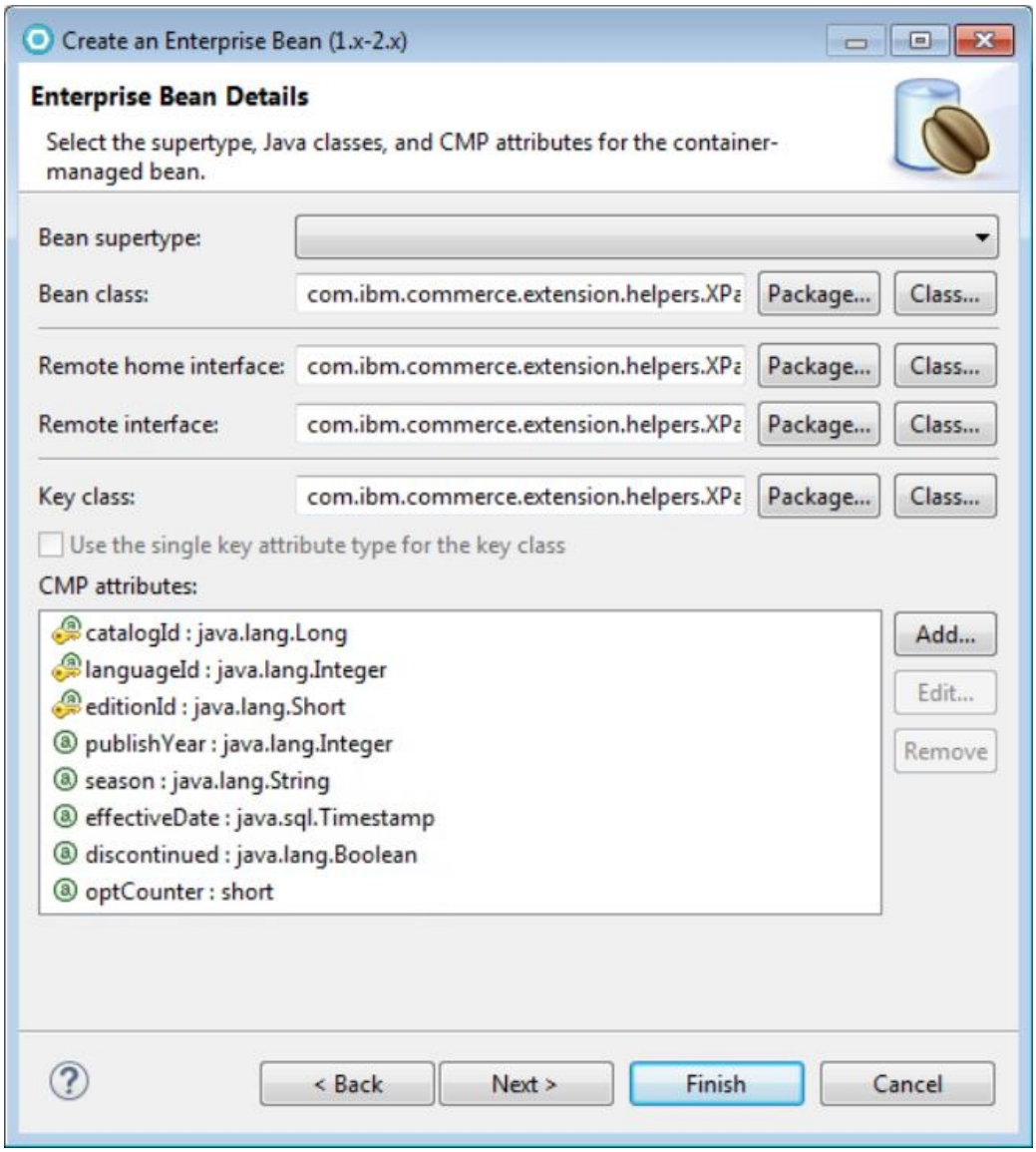



6. While still in the **Enterprise Bean Details** dialog, Click **Add** to add the following CMP attributes:

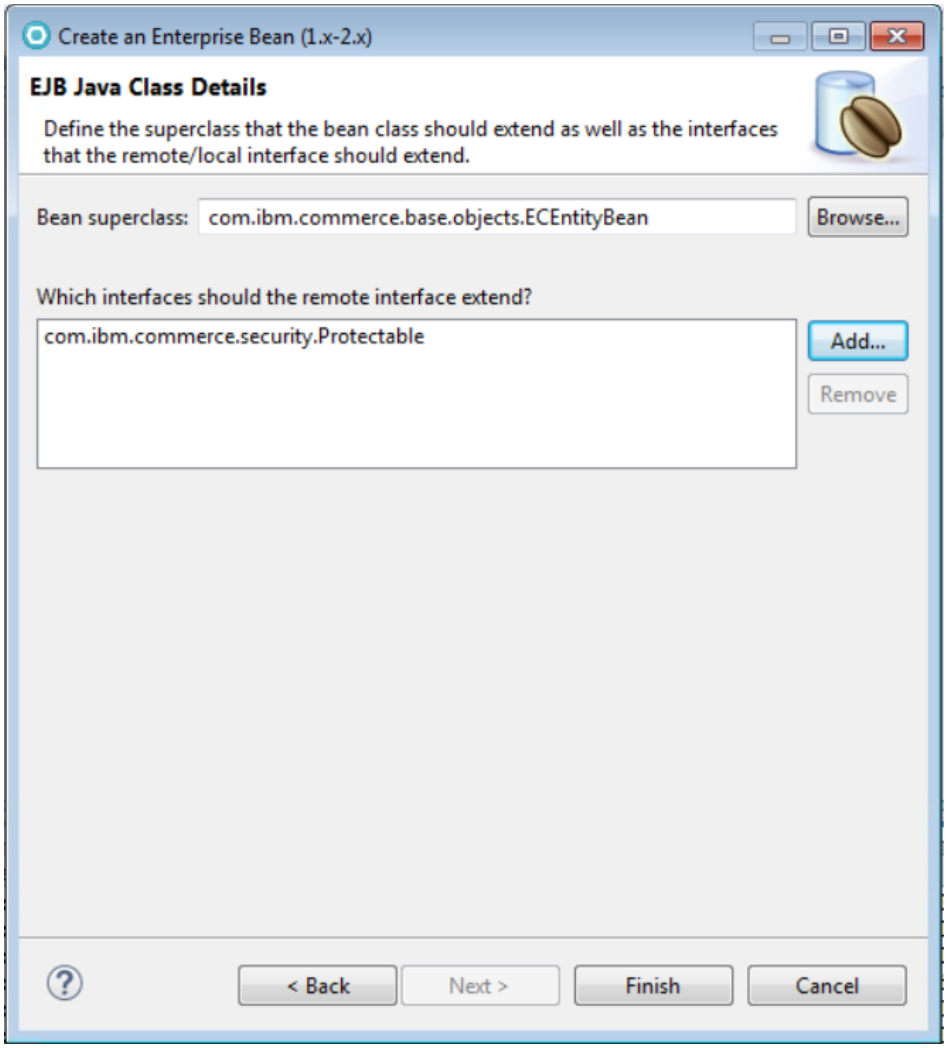
| Name | Type | Key field | Access with getter and setter | Promote getter and setter... |
|---|---|---|---|---|
| catalogId | java.lang.Long | true | n/a | n/a |
| languageId | java.lang.Integer | true | n/a | n/a |
| editionId | java.lang.Short | true | n/a | n/a |
| publishYear | java.lang.Integer | false | true | false |
| season | java.lang.String | false | true | false |
| effectiveDate | java.sql.Timestamp | false | true | false |
| discontinued | java.lang.Boolean | false | true | false |
| optCounter | short | false | false | n/a |



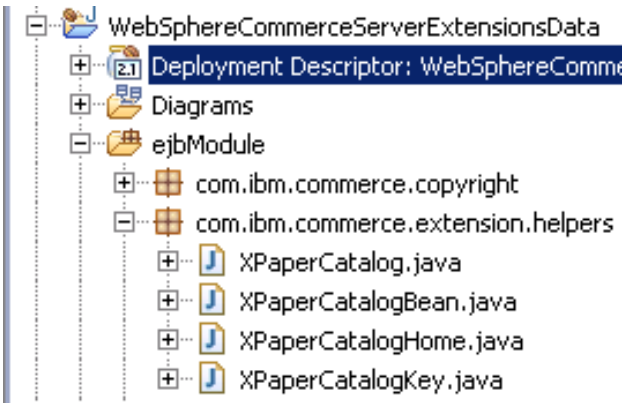

- After adding each attribute, click **Apply**. Once all the fields are added, click **Close** to close the dialog.
- Finally, the dialog will look like the screen capture shown below.
- Click **Next**.




7. In the **EJB Java Class Details** dialog:
    a. Enter or browse for **com.ibm.commerce.base.objects.**ECEntityBean for the bean super class.
    b. Browse for **com.ibm.commerce.security.Protectable** as the interface that the remote interface should extend. **Protectable** is needed to protect the EJB with access control.
    c. Click **Finish**.




The EJB classes are created and can be found under **ejbModule**.

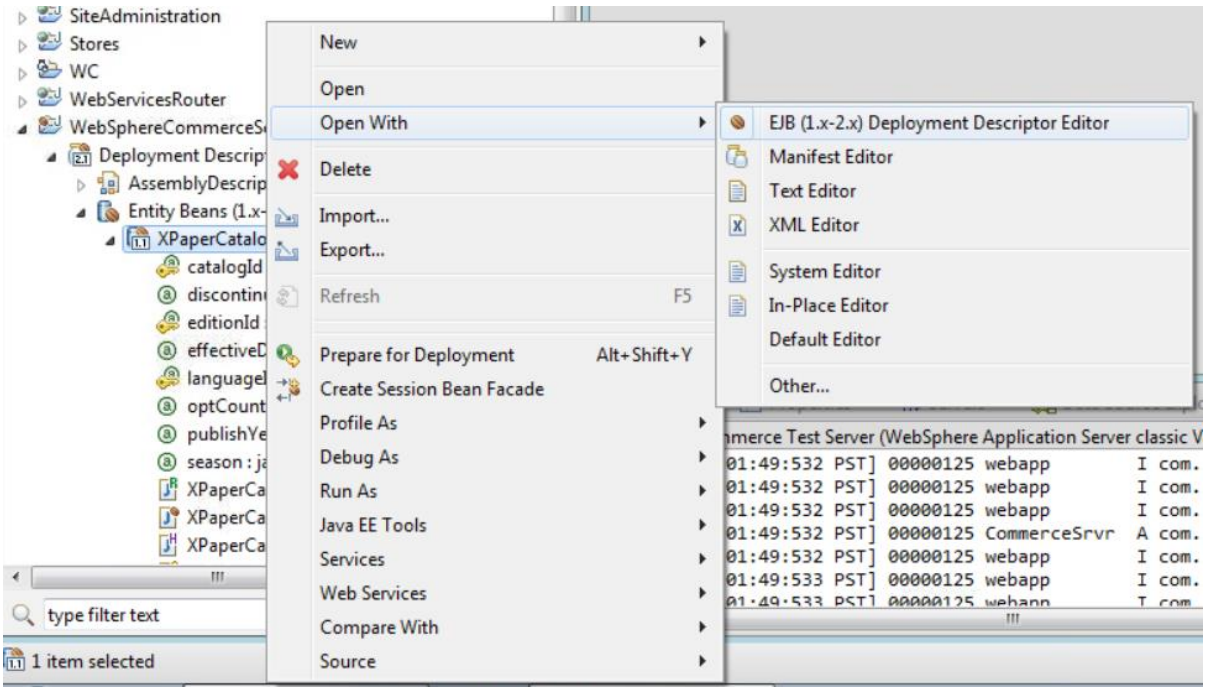


## Part 3: Configure the EJB properties

### Set the EJB Isolation Level

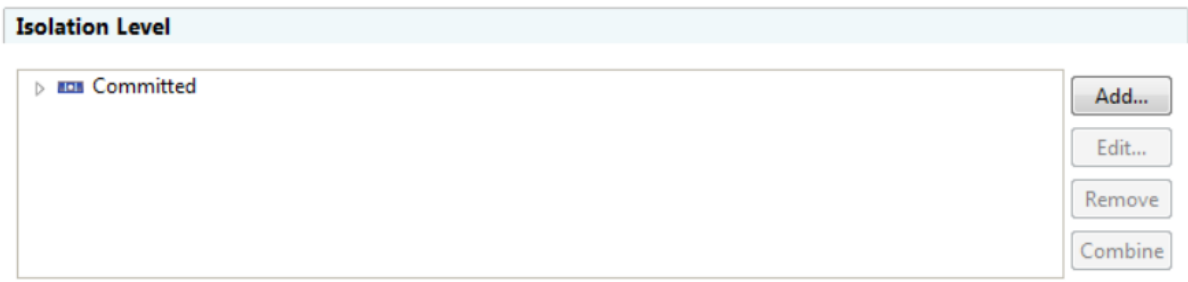
1. In the Java EE Enterprise Explorer view, expand **WebSphereCommerceServerExtensionsData > Deployment Descriptor > Beans (1.x-2.x)**.

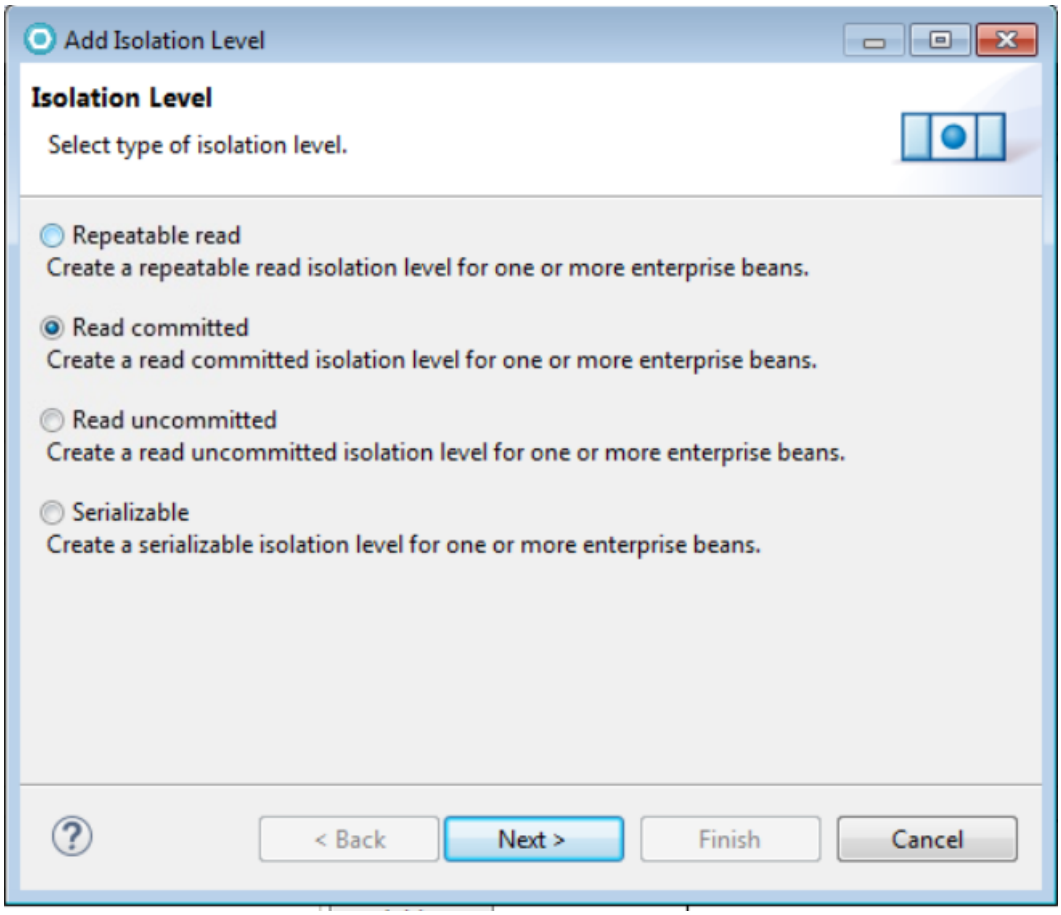
2. Right-click the **XPaperCatalog** bean and open it in the deployment descriptor.




3. Click the **Access** tab.

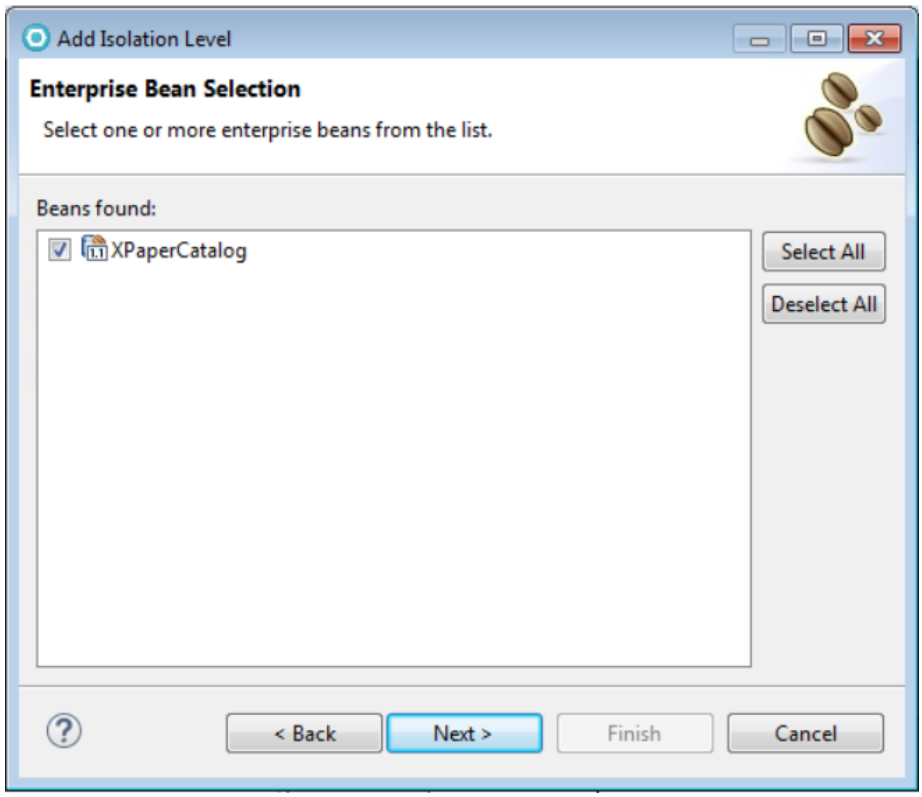
4. Scroll down and click **Add** in the **Isolation Level** section. The **Add Isolation Level** window is displayed.



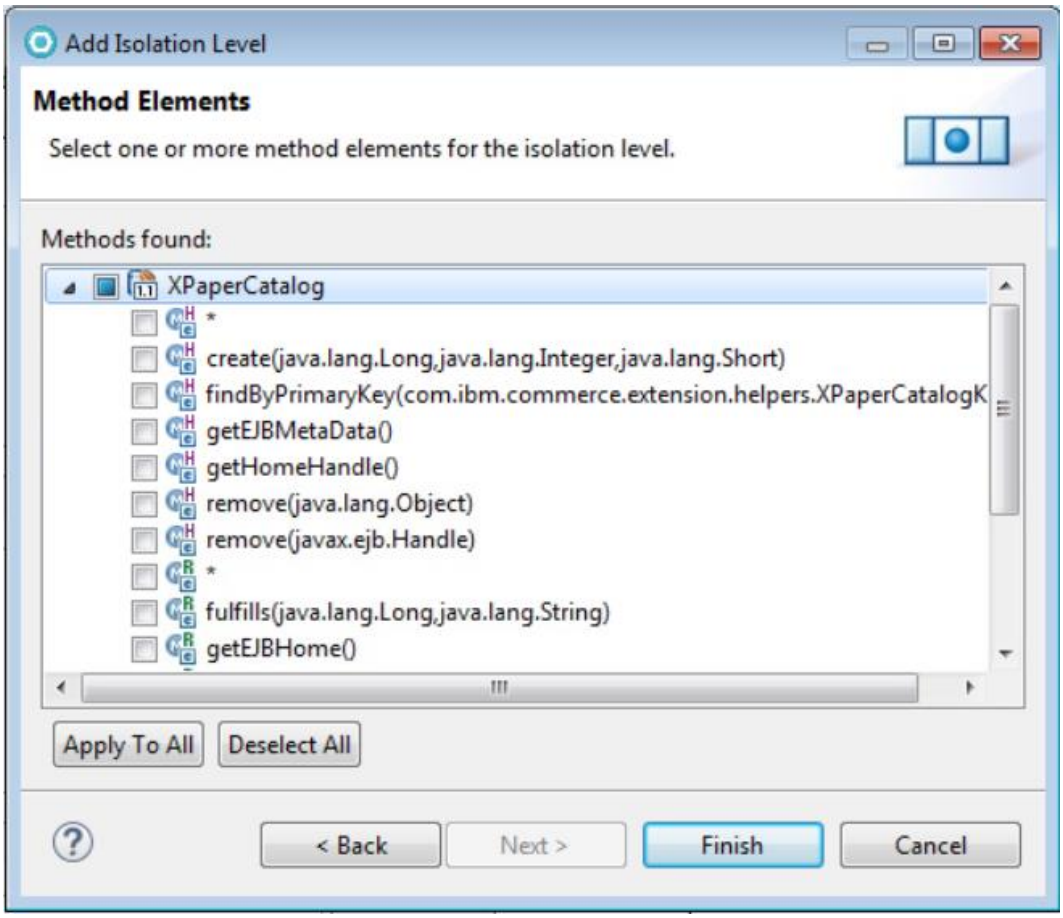



5. Select **Read committed**. Click **Next**.

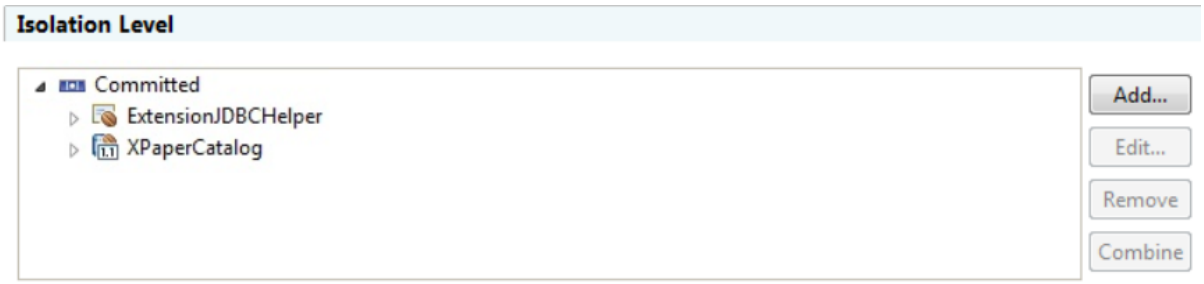
6. In the **Enterprise Bean Selection** dialog, select the **XPaperCatalog** check box. Click **Next**.




7. In the **Method Elements** dialog, select the **XPaperCatalog** check box. Click **Finish**.

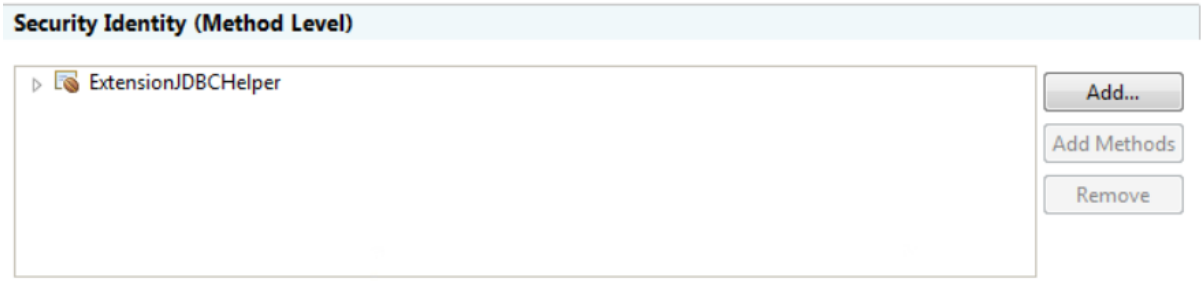



In the **Isolation Level** section, you will see XPaperCatalog with a Committed isolation level.

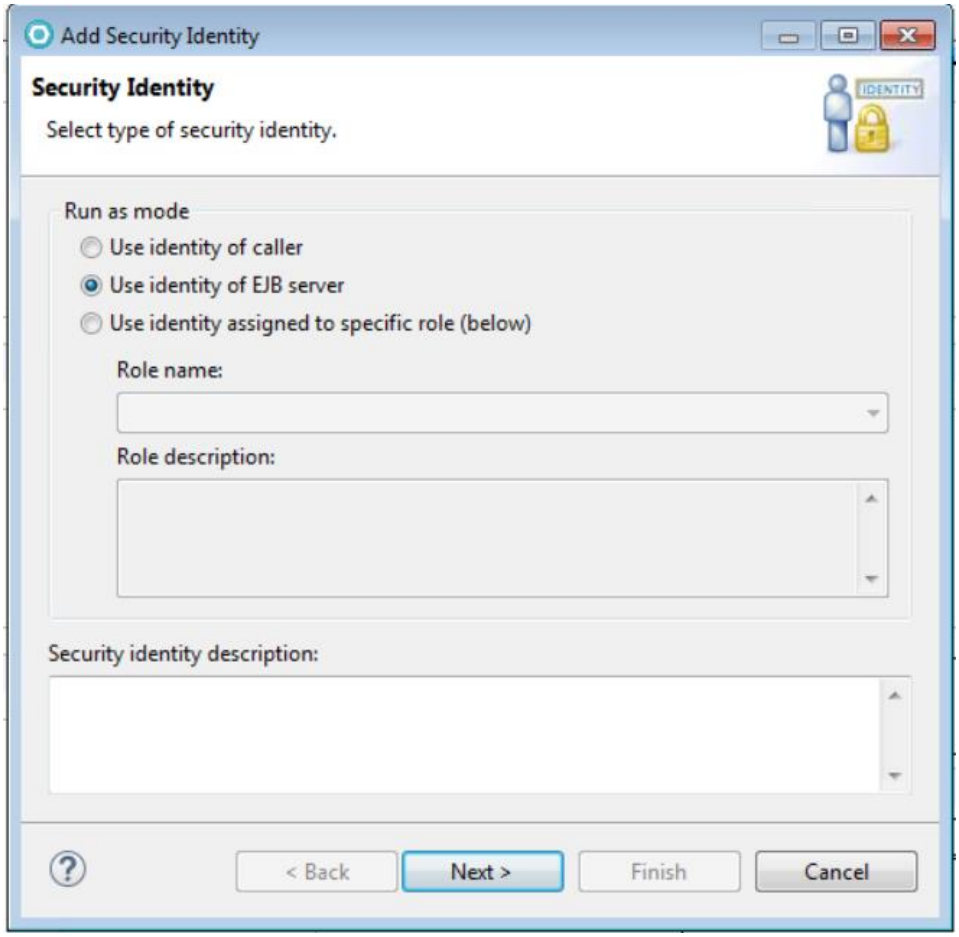



## Set the EJB Security Identity

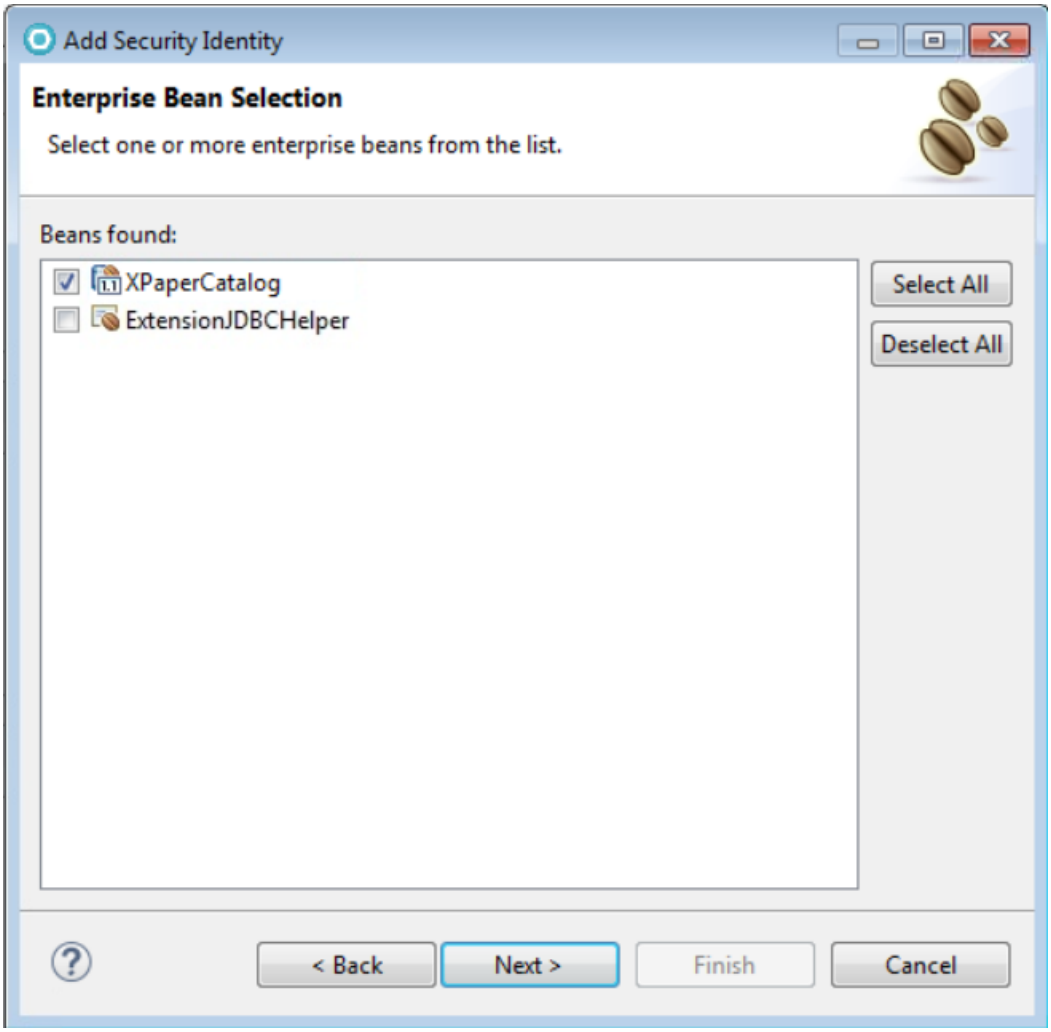
1. Staying on the **Access** tab, move to the **Security Identity (Method Level)** section.



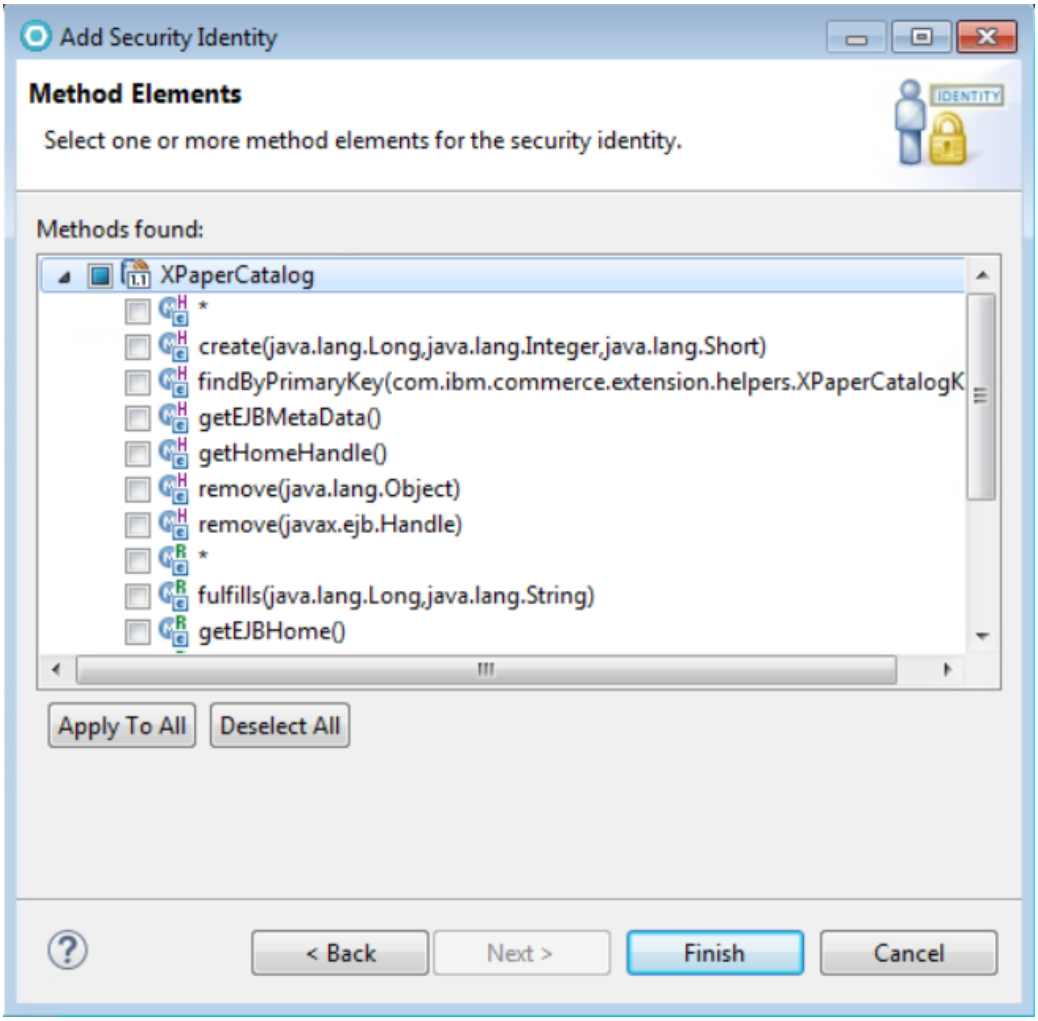

2. Click **Add**. In the **Add Security Identity** window, click **Use identity of EJB server**. Click **Next**.



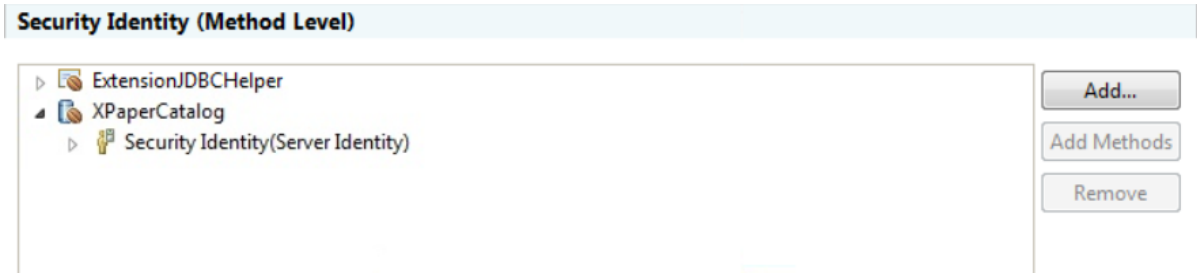

3. Select the **XPaperCatalog** check box. Click **Next**.




4. In the **Method Elements** dialog, select the **XPaperCatalog** check box. Click **Finish**.



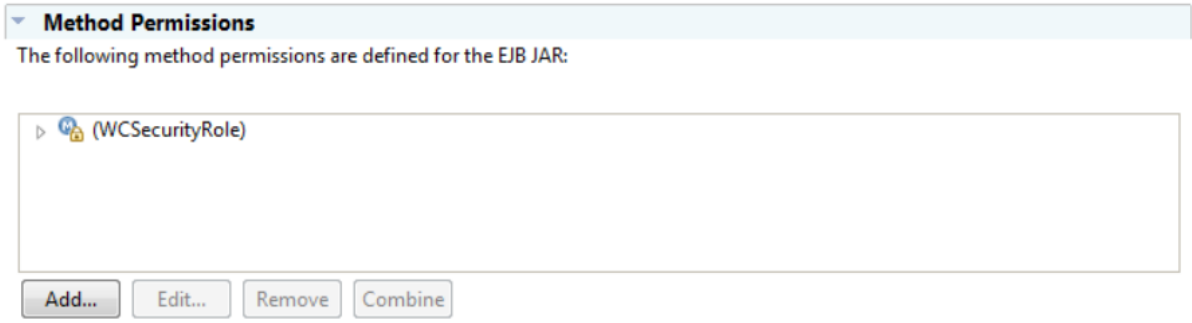

5. You will see the **XPaperCatalog** bean in the **Security Identity (Method Level)** section. Save the Deployment Descriptor. Don't close the editor yet.



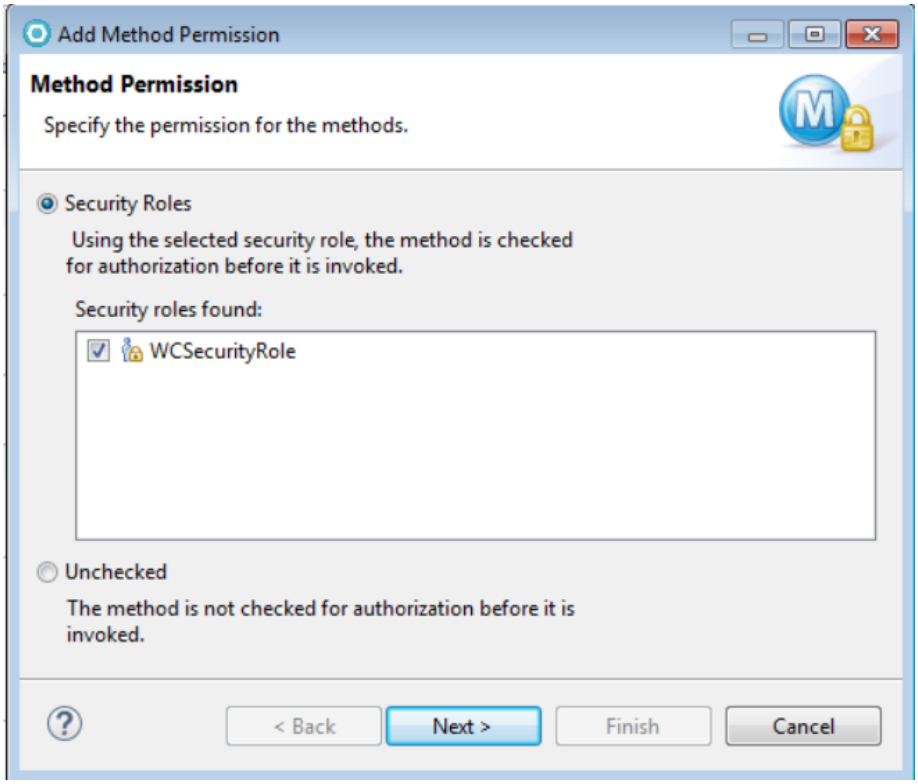

## Set the Method Permissions for security

1. Click on the **Assembly** tab.

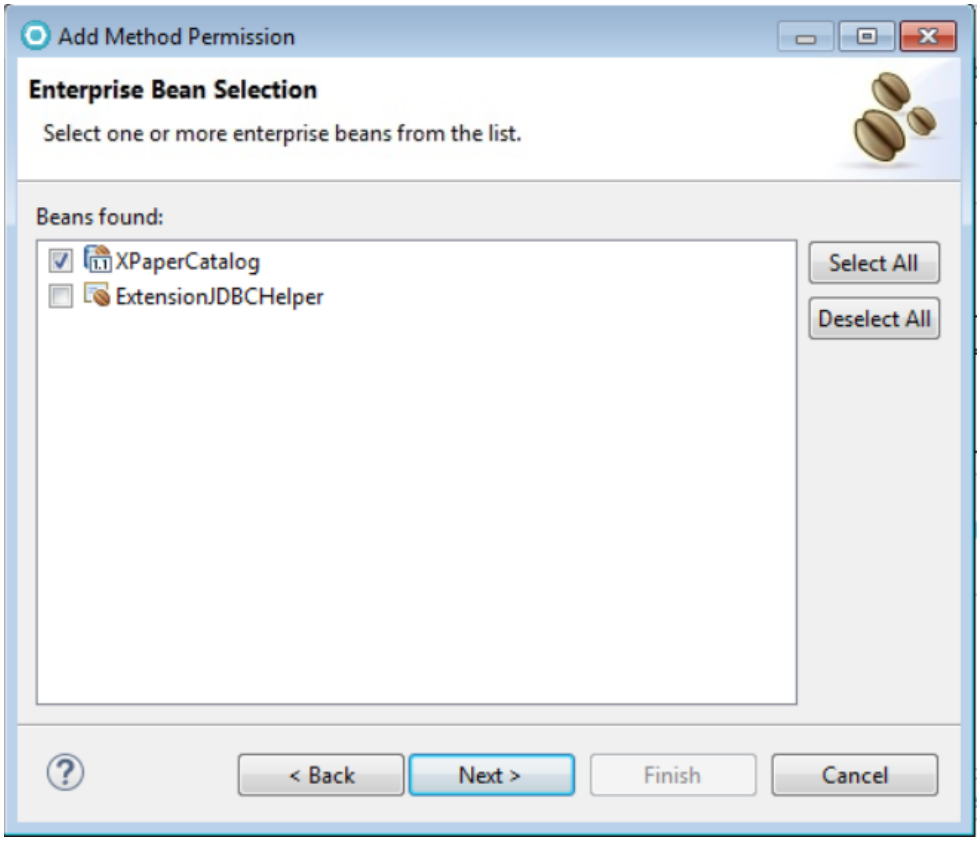
2. In the **Method Permissions** section, click **Add**.



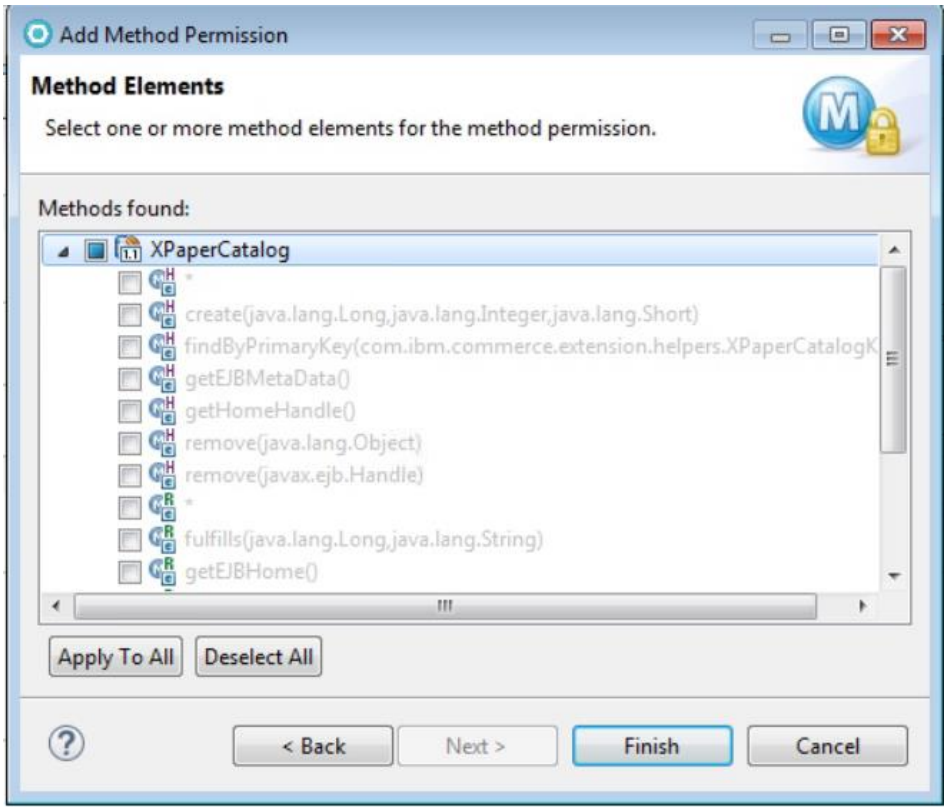

3. In the **Add Method Permission** dialog, select **Security Roles** and then select the **WCSecurityRole** check box. Click **Next**.



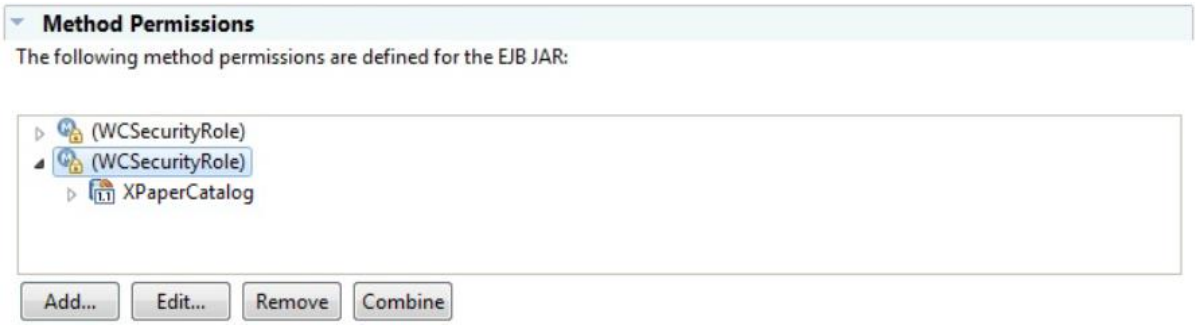

4. On the **Enterprise Bean Selection** page, select the **XPaperCatalog** check box. Click **Next**.




5. On the **Add Method Permission** page, select the **XPaperCatalog** check box. Click **Finish**.



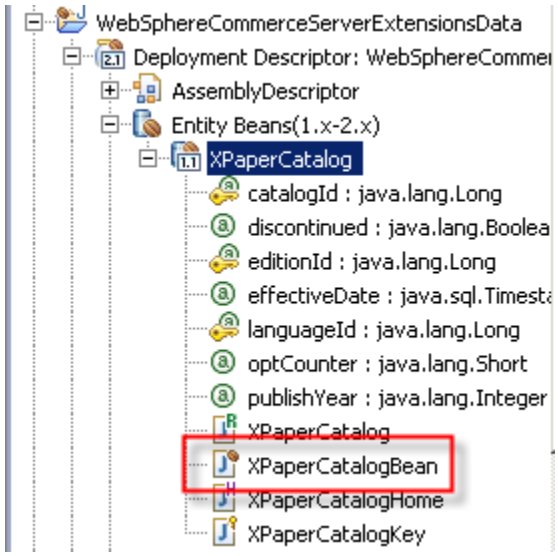

6. Save the Deployment Descriptor. Don't close the editor yet.




The next step is to remove some of the fields and methods that are related to the entity context that the WebSphere Commerce Developer generates. The reason these fields need to be deleted is that the **ECEntityBean** base class (from which **XPaperCatalog** is inherited) provides its own implementation of these methods.

## Modify EJB Methods

1. In the Java EE Perspective, Enterprise Explorer view, expand **WebSphereCommerceServerExtensionsData > Deployment Descriptor > Entity Beans (1.x-2.x) > XPaperCatalog**.

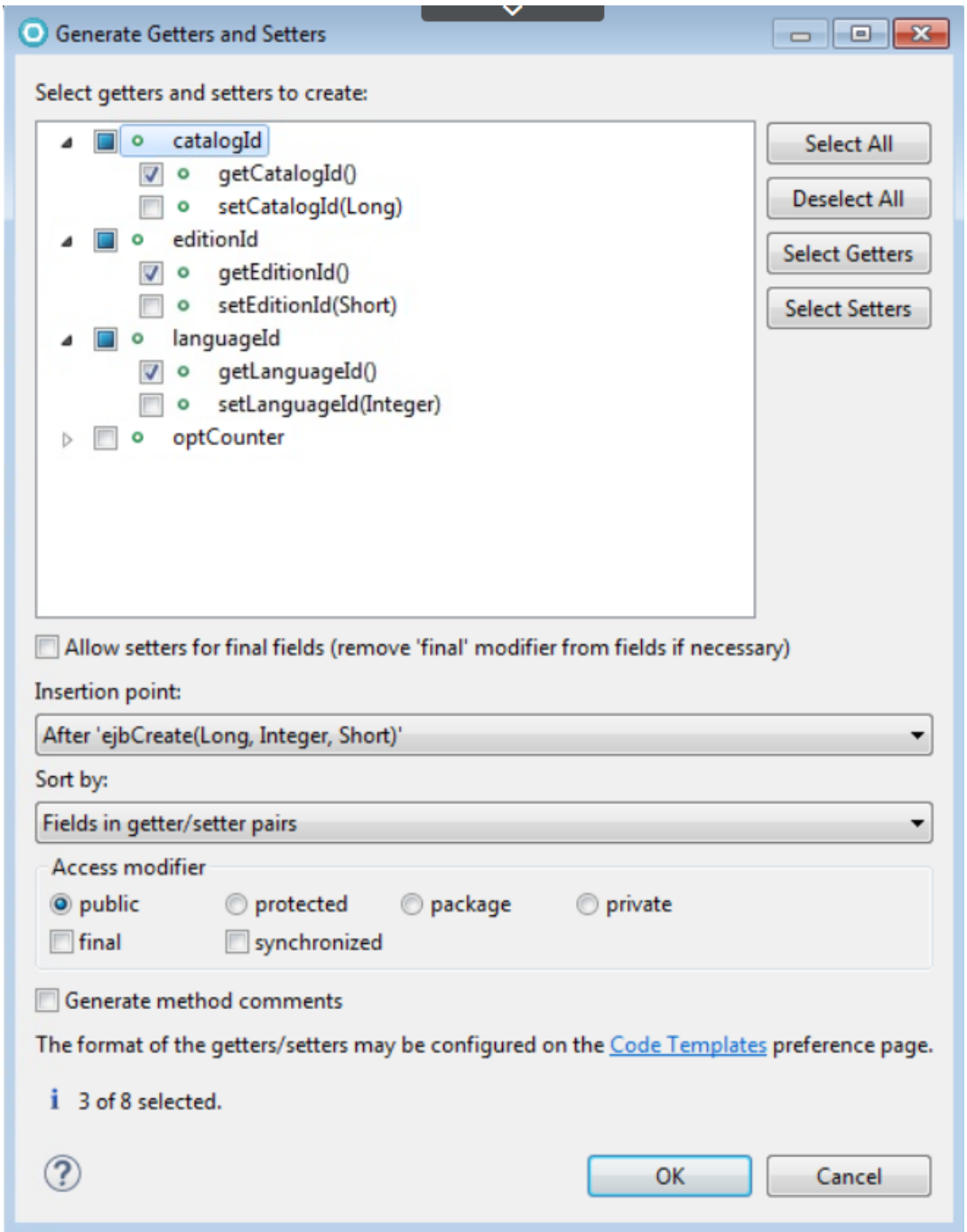



2. Double click the **XPaperCatalogBean** to open the Java editor.

3. In the **Outline** view, right click **myEntityCtx** and click **Delete**. Click **OK** to confirm. If you don't see the **Outline** view, click **Window > Show View > Outline**.

4. Repeat the previous step to delete the **getEntityContext**, **setEntityContext,** and **unsetEntityContext** methods.

5. Modify the bean to provide getter methods for the primary key fields.
    a. Right click anywhere in the code and select **Source > Generate Getters and Setters**.
    b. Select the getter methods for catalogId, languageId and editionId.
    c. Click **OK**.
    d. Save the bean.




6. In the **Outline** view, select **getCatalogId**, **getLanguageId**, and **getEditionId** methods and right click. Select **Enterprise Bean (1.x-2.x) > Promote to Remote Interface**.

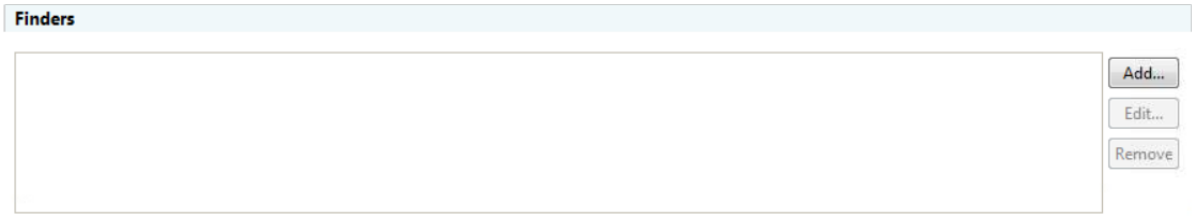
7. Save the bean and close the Java editor.

## Add EJB Finder Methods

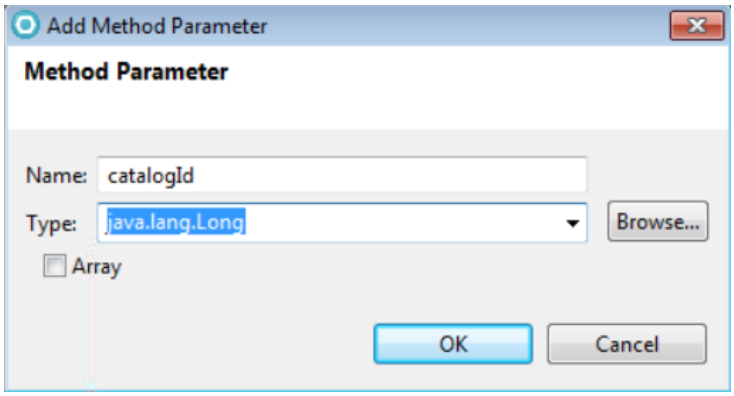
1. In the Deployment Descriptor, click the **Bean** tab.

2. Select the **XPaperCatalog** bean in the left pane.

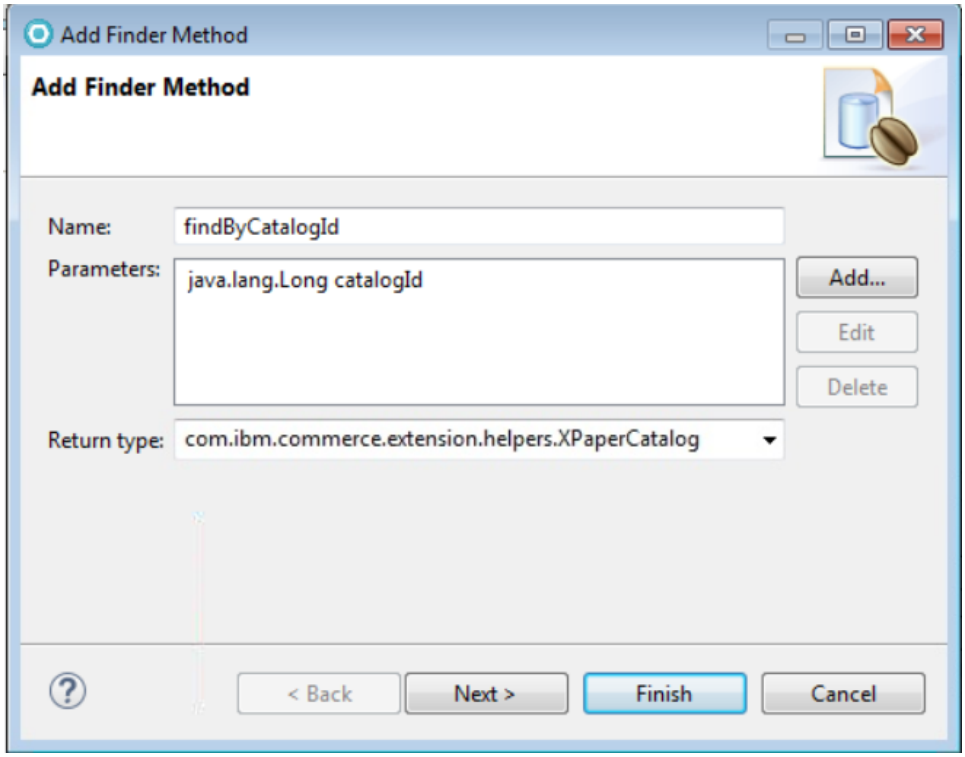
3. In the bottom-right corner section, scroll down to the **Finders** section. Click **Add**.




4. In the **Add Finder Method** dialog, enter the following information:
   - Enter name as **findByCatalogId**.
   - Add a parameter called **catalogId** with data type **java.lang.Long**.

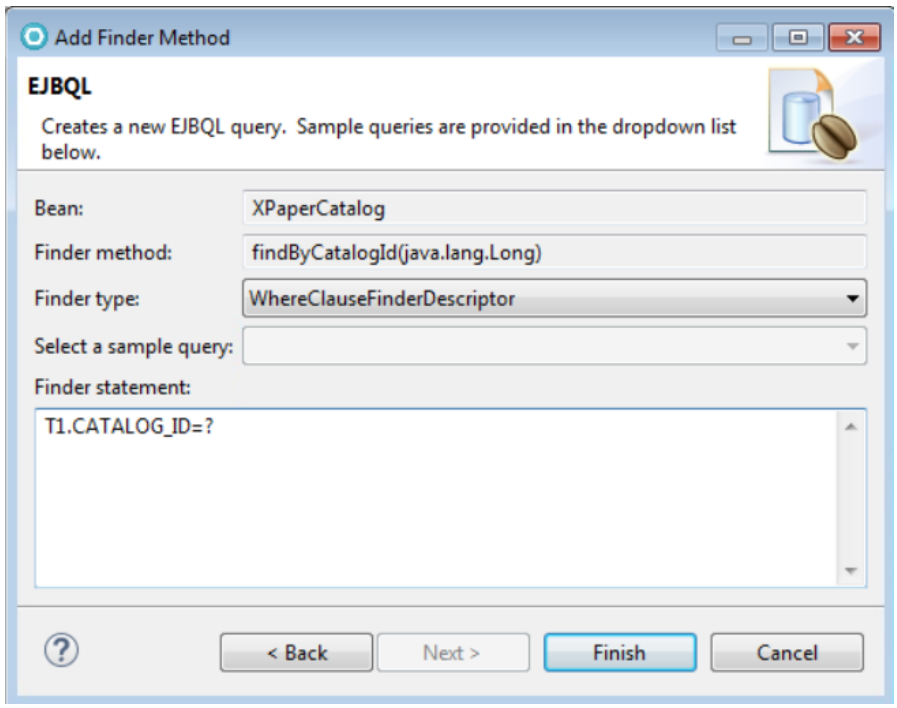



   - Click **OK**.




- Click **Next**.

5. In the **EJBQL** dialog, select **WhereClauseFinderDescriptor** as **Finder type**.
   In the **Finder statement** text area, enter: **T1.CATALOG_ID=?**
   Click **Finish**.



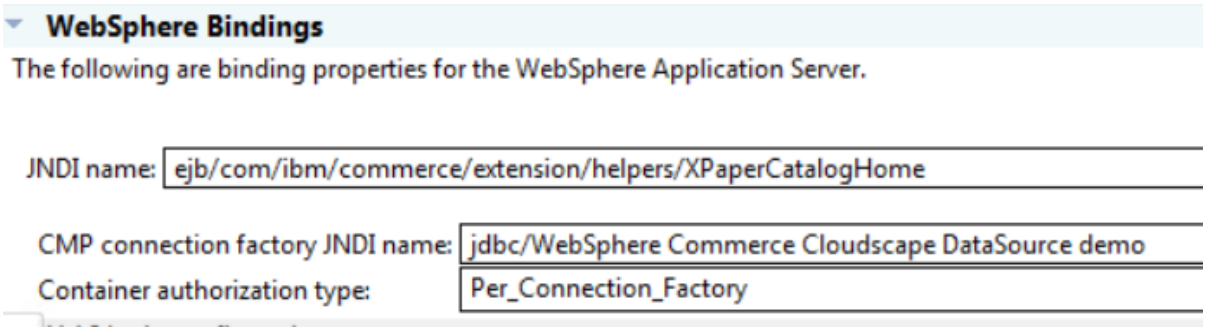
Repeating the above process, you can add any number of finder methods that you require.

## Configure JNDI

1. In the same section where you added the finder method, scroll up from the **Finders** section and come to the **WebSphere Bindings** section.

2. Set the CMP connection factory JNDI name. The spelling of the JNDI name should be exactly as given below:
   - For DB2: jdbc/WebSphere Commerce DB2 DataSource demo
   - For Derby: jdbc/WebSphere Commerce Cloudscape DataSource demo

3. Set the **Container authorization type** to **Per_Connection_Factory**.




4. Save and close the Deployment Descriptor.

## Add ejbCreate and ejbPostCreate Methods

The **ejbCreate** method is used to insert a row in to the table in the database. It is recommended to include all the columns in the **ejbCreate** signature. This allows us to create a row in the table with all values in one efficient step. Any columns that are nullable can be set to null if needed.

1. Open **XPaperCatalogBean** to view its source code.

2. Create the following **ejbCreate** method:

```
public XPaperCatalogKey ejbCreate(Long catalogId, Integer languageId,
Short editionId, Integer publishYear, String season, java.sql.Timestamp
effectiveDate, java.lang.Boolean discontinued) throws
javax.ejb.CreateException {

   this.initializeFields();
```

```
    _initLinks();
    this.catalogId = catalogId;
    this.languageId = languageId;
    this.editionId = editionId;
    this.publishYear = publishYear;
    this.season = season;
    this.discontinued = discontinued;
    this.effectiveDate = effectiveDate;
    XPaperCatalogKey key = new XPaperCatalogKey(catalogId, languageId,
editionId);
    this.initializeOptCounter(key);


    return null;
}
```

**Note:** This code is also available in **C:\exercises\create-ejb\CodeSnippet1.txt**. If you copy the code into the workspace using Windows File Explorer, make sure you refresh the **WebSphereCommerceServerExtensionsData** project in RAD, so that the changes are detected and compiled.

3. Create an **ejbPostCreate** method with the same signature as the **ejbCreate** method you just created. It will have an empty implementation.

```
public void ejbPostCreate(Long catalogId, Integer languageId, Short
editionId, Integer publishYear, String season, java.sql.Timestamp
effectiveDate, java.lang.Boolean discontinued) throws
javax.ejb.CreateException { }
```

**Note:** This code is also available in **C:\exercises\create-ejb\CodeSnippet2.txt**. If you copy the code into the workspace using Windows File Explorer, make sure you refresh the **WebSphereCommerceServerExtensionsData** project in RAD, so that the changes are detected and compiled.

4. In the **Outline** view, right click the **ejbCreate** method you created and click **Enterprise Bean (1.x-2.x) > Promote to Home Interface**. (Ensure that you pick the correct **ejbCreate** method, since there are two of them.)

5. Save the changes.
6. Add the following methods to **XPaperCatalogBean** so that it can be protected by the access control system.

```
 /**
* Since this bean is really a child to the Catalog bean, just
* delegate to the Catalog bean's getOwner and fulfills methods.
*/


public Long getOwner() throws Exception {
CatalogAccessBean bean = getCatalogBean(getCatalogId());
return bean.getOwner();
}


public boolean fulfills(Long owner, String relation) throws Exception {
CatalogAccessBean bean = getCatalogBean(getCatalogId());
return bean.fulfills(owner, relation);
}


private CatalogAccessBean getCatalogBean(Long catalogId) throws
Exception {
Integer storeId = null;
StoreCatalogAccessBean scBean = null;
Enumeration scEnum = new
StoreCatalogAccessBean().findByCatalogId(getCatalogId());


if (scEnum != null) {
   while (scEnum.hasMoreElements()) {
         scBean = (StoreCatalogAccessBean)scEnum.nextElement();
         if (scBean.getMasterCatalog().equalsIgnoreCase("1")) {
               storeId = scBean.getStoreEntryIDInEJBType();
               break;
         }
   }
```

```
}


CatalogAccessBean bean = new CatalogAccessBean();
CatalogAccessBean result = null;


try {
   result = bean.findByCatalogIdentifierAndStore(catalogId.toString(),
storeId);
} catch (RemoteException exception) {
   throw exception;
} catch (FinderException exception) {
   throw exception;
} catch (NamingException exception) {
   throw exception;
}
return result;
}
```

**Note:** This code is also available in **C:\exercises\create-ejb\CodeSnippet3.txt**. If you copy the code into the workspace using Windows File Explorer, make sure you refresh the **WebSphereCommerceServerExtensionsData** project in RAD, so that the changes are detected and compiled.

7. You might find some errors in the code as the required packages do not get imported. From the context menu in the editor, select **Source > Organize Imports**.
8. In the **Outline** view, select **ejbLoad** and add the following code as the first line in the method: `super.ejbLoad();`
9. In the **Outline** view, select **ejbStore** and add the following code as the first line in the method: `super.ejbStore();`

   The resulting methods should resemble the following screenshots:

```
/**
 * ejbLoad
 */
public void ejbLoad() {
    super.ejbLoad();
    _initLinks();
}
```

```
/**
 * ejbStore
 */
public void ejbStore() {
    super.ejbStore();
}
```

**Note:** The completed **XPaperCatalogBean** is available in **C:\exercises\create-ejb\ejb**. If you copy the code into the workspace using Windows File Explorer, make sure you refresh the **WebSphereCommerceServerExtensionsData** project in RAD, so that the changes are detected and compiled.

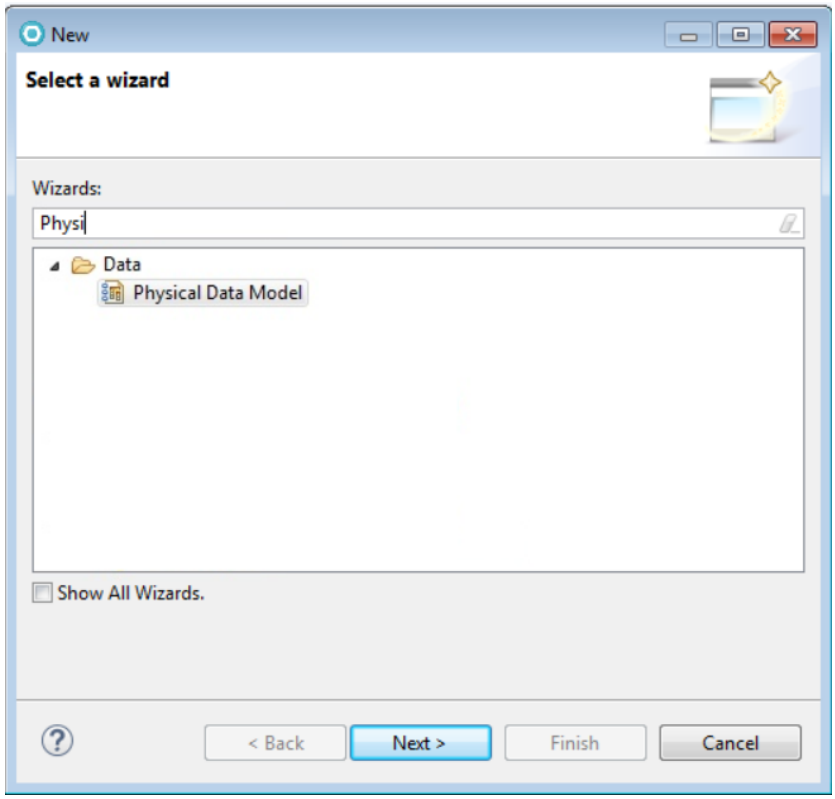
## Part 5: Map the database table to the EJB

## Creating a database and table definition

1. In the Java EE perspective, Enterprise Explorer view, right click **WebSphereCommerceServerExtensionsData** and select **New > Other**.

   **Important:** It is important that you right click on **WebSphereCommerceServerExtensionsData** only, and not on any other item (for example, the Deployment Descriptor, when following these steps. This will ensure that the destination folder for the physical data model is correctly populated automatically in a subsequent step.

2. In the wizard, select **Data > Physical Data Model**. Click **Next**.



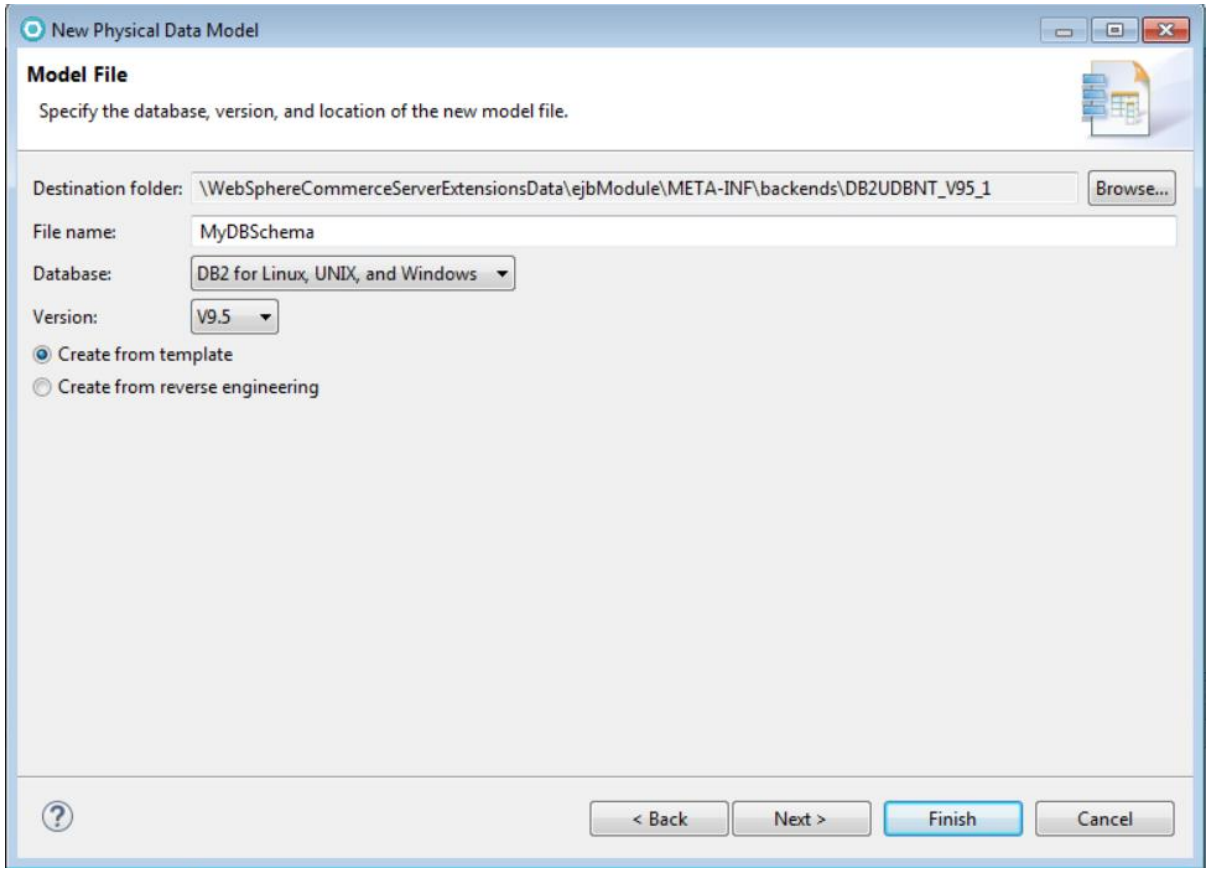

3. Enter **MyDBSchema** as the schema name.

4. Select **DB2 for Linux, UNIX, and Windows** as the database type. Selection **Version** as **9.5**.

---

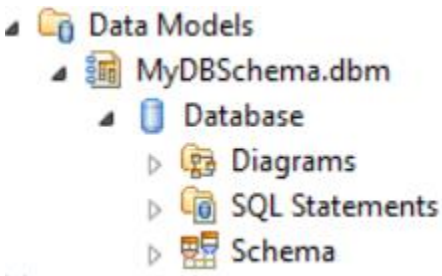
**Important:** Even if you are using **Apache Derby** as the development database, you need to select **DB2** as the database type in the above step. This is so that the generated entity bean is compatible with DB2. This assumes that, in a real-life scenario, the production environment would be using DB2 (since Derby is supported only for WC Developer).

---




5. Click **Finish**.

6. Navigate to **WebSphereCommerceServerExtensionsData > Data Models > MyDBSchema.dbm**. You will see a **Database** option under it.

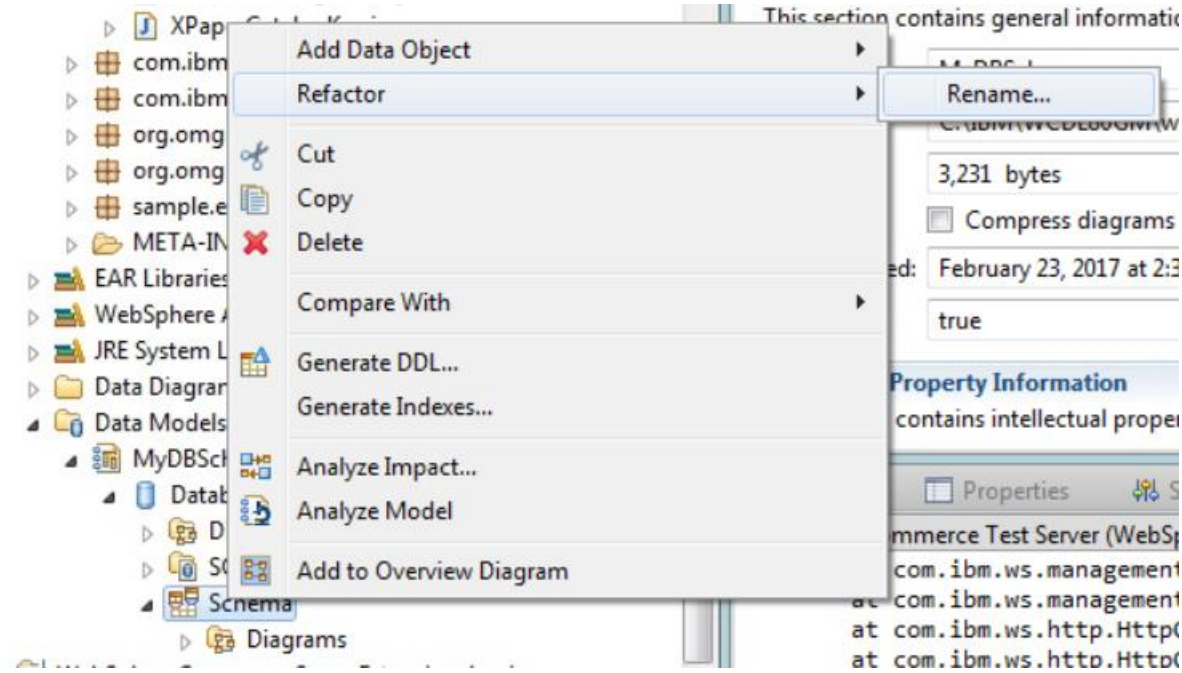



7. Right click on **Database > Schema** and select **Refactor > Rename**.

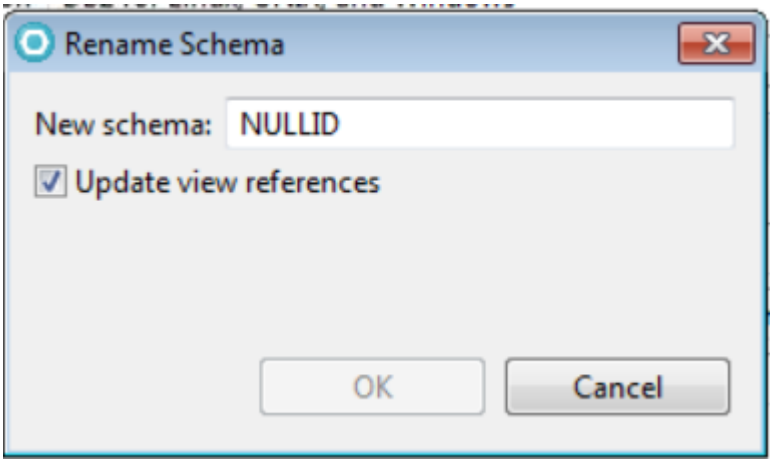
---

**Note:** If you don't see the **Database** option under **MyDBSchema.dbm**, then double click on **MyDBSchema.dbm** to open the file. The **Database** option will become visible.

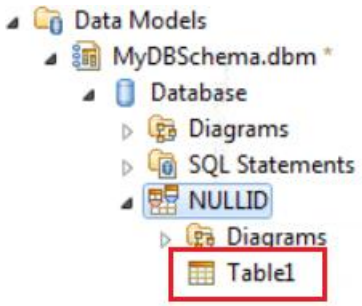
---



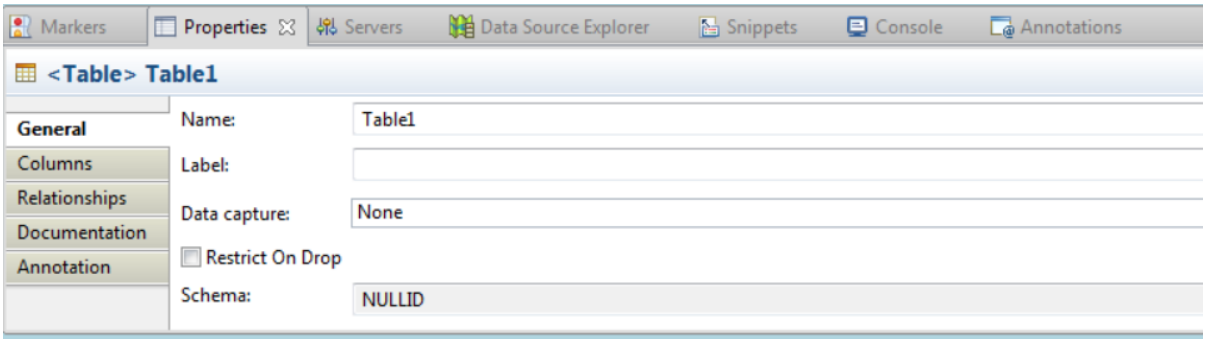

8. Rename the schema to **NULLID**. Select the **Update view references** check box. Click **OK**.




9. Right click **NULLID** and select **Add Data Object > Table**. A new table is created.




10. Select **Table1**. In the **Properties** view, a new table form opens.



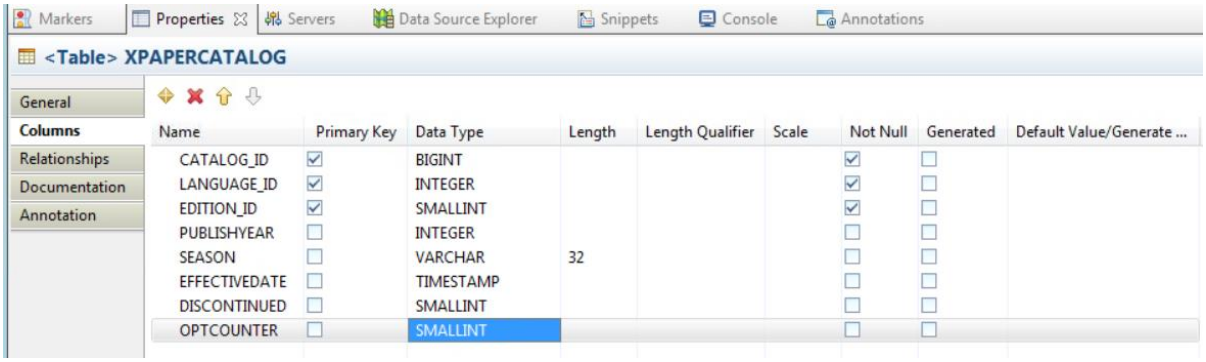

11. Change the table name to **XPAPERCATALOG**.

12. Move to **Columns** tab and create the columns as given in the following table:

| Name | Type | Primary Key |
|---|---|---|
| CATALOG_ID | BIGINT | true |
| LANGUAGE_ID | INTEGER | true |
| EDITION_ID | SMALLINT | true |
| PUBLISHYEAR | INTEGER | false |
| SEASON | VARCHAR (32) | false |
| EFFECTIVEDATE | TIMESTAMP | false |
| DISCONTINUED | SMALLINT | false |
| OPTCOUNTER | SMALLINT | false |

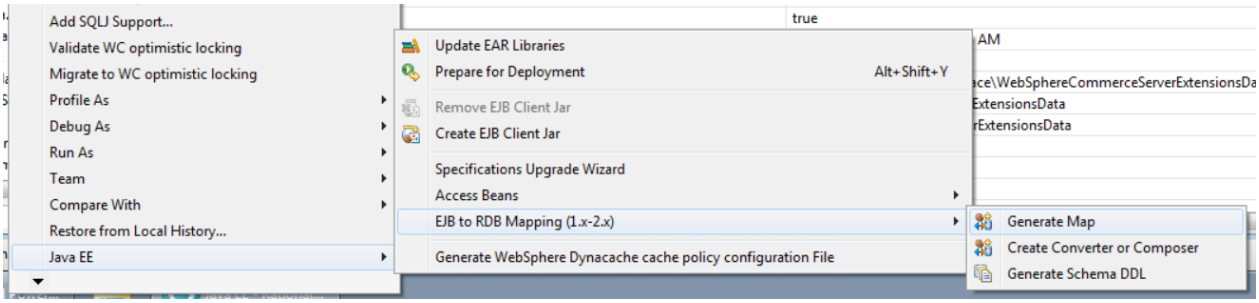
The table definition should match the definition of the actual table created in the database earlier.

13. Save the changes.

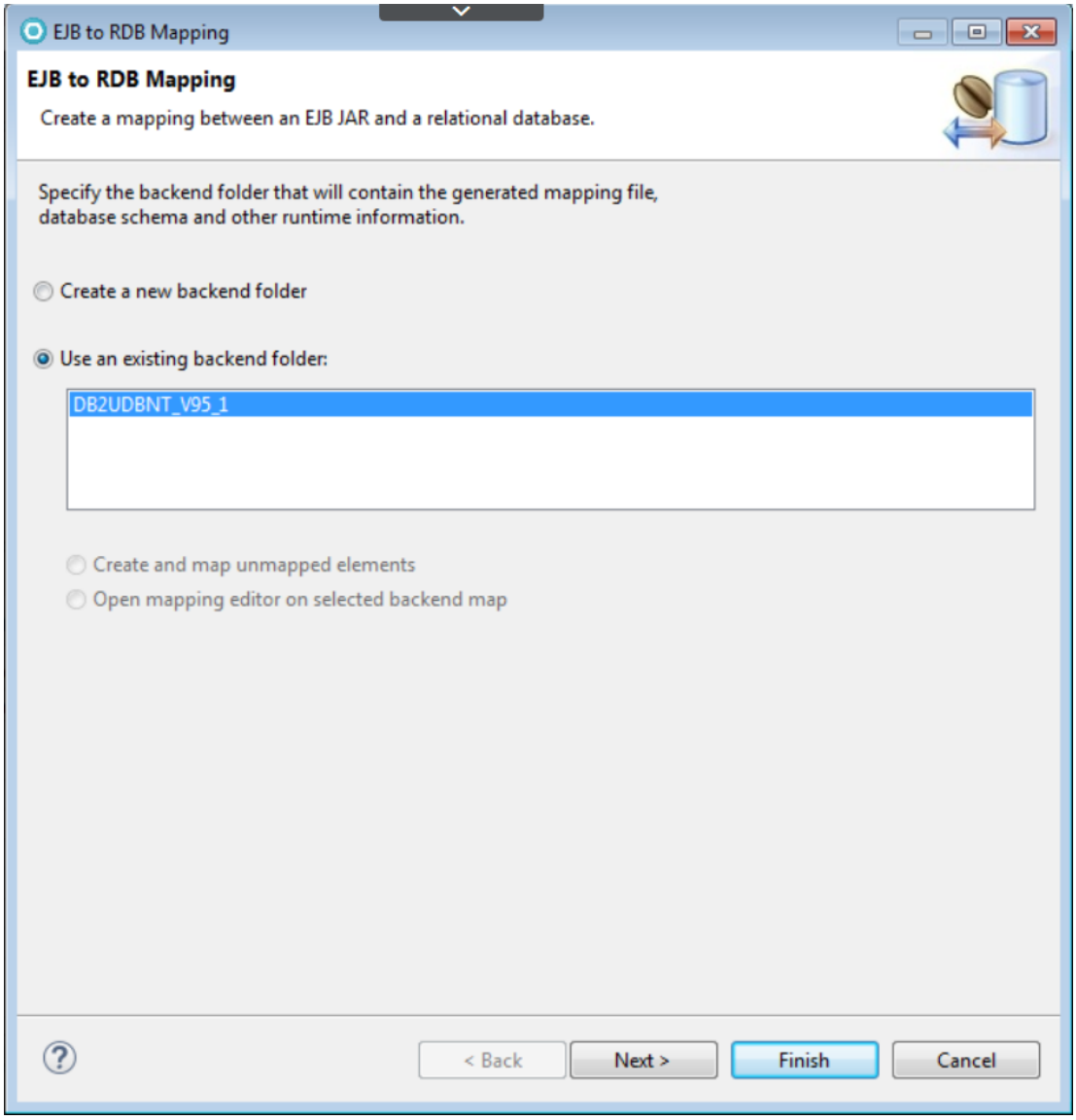



## Map the table to the entity bean

1. In the Java EE perspective, right click **WebSphereCommerceServerExtensionsData** and select **Java EE > EJB to RDB Mapping (1.x-2.x) > Generate Map**.

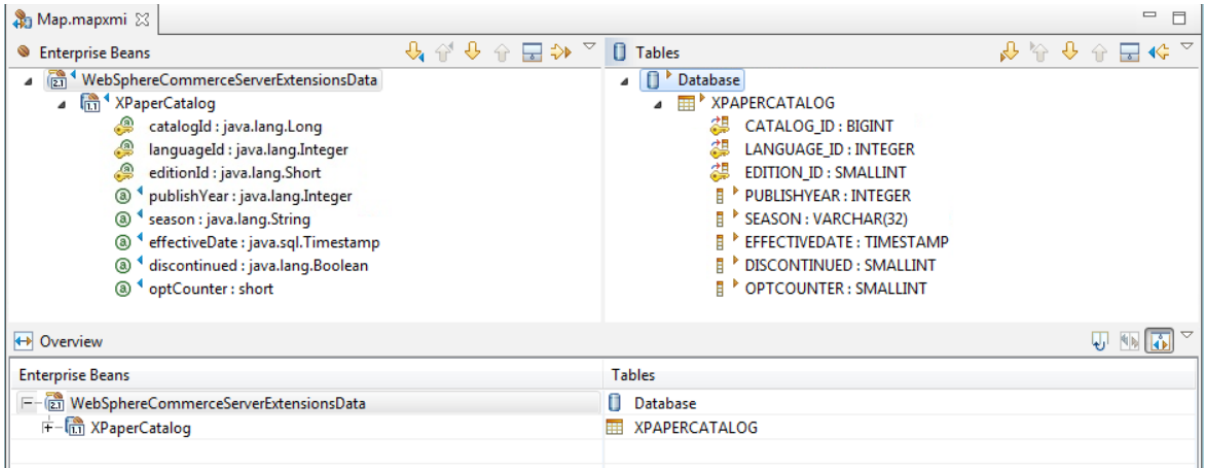



2. In the mapping wizard, the existing backend folder option should be selected by default. Click **Next**.



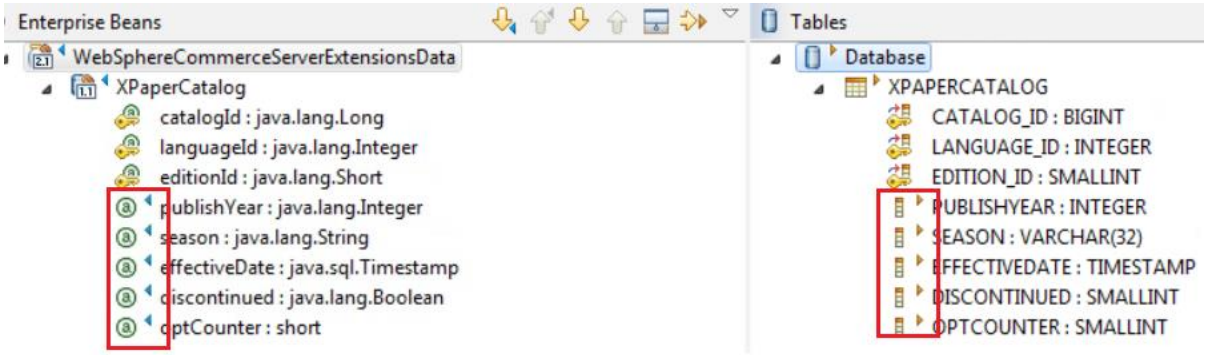

3. Select **Meet-in-the-middle** and click **Next**.

4. Select **Match by name** as the mapping option and click **Finish**. The **Map.mapxmi** editor opens.




5. In the **Enterprise Beans** pane, highlight **XPaperCatalog**. In the **Tables** pane, highlight the **XPAPERCATALOG** table.

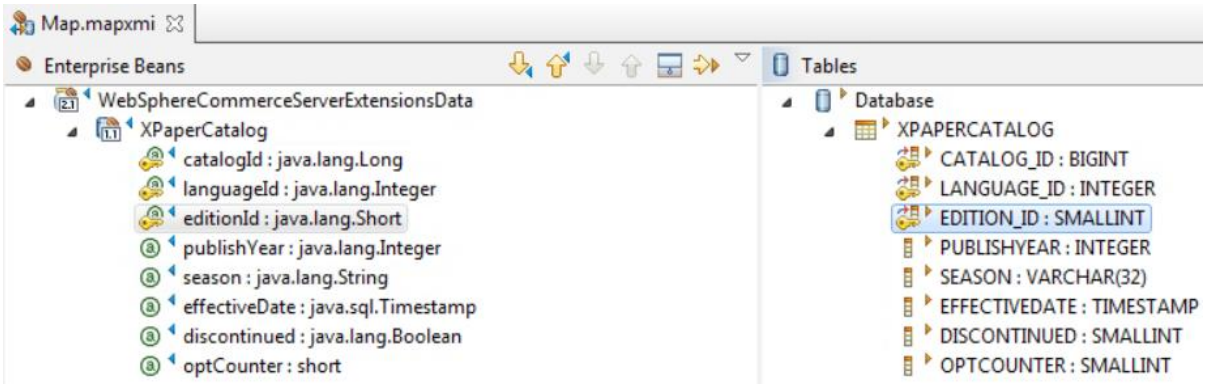
Since we selected the **Match by name** option in the previous step, you can see that the bean fields and table columns that have the same names have already been mapped. You can tell this by the small triangles next to the bean fields and table columns.




By the same token, you can tell that **catalogId**, **languageId** and **editionId** fields have not been mapped to the corresponding columns. This is because the names do not match. These fields need to be mapped manually.

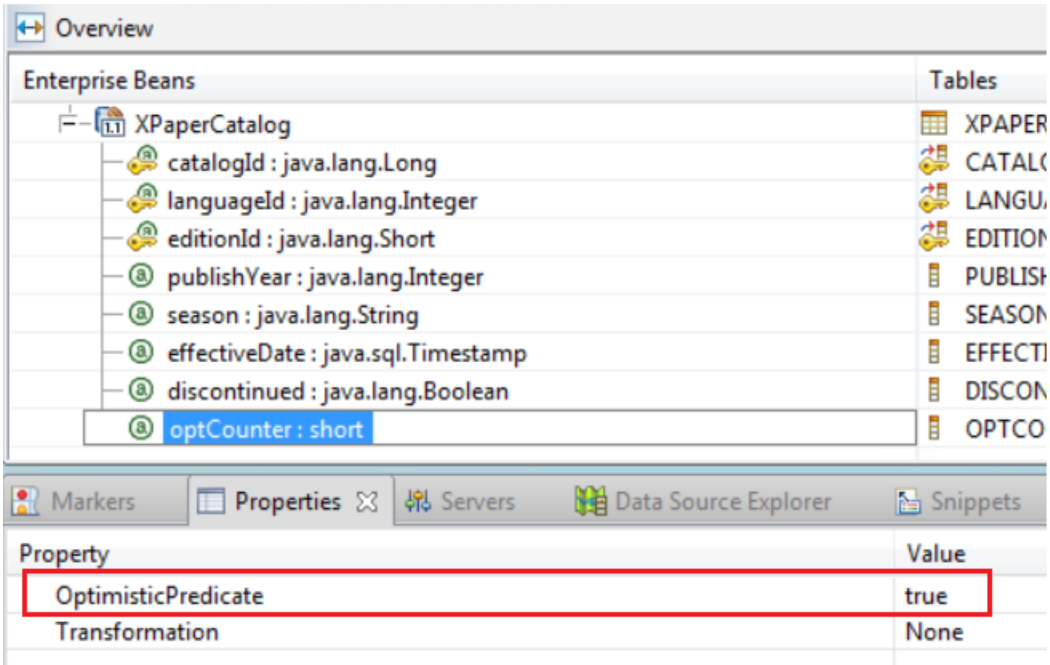
6. Highlight **catalogId** in the **Enterprise Beans** pane, and **CATALOG_ID** in the **Tables** pane. Right click on **CATALOG_ID** and select **Create Mapping**. Repeat the same for **languageId** and **editionId**.

7. Save the changes and leave **Map.mapxmi** open. All the fields should now be mapped.




8. Verify that all the mappings between the EJB and the table are correct in the **Overview** pane. When you select a field from the **Overview**, the matching Enterprise Bean field and column from the table is selected automatically.

9. In the **Overview** pane, select the **optCounter** field. In the **Properties** pane, set **OptimisticPredicate** to **true**.




10. Save the changes and close **Map.mapxmi**.

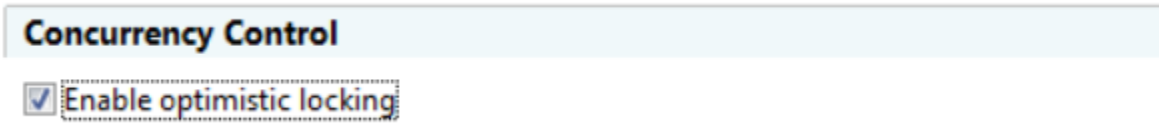
## Enable optimistic locking

1. In the Java EE Enterprise Explorer view, expand **WebSphereCommerceServerExtensionsData > Deployment Descriptor > Entity Beans (1.x-2.x)**.

2. Right-click the **XPaperCatalog** bean and open it in the deployment descriptor.

3. Open the **Bean** tab, and select the **XPaperCatalog** bean.

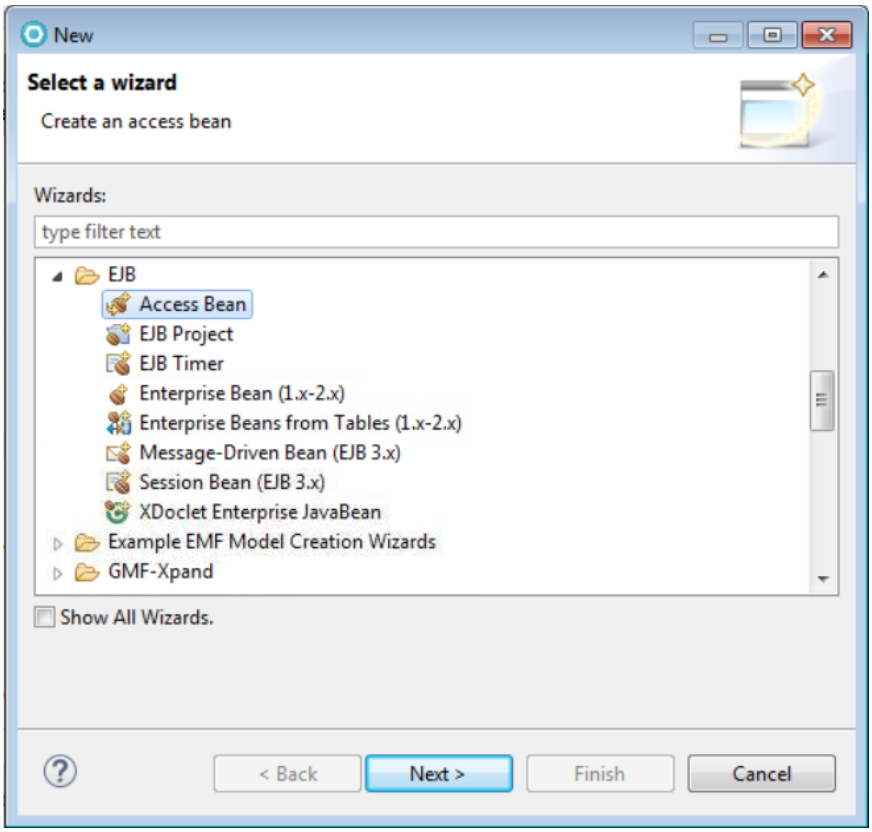
4. In the lower right pane, scroll down to the **Concurrency Control** section and select the **Enable optimistic locking** check box.




5. Save the changes and close the deployment descriptor.

## Part 6: Generate the Access Bean for the EJB

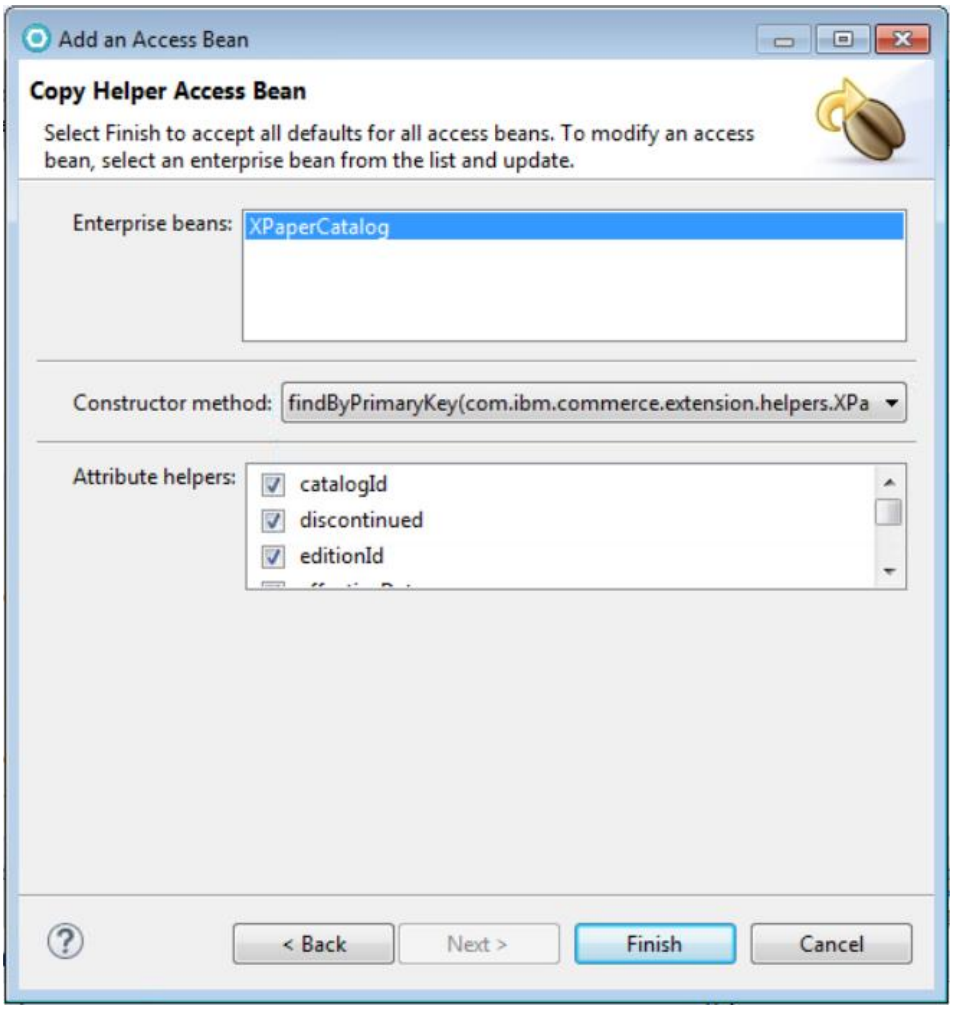
1. In the Java EE perspective, Enterprise Explorer view, right-click **WebSphereCommerceServerExtensionsData**. Select **New > Other**. In the list of options, expand **EJB** and select **Access Bean**. Select **Next**.



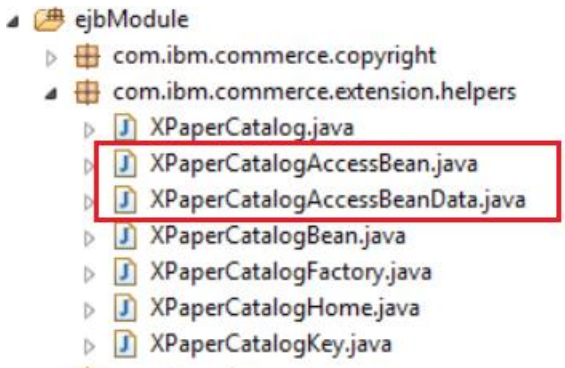

2. On the **Select an Access Bean Type** page, select **Copy helper**. Click **Next**.

3. On the **Select EJB Project** page, select the **XPaperCatalog** bean. Click **Next**.

4. On the **Copy Helper Access Bean** page, for the **Constructor method**, select **findByPrimaryKey (com.ibm.commerce.extension.helpers.XPaperCatalog)**. Click **Finish**.



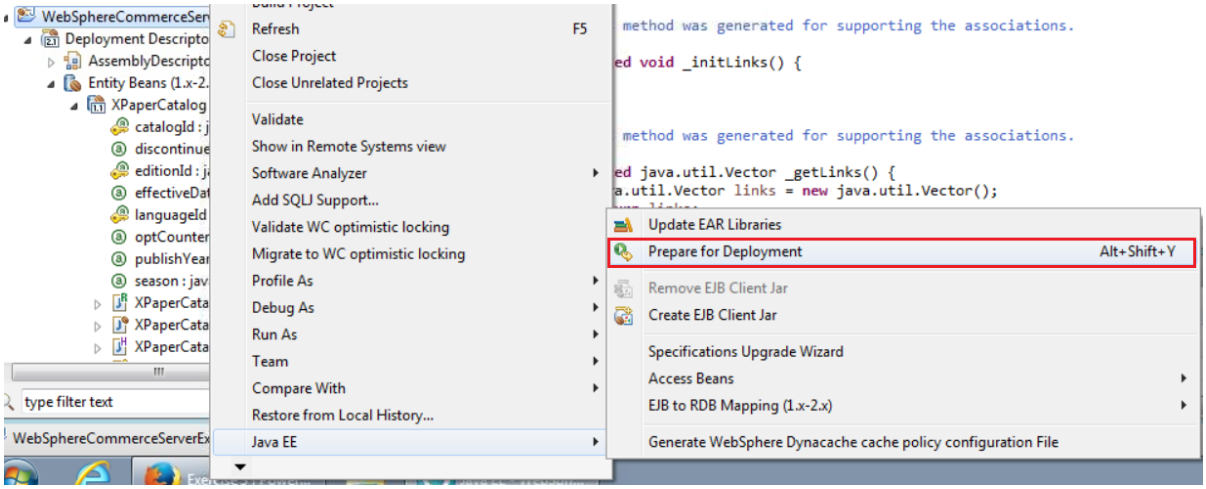

The access bean classes are generated. You can find them under **ejbModule**.




---

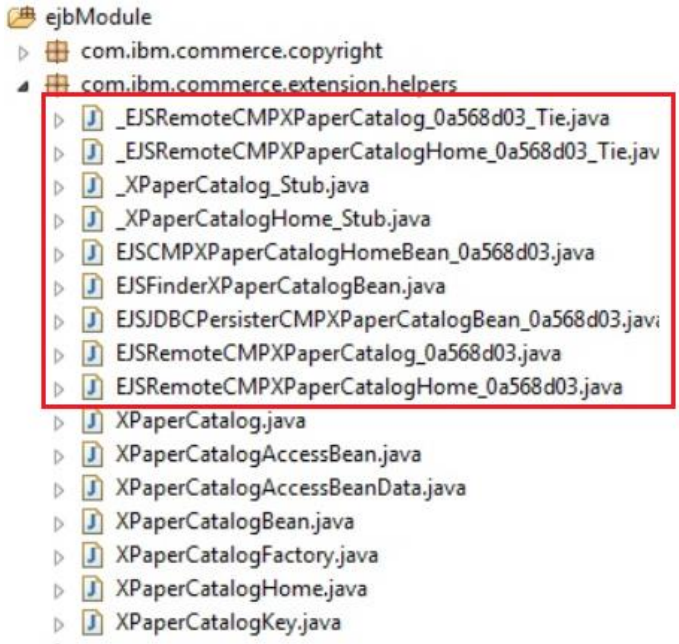
**Note:** All the Java files related to the EJB and access bean are also provided in **C:\exercises\create-ejb\ejb** directory for your reference. If you copy the files into the workspace using Windows File Explorer, make sure you refresh the **WebSphereCommerceServerExtensionsData** project in RAD, so that the changes are detected and compiled.

5. Right-click **WebSphereCommerceServerExtensionsData** and click **Java EE > Prepare for Deployment**.




After the deploy code is generated, Rational Application Developer rebuilds the workspace. When the workspace building completes, ensure you have no errors in the Problems view.

You can find the generated classes under **ejbModule**.

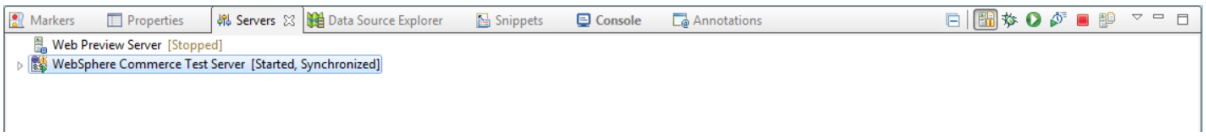



**Troubleshooting:** In case you get any errors while generating the deployment code, you can go through this log file: **C:\IBM\WCDE80GM\workspace\.metadata\.log**.

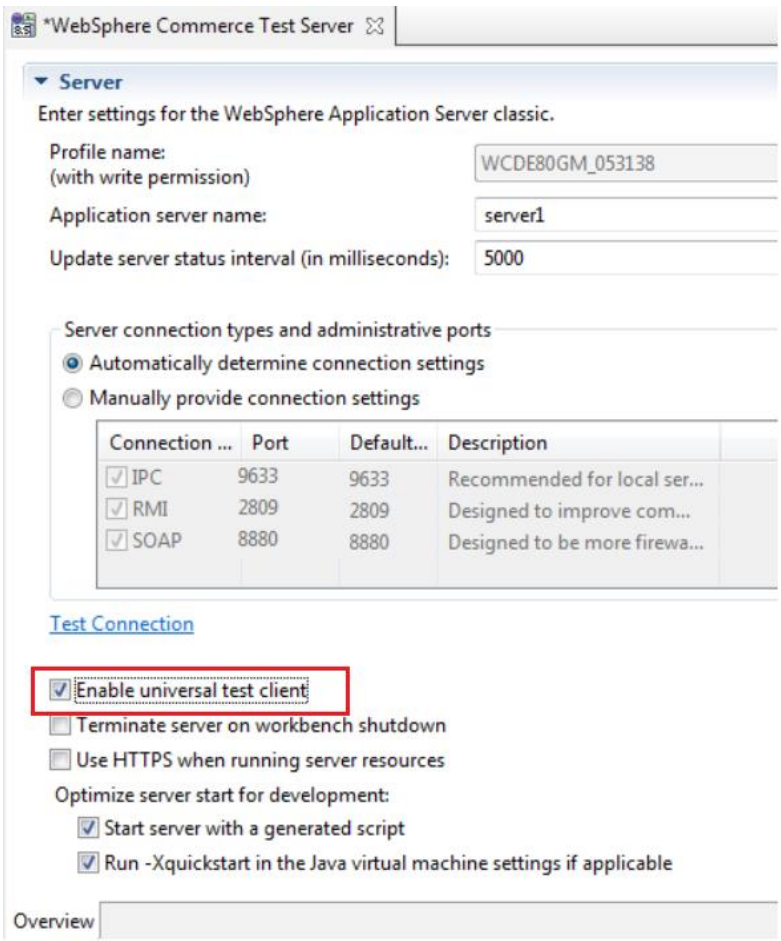
**Note:** Sometimes there is a need to regenerate the access bean after deployment. To regenerate the access bean, right click the Deployment Descriptor and click **Java EE > Access Beans > Regenerate Access Beans.**

## Part 7: Test the Entity bean

1. Restart WebSphere Commerce Test Server. Republish the WC application, if needed.
2. Once the server is started, double-click the server in the **Servers** view.

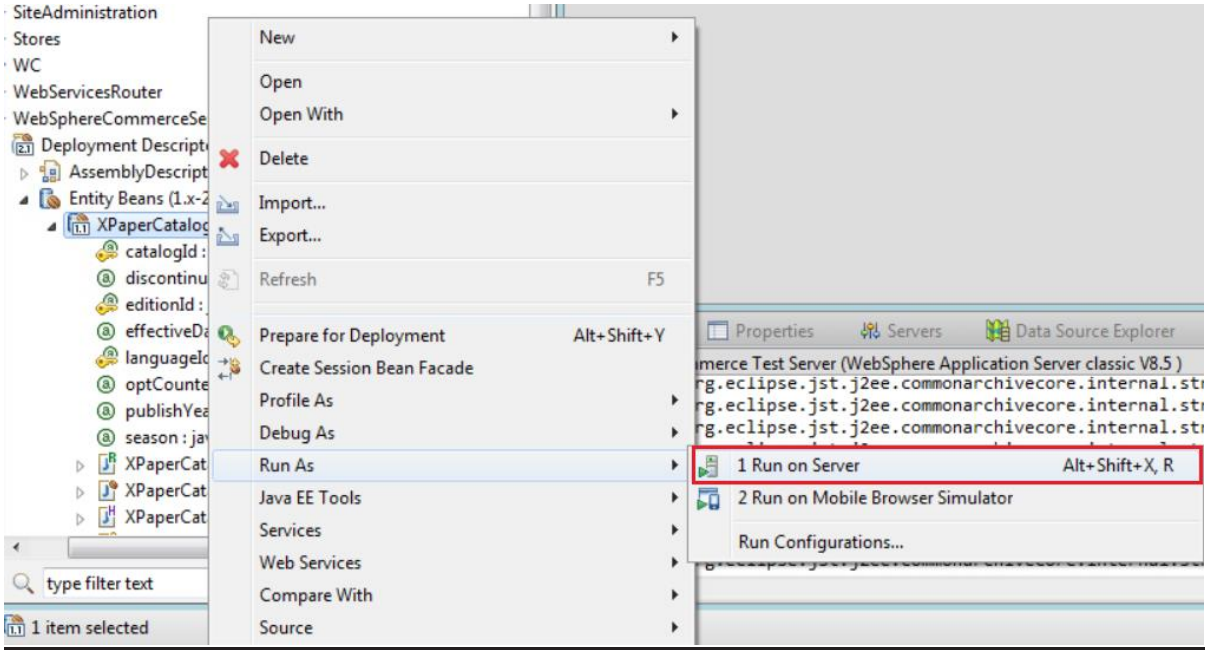



3. Select the **Enable universal test client** check box. This publishes the Universal Test Client, a web application that is used for testing EJBs.




4. Save the changes and close the **Server Overview** window.
5. In the Enterprise Explorer, expand **WebSphereCommerceServerExtensionsData > Deployment Descriptor > Entity Beans (1.x-2.x)**.

6. Right-click **XPaperCatalog** and click **Run As > Run on Server**.




7. On the **Run on Server** page, click **Finish**. This action starts the test client. It takes several minutes for the test client to display.

---

**Troubleshooting:** In some cases, the UTC application is not started and the console shows an error that indicates:

A WebGroup/Virtual Host to handle /UTC/jndiLookup has not been defined.

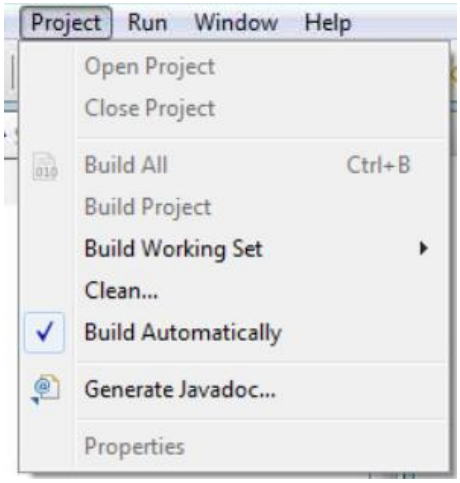
In such cases, leave the server running, open the Server Overview again, clear the **Enable test client** check box, save the configuration, select the **Enable test client** check box, and save again. The UTC can be published and started now. You can repeat Step 5.

---

**Troubleshooting:** If the test client is launched but the **XPaperCatalog** bean is not shown in the test client, perform a project clean against the **WebSphereCommerceServerExtensionsData** project.

Click **Project** > **Clean...**> **Clean projects selected below.** Ensure that the **WebSphereCommerceServerServerExtensionsData** project is selected. Click **Build only the selected projects** option and click **OK.**

Also, ensure that the **Project > Build Automatically** option is enabled in RAD. This way you won't need to manually build the projects every time a code change is made.

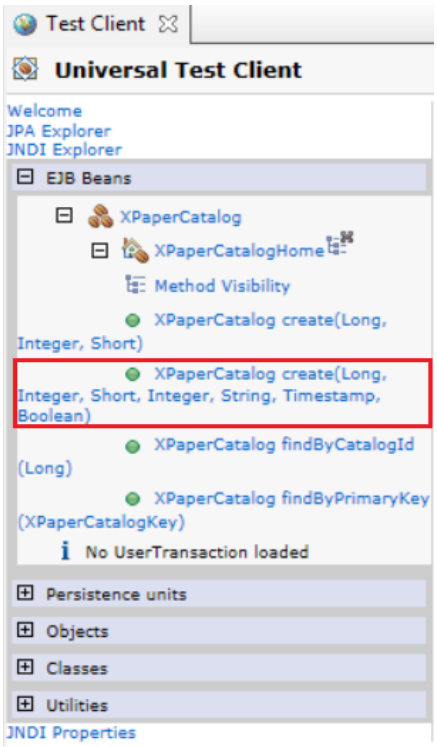



Once the build is complete, publish the changes to the server and restart the server. Rerun the step to run the test client.

Another common cause is the name of the data source that is defined for the bean. It must be precise. Verify that the **JNDI** name used for the bean is **jdbc/WebSphere Commerce Cloudscape DataSource demo**.

---

8. The **XPaperCatalog** bean will be displayed in the test client. Expand **EJB Beans > XPaperCatalog**. Click **XPaperCatalog create(Long, Integer, Short, Integer, String, Timestamp, Boolean)**.

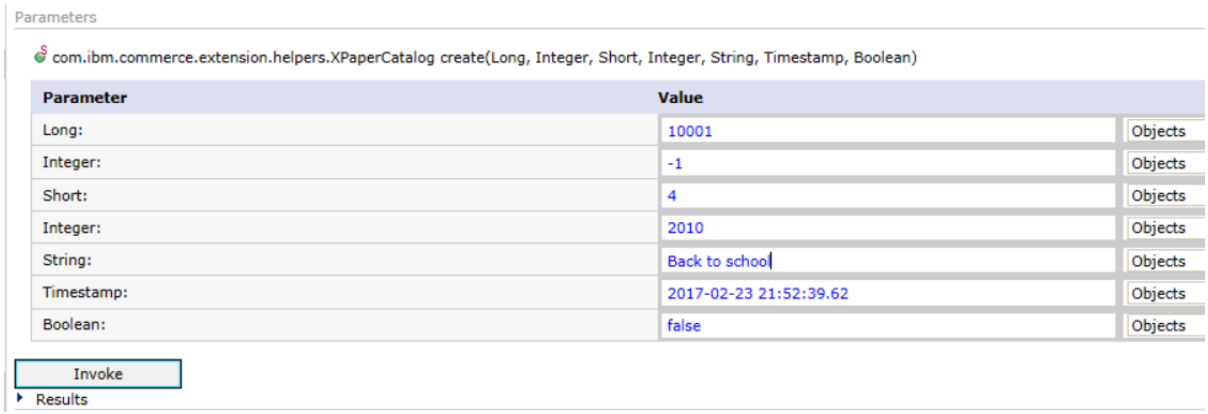



9. Enter the following values in the form on the right-hand side, in the sequence given below:
   - Long: 10001
   - Integer: -1
   - Short: 4
   - Integer: 2010
   - String: Back to school
   - Leave the Timestamp and Boolean fields with the default values.

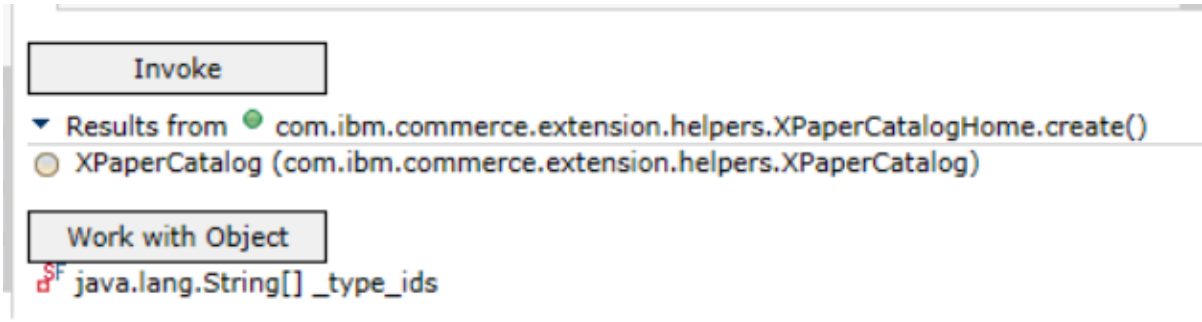
The EJB field names are not shown in the form. Here is the actual **ejbCreate** method signature for reference:

```
ejbCreate(Long catalogId, Integer languageId, Short editionId,
Integer publishYear, String season, java.sql.Timestamp
effectiveDate, java.lang.Boolean discontinued)
```

As you can see, the first field (Long) is the **catalogId**. In our case, this is **10001**. If you are using a different environment, the value may be different. You can verify the value by querying the **CATALOG** table for the catalog id of the master catalog **Extended Sites Catalog Asset Store**. The second field (Integer) is the **langId**. In our case, this is **-1** (US English). For rest of the fields, you may give values different from the ones given here, if you wish.




10. Click **Invoke**. The **Results** section should be updated and should look something like this:




**Troubleshooting:** You can get a **javax.ejb.DuplicateKeyException** error if you click **Invoke** twice, because you can create an entry with the same **editionId** (third field, Short) only once as it is part of the table's primary key. In that case, either delete the record from the table or rerun the test by providing a different value for **editionId**.

11. Now, we will verify that the row has been created successfully in **XPAPERCATALOG** table.
    - In a web browser, enter the following URL: http://localhost/webapp/wcs/admin/servlet/db.jsp
    - In the text area, enter the following SQL query:

```
select * from XPAPERCATALOG where EDITION_ID=4;
```

    - Verify that the query executed successfully and returned one result.

Enter SQL statements then click **Submit Query**. Terminate all SQL statements with a semi-colon (;)

```
select * from XPAPERCATALOG where EDITION_ID=4;
```

Submit Query  Clear All

**Query: select * from XPAPERCATALOG where EDITION_ID=4**

| CATALOG_ID | LANGUAGE_ID | EDITION_ID | PUBLISHYEAR | SEASON | EFFECTIVEDATE |
|---|---|---|---|---|---|
| 10001 | -1 | 4 | 2010 | 'Back to school' | 2017-02-23 21:52:39.62 |

## What you did in this exercise

In this exercise, you learned how to create an entity EJB. You learned how to configure properties of the bean, and how to create methods and a Finder for it. You also learned how to map the EJB to a database table. An Access bean and deployment code were generated. Finally, you learned how to test the EJB without writing code.

**End of exercise**

# Exercise 4. Extending an SOI service with UserData

## Estimated time

02:00 hours

## What this exercise is about

Extending SOI service with User data

## What you should be able to do

This exercise is designed to enable you to:

- Extend a noun to add new attributes with the UserData method
- Change the store to populate and display the new attributes

## Introduction

This tutorial is designed to show you the steps involved in extending the OrderItem noun of the Web services architecture to include new information. UserData will be used to transfer the new Order information from the front end to the WebSphere Commerce server where it persists to the database. In this tutorial, you customize the shopping flow to allow the customer the ability to record engraving information for an Order Item. After this customization, a customer begins to see options within their shopping cart before checking out for any items that can be engraved. They will be provided with options for the text to be engraved as well as the size and font of that text.

## Requirements

- WebSphere Commerce Developer V8
- Extended sites store archive published
- AuroraESite store created

# Exercise Instruction

## Part 1: Preparing the database for new engraving attributes

In order to capture the engraving information in the shopping cart, we will use WebSphere Commerce personalization attributes. These attributes are not exposed in Management Center, so you need to use SQL to define them. In this step, you create engraving attributes. These attributes are defined in the **PATTRIBUTE** table. You can create custom personalization attributes for products & items.

1. Start WebSphere Commerce Developer. Start the test server.
2. Open a new browser window and enter the following URL:

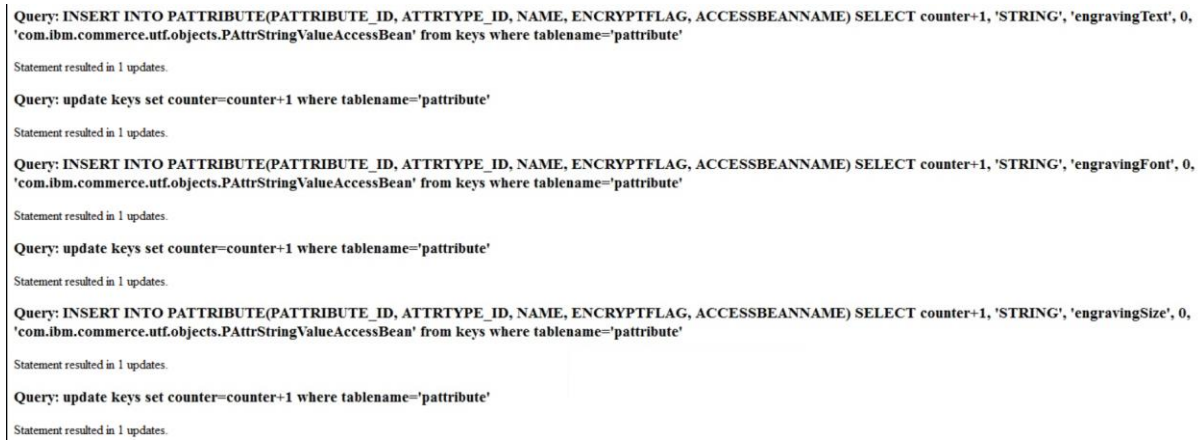
   https://localhost/webapp/wcs/admin/servlet/db.jsp

3. To create text, size and font attributes, run the following SQL statement:

```
INSERT INTO PATTRIBUTE(PATTRIBUTE_ID, ATTRTYPE_ID, NAME,
ENCRYPTFLAG, ACCESSBEANNAME)

SELECT counter+1, 'STRING', 'engravingText', 0,
'com.ibm.commerce.utf.objects.PAttrStringValueAccessBean' from keys
where tablename='pattribute';

update keys set counter=counter+1 where tablename='pattribute';

INSERT INTO PATTRIBUTE(PATTRIBUTE_ID, ATTRTYPE_ID, NAME,
ENCRYPTFLAG, ACCESSBEANNAME)

SELECT counter+1, 'STRING', 'engravingFont', 0,
'com.ibm.commerce.utf.objects.PAttrStringValueAccessBean' from keys
where tablename='pattribute';

update keys set counter=counter+1 where tablename='pattribute';

INSERT INTO PATTRIBUTE(PATTRIBUTE_ID, ATTRTYPE_ID, NAME,
ENCRYPTFLAG, ACCESSBEANNAME)

SELECT counter+1, 'STRING', 'engravingSize', 0,
'com.ibm.commerce.utf.objects.PAttrStringValueAccessBean' from keys
where tablename='pattribute';

update keys set counter=counter+1 where tablename='pattribute';
```

**Note:** The above SQLs are also available in **C:\exercises\services-user-data\PATTRIBUTE.sql**.

4. Verify that the SQLs executed successfully.




5. You must associate the **PATTRIBUTE_ID** of each engraving property with the products that will allow engraving. For the purpose of this exercise, we will assume that the item named **Cristie Truffle Set**, with part number **HTA029_291401**, supports engraving. Run the following SQLs to insert entries into **PATTRPROD** table:

```
INSERT INTO PATTRPROD(PATTRIBUTE_ID, CATENTRY_ID) VALUES ((SELECT
PATTRIBUTE_ID FROM PATTRIBUTE WHERE NAME='engravingText'), (select
catentry_id from catentry where partnumber='HTA029_291401'));

INSERT INTO PATTRPROD(PATTRIBUTE_ID, CATENTRY_ID) VALUES ((SELECT
PATTRIBUTE_ID FROM PATTRIBUTE WHERE NAME='engravingFont'), (select
catentry_id from catentry where partnumber='HTA029_291401'));

INSERT INTO PATTRPROD(PATTRIBUTE_ID, CATENTRY_ID) VALUES ((SELECT
PATTRIBUTE_ID FROM PATTRIBUTE WHERE NAME='engravingSize'), (select
catentry_id from catentry where partnumber='HTA029_291401'));
```

**Note:** The above SQLs are also available in **C:\exercises\services-user-data\PATTRPROD.sql**.

## Part 2: Customizing the business logic

In this step, you will implement the business logic to support the new engraving attributes.

1. Open WebSphere Commerce Developer.
2. In the Java EE perspective, Enterprise Explorer view, expand **WebSphereCommerceServerExtensionsLogic > src**.
3. Create a package:
    - Right-click **src** and select **New** > **Package**.
    - In the name field type **com.mycompany.commerce.customization.order**.
    - Click **Finish**.
4. Using Windows File Explorer, copy the following Java files from **C:\exercises\services-user-data** to **C:\IBM\WCDE80GM\workspace\WebSphereCommerceServerExtensionsLogic\src\com\mycompany\commerce\customization\order**:
    - **MyExtendOrderItemProcessCmdImpl**
    - **MyComposeOrderDetailsCmdImpl**
5. In RAD, right click **WebSphereCommerceServerExtensionsLogic** and click **Refresh**.
6. Clean and rebuild your entire workspace by selecting **Projects** in WebSphere Commerce Developer. Then select **Clean**, ensure that **Clean all projects** is selected and click **OK**. Ensure that there are no build errors.

## Step 3: Updating the order shopping flow to include your customized code

In this step, you update the order shopping flow to include the customized code.

1. Register your new commands
    a. Start WebSphere Commerce test server
    b. Open a browser and enter the following URL:

        http://localhost/webapp/wcs/admin/servlet/db.jsp

    c. In the text area enter the following SQL statement to register the new commands:

```
INSERT INTO CMDREG(STOREENT_ID, INTERFACENAME, CLASSNAME,
TARGET) VALUES (0,
'com.ibm.commerce.orderitems.commands.ExtendOrderItemProces
sCmd',
'com.mycompany.commerce.customization.order.MyExtendOrderIt
emProcessCmdImpl','Local');

UPDATE CMDREG SET CLASSNAME =
'com.mycompany.commerce.customization.order.MyComposeOrderD
etailsCmdImpl' WHERE INTERFACENAME =
'com.ibm.commerce.order.facade.server.commands.ComposeOrder
Cmd+IBM_Details';
```

**Note:** The above SQLs are also available in **C:\exercises\services-user-data\CMDREG.sql**.

Verify that the SQLs execute without any errors.

Enter SQL statements then click **Submit Query**. Terminate all SQL statements with a semi-colon (;)

```
INSERT INTO CMDREG(STOREENT_ID, INTERFACENAME, CLASSNAME, TARGET) VALUES (0,
'com.ibm.commerce.orderitems.commands.ExtendOrderItemProcessCmd',
'com.mycompany.commerce.customization.order.MyExtendOrderItemProcessCmdImpl','Local');

UPDATE CMDREG SET CLASSNAME =
'com.mycompany.commerce.customization.order.MyComposeOrderDetailsCmdImpl' WHERE INTERFACENAME =
'com.ibm.commerce.order.facade.server.commands.ComposeOrderCmd+IBM_Details';
```

Submit Query  Clear All

**Query: INSERT INTO CMDREG(STOREENT_ID, INTERFACENAME, CLASSNAME, TARGET) V**
**'com.ibm.commerce.orderitems.commands.ExtendOrderItemProcessCmd', 'com.mycompany.commerce**

Statement resulted in 1 updates.

**Query: UPDATE CMDREG SET CLASSNAME = 'com.mycompany.commerce.customization.order.My**
**'com.ibm.commerce.order.facade.server.commands.ComposeOrderCmd+IBM_Details'**

Statement resulted in 1 updates.

## Step 4: Customizing the Aurora store to show engraving attributes

This section explains how to customize the Aurora store to show the engraving attributes.

1. Update shopping cart JSP to show the engraving information.
    a) In the Enterprise Explorer view, navigate to **Stores** > **WebContent** > **AuroraStorefrontAssetStore** > **Snippets** > **Order** > **Cart**.
    b) Open **OrderItemDetail.jsp** and locate the following code:

```
<%-- row to display product level discount --%>
<c:if test="${!empty orderItem.adjustment}">
```

    c) Copy the code snippet from **C:\exercises\services-user-data\JSPSnippet.txt** and paste it into **OrderItemDetail.jsp** directly above the code you located in the previous step.

---

**Note:** You can also use the completed **OrderItemDetail.jsp** available in **C:\exercises\services-user-data** for reference. If you copy the JSP into the workspace using Windows File Explorer, make sure you refresh the **Stores** project in RAD, so that the changes are detected and built.

---

**Important:** If you are performing this exercise in your own environment, please note that the **OrderItemDetail.jsp** provided with this exercise may be of a different version than the one in your environment. This may happen due to version differences, for example, a different fix pack or mod pack. In such a case, it is recommended to compare the original file with the file provided with the exercise and then copy only the changes that are relevant to the exercise.

---

    d) Save and close the file.

2. Customize JavaScript logic to send engraving parameters to server.
    a) In Java EE perspective, Enterprise Explorer, navigate to **Stores > WebContent > AuroraStorefrontAssetStore > javascript > CheckoutArea**.
    b) Open **CheckoutHelper.js** and add a comma to the end of the last function, before the last bracket:

```
    toggleOrderItemDetailsSummary: function(orderItemDetailsDiv, contextId, beginIndex,
        var div = dojo.byId(orderItemDetailsDiv);
        if (div.style.display == "none") {
            if (!dojo.hasClass(div, 'orderRetrieved')) {
                cursor_wait();
                wc.render.updateContext(contextId, {'beginIndex': '', 'pageSize': pageS:
                dojo.addClass(div, 'orderRetrieved');
            }
        }
        this._toggleOrderItemDetails(div);
    },
}
```

c) Add a new function **updateCartEngraving** present in **C:\exercises\services-user-data\JSSnippet.txt** to the end of the file, after the comma that you added in the previous step.

---

**Note:** You can also use the completed **CheckoutHelper.js** available in **C:\exercises\services-user-data** for reference.

---

**Important:** If you are performing this exercise in your own environment, please note that the **CheckoutHelper.js** provided with this exercise may be of a different version than the one in your environment. This may happen due to version differences, for example, a different fix pack or mod pack. In such a case, it is recommended to compare the original file with the file provided with the exercise and then copy only the changes that are relevant to the exercise.

---

```
        this._toggleOrderItemDetails(div);
    },

    updateCartEngraving:function(engravingTextBox,engravingSizeBox,engravingFontBox, orderItemId) {
        // summary: This function updates the Order to include engraving information
        // description: This function updates the Order with new engraving specified
        //              by the user for a particular order item. Text, font and size
        //              are sent to the Client API
        // engravingTextBox: DOM Node representing the engravingText element.
        // engravingSizeBox: DOM Node representing the engravingSize element.
        // engravingFontBox: DOM Node representing the engravingFont element.
        // orderItemId: Number representing the OrderItemId for the Item being updated.
        // assumptions: None.
        // dojo API: None.

        var engravingText = engravingTextBox;
        var engravingSize = engravingSizeBox;
        var engravingFont = engravingFontBox;

        var params = [];
        params.orderId = ".";
        params["storeId"] = this.storeId;
        params["catalogId"] = this.catalogId;
        params["langId"] = this.langId;

        this.updateParamObject(params,"orderItemId",orderItemId,false,-1);
        this.updateParamObject(params,"engravingText",engravingText,false,-1);
        this.updateParamObject(params,"engravingSize",engravingSize,false,-1);
        this.updateParamObject(params,"engravingFont",engravingFont,false,-1);

        cursor_wait();
        wc.service.invoke("AjaxUpdateOrderItem",params);

    }
}
```

d) Save and close the file.

3. Add properties text.
   a) Open the **storetext_v2.properties** file found in **C:\IBM\WCDE80GM\workspace\Stores\src\AuroraStorefrontAssetStore** with a text editor.
   b) Paste the below text at the bottom of the file.

```
#Engravings for Checkout Page
ENGRAVING_TITLE=Engraving Text:
SIZE_PROMPT=Select Size
FONT_PROMPT=Select Font
SIZE_OPTION1=Small
SIZE_OPTION2=Medium
SIZE_OPTION3=Large
FONT_OPTION1=Arial
FONT_OPTION2=Helvetica
FONT_OPTION3=Courier
```

   c) Save and close the file.

4. Update REST template.

a) Open **rest-template-config.xml** available at **C:\IBM\WCDE80GM\workspace\Stores\WebContent\WEB-INF\config\com.ibm.commerce.order**.
b) Locate the **updateOrderItem** method and add a comma after the following line:

```
"xitem_field2" : "$xitem_field2"
```

c) Paste the following lines inside the **orderItem** JSON object, after the comma that you added in the previous step:

```
"xitem_engravingText" : "$engravingText",
"xitem_engravingFont" : "$engravingFont",
"xitem_engravingSize" : "$engravingSize"
```

---

**Note:** You can also use the completed **rest-template-config.xml** available in **C:\exercises\services-user-data** for reference.

---

**Important:** If you are performing this exercise in your own environment, please note that the **rest-template-config.xml** provided with this exercise may be of a different version than the one in your environment. This may happen due to version differences, for example, a different fix pack or mod pack. In such a case, it is recommended to compare the original file with the file provided with the exercise and then copy only the changes that are relevant to the exercise.

---

The resulting code should look like this:

```
<method name="updateOrderItem" httpMethod="PUT" path="@self/update_order_item">
    <template>
        <![CDATA[{
            "orderId" : "$orderId",
            "orderItem" : [
                {
                    "productId" : "$catEntryId",
                    "quantity" : "$quantity",
                    "orderItemId" : "$orderItemId",
                    "contractId" : "$contractId",
                    "partNumber" : "$partNumber",
                    "xitem_field1" : "$xitem_field1",
                    "xitem_field2" : "$xitem_field2",
                    "xitem_engravingText" : "$engravingText",
                    "xitem_engravingFont" : "$engravingFont",
                    "xitem_engravingSize" : "$engravingSize"
                }
            ],
            "x_calculationUsage" : "$calculationUsage",
            "x_calculateOrder" : "$calculateOrder",
            "x_inventoryValidation" : "$inventoryValidation",
            "x_remerge" : "$remerge",
            "x_merge" : "$merge",
            "x_check" : "$check",
            "x_allocate" : "$allocate",
            "x_backorder" : "$backorder",
            "x_reverse" : "$reverse"
        }]]>
    </template>
</method>
```

5. Right click **Stores** and select **Refresh** to refresh the Stores project.

## Step 5: Testing your customization

In this section, we will test the customization and learn how the code works.

1. Restart WebSphere Commerce Test Server. Publish the WC application, if needed.

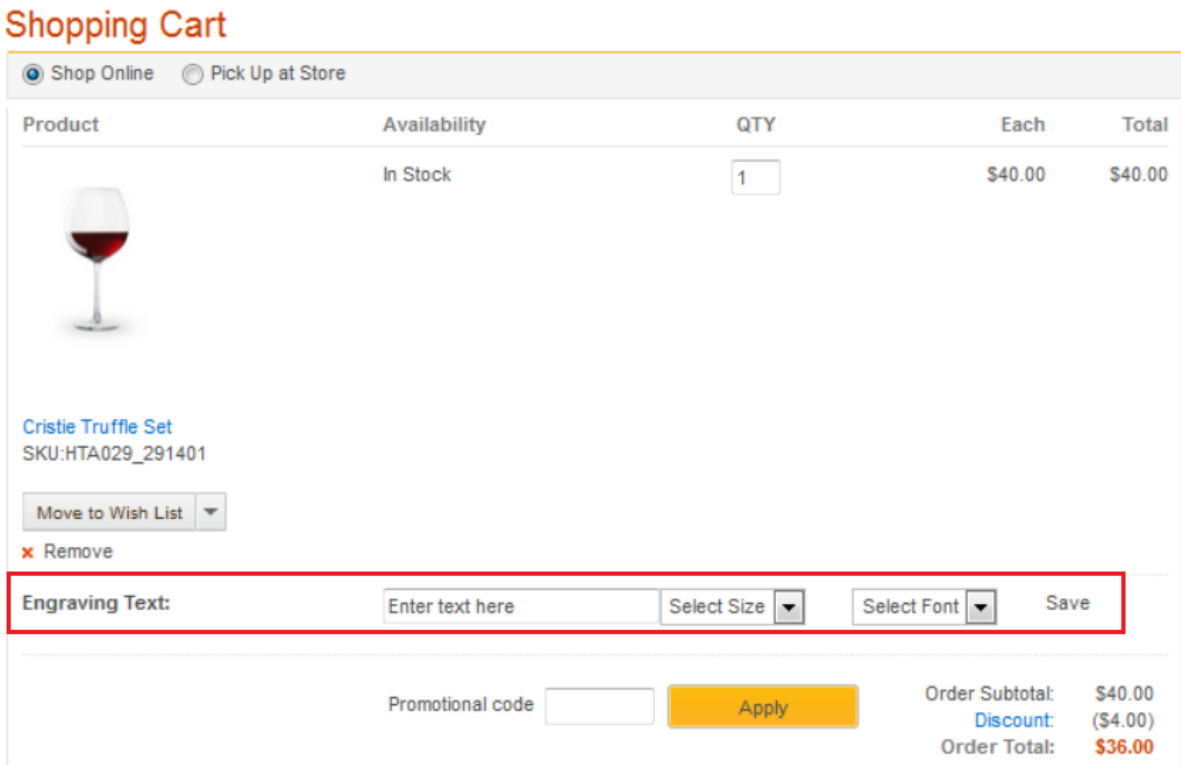
2. Open a new browser window and enter the following URL:

   http://localhost/webapp/wcs/stores/servlet/en/auroraesite/cristie-truffle-set-hta029-291401

   This is the item for which we had configured the engraving attributes in **PATTRPROD** table in a previous step.

3. Click **Add to Cart** to add the item to cart.

4. Go to the shopping cart page. The shopping cart page displays the engraving attributes section:




On the shopping cart page, the cart details are retrieved through the **cart** REST service (**store/{storeId}/cart/@self**). The REST service internally calls the **GetOrder** SOI service with the default access profile – **IBM_Details** [Access profiles used by REST services are defined in **Rest\WebContent\WEB-**

**INF\config\wc-rest-resourceconfig.xml**]. The **GetOrder** service uses **ComposeOrderDetailsCmdImpl** to compose the order BOD response which is passed back to the **cart** REST handler which converts the response BOD into a JSON object which is then returned to the JSP. The mapping between the BOD and the JSON elements is stored in **Rest\WebContent\WEB-INF\config\bodMapping\rest-order-clientobjects.xml**. In particular, the **OrderItem UserData** elements are mapped to JSON parameters starting with '**xitem_**':

```
<_config:URLParameter name="orderItem/xitem_"
nounElement="/OrderItem/UserData/UserDataField" key="false" return="true"
type="UserData" />
```

We have extended **ComposeOrderDetailsCmdImpl** and have implemented business logic to retrieve the engraving attributes using **PATTRIBUTE**, **PATTRPROD** and **PATTRVALUE** tables and add them to the order BOD response.

```
<_wcf:UserDataField name="engravingText">Enter text
here</_wcf:UserDataField>
```

When the shopping cart page is loaded for the first time, the engraving attributes do not exist for the order item. In such a case, the **MyComposeOrderDetailsCmdImpl** command adds 'Enter text here' as engraving text to the BOD response, if the item allows engraving (i.e. the item has an entry in **PATTRPROD**). This allows the JSP (**OrderItemDetail.jsp**) to decide whether the engraving fields need to be displayed or not, depending on whether engraving is supported by the item or not. In the JSP, the **engravingText** UserData field maps to **xitem_engravingText** JSON parameter.

```
<c:when test="${empty orderItem.xitem_engravingText}">

      <%-- This is not an engravable item --%>

</c:when>
```

5. Enter some values and click **Save**. It may take a few seconds to save. The shopping cart widget will refresh once the operation is complete.


When you click **Save**, a JavaScript function **updateCartEngraving** is called. This function calls the **AjaxUpdateOrderItem** Dojo service. This Dojo service is mapped

to the **AjaxRESTOrderItemUpdate** action in **ServicesDeclaration.js (Stores\WebContent\AuroraStorefrontAssetStore\javascript)**.

```
wc.service.declare({

      id: "AjaxUpdateOrderItem",

      actionId: "AjaxUpdateOrderItem",

      url: getAbsoluteURL() + "AjaxRESTOrderItemUpdate",

      formId: ""
```

The **AjaxRESTOrderItemUpdate** action is defined in **Stores\WebContent\WEB-INF\struts-config-order-rest-services.xml**.

```
<action parameter="orderlist.updateOrderItem"
path="/AjaxRESTOrderItemUpdate"
type="com.ibm.commerce.struts.AjaxRESTAction">

      <set-property property="authenticate" value="0:0"/>

      <set-property property="https" value="0:1"/>

</action>
```

The action parameter **orderlist.updateOrderItem** is a REST call from the cart handler (**CartHandler.updateOrderItem**). The JSON request template for this REST call is defined in **Stores\WebContent\WEB-INF\config\com.ibm.commerce.order\rest-config-template.xml**. We have modified this XML to add the engraving parameters to the JSON request so that they are also included in the REST request.

The **CartHandler.updateOrderItem** method calls the **ChangeOrder** SOI service with **Update** action parameter and action expression as **/Order/OrderItem**. The **ChangeOrder** service is mapped to the **OrderItemUpdateCmd** in **WC\xml\messaging\component-services\ChangeOrderSOIBODMapping.xmlf**.

```
<?xml version="1.0" encoding="UTF-8"?>
<TemplateDocument>
    <DocumentType version="*">ChangeOrder</DocumentType>
    <StartElement>ChangeOrder</StartElement>
    <TemplateTagName>ChangeOrderMap</TemplateTagName>
    <CommandMapping>
        <!--  command mapping -->

<Command CommandName="com.ibm.commerce.orderitems.commands.OrderItemUpdateCmd" Condition='actionCode="Update" AND actionExpression="/Order/OrderItem"'>
              <Constant Field="URL">NoURL</Constant>
              <Constant Field="ReUrl">NoReURL</Constant>
              <Constant FieldInfo='CONTROL' Field='flattenMessageClass'>com.ibm.commerce.messaging.programadapter.messagemapper.ecsax.ECSAXOrderMessageFlattener</Constant>
              <Constant FieldInfo='CONTROL' Field='responseCommand'>com.ibm.commerce.order.facade.server.commands.RespondOrderBuildCmdImpl</Constant>
          </Command>
```

The engraving attributes are automatically sent through to the **OrderItemUpdate** command via the REST JSON request and then **ChangeOrder UserData**.

```
<_wcf:UserDataField name="engravingText">Hello</_wcf:UserDataField>

<_wcf:UserDataField name="engravingSize">Medium</_wcf:UserDataField>
```

```
<_wcf:UserDataField name="engravingFont">Helvetica</_wcf:UserDataField>
```

One of the commands that **OrderItemUpdate** calls is **ExtendOrderItemProcessCmd**. We have provided a custom implementation (**MyExtendOrderItemProcessCmdImpl**) for this command which saves the engraving information, that the user entered, in to the **PATTRVALUE** table. Once the **OrderItemUpdate** command completes processing, the shopping cart Dojo widget on the shopping cart page gets refreshed with updated data. The **cart** REST service calls **GetOrder** SOI service and this flow is the same as we described earlier.

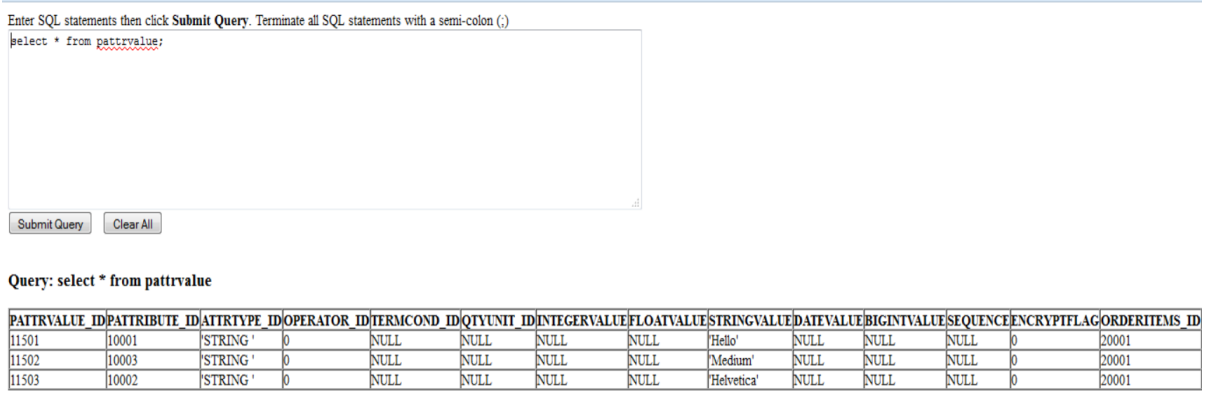
6. Verify that the engraving information has been saved in the database.
    a) Open a new browser window and enter the following URL:

       https://localhost/webapp/wcs/admin/servlet/db.jsp

    b) Enter the following SQL in the text area:

```
select * from PATTRVALUE;
```

Verify that the information has been saved correctly.




| PATTRVALUE_ID | PATTRIBUTE_ID | ATTRTYPE_ID | OPERATOR_ID | TERMCOND_ID | QTYUNIT_ID | INTEGERVALUE | FLOATVALUE | STRINGVALUE | DATEVALUE | BIGINTVALUE | SEQUENCE | ENCRYPTFLAG | ORDERITEMS_ID |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11501 | 10001 | 'STRING ' | 0 | NULL | NULL | NULL | NULL | 'Hello' | NULL | NULL | NULL | 0 | 20001 |
| 11502 | 10003 | 'STRING ' | 0 | NULL | NULL | NULL | NULL | 'Medium' | NULL | NULL | NULL | 0 | 20001 |
| 11503 | 10002 | 'STRING ' | 0 | NULL | NULL | NULL | NULL | 'Helvetica' | NULL | NULL | NULL | 0 | 20001 |

## What you did in this exercise

In this exercise, you learned how to perform the following tasks:

- Extend a noun to add new attributes using UserData.
- Change your store to populate and display the new attributes.
- Test the customization from the previous two tasks.

**End of Exercise**

# Exercise 5: Creating and customizing REST services

## Estimated time

02:30 hours

## What this exercise is about

The objective of this exercise is to demonstrate the development of REST services using the configuration-based controller command and data bean mapping framework.

## What you should be able to do

After completing this exercise, you will be able to:

- Create REST handlers for custom controller commands
- Create REST handlers for custom data beans
- Customize an existing configuration-based data bean mapping to return additional data.
- Use Swagger for REST annotations.

## Introduction

In this exercise, you use the configuration-based controller command and data bean mapping framework to add support for new REST services.

REST services help facilitate the invocation of classic controller commands and the activation of data beans. The REST framework is easy to learn and customize. The framework creates custom REST resource handlers that start controller commands to create, update and delete operations, and activate data beans to retrieve data. To simplify the creation and consumption of REST APIs, WebSphere Commerce provides a utility to create configuration-based controller command and data bean mappings. The utility auto-generates the configuration mapping files and sample handler code for your custom data bean or controller command. It explicitly describes what input is required and what data returns from execution, which reduces the customization effort.

You can create custom handlers to access your own data beans and controller commands easily. Alternatively, you can use the default REST beans and controller commands to return an augmented set of response data to satisfy your customization requirements.

The first part of this tutorial uses a sample data bean and controller command provided with this exercise. From the sample, you will create your own mapping configuration files and REST resource handler to make the REST calls to the sample data bean and controller command. The second part of the tutorial takes the default RequisitionListDataBean as an example, and shows how to customize an existing configuration-based data bean mapping to return more data.

## Requirements

- WebSphere Commerce Developer V8.0.0 Enterprise
- Extended sites store archive published
- AuroraESite store created
- Firebug browser plug-in for Firefox
- Poster browser plug-in for Firefox

## Exercise Instructions

## Part 1: Preparing the data bean and controller command

In this lesson, you create a sample table, database, and controller command. These samples act as a placeholder for your own data beans and controller commands, and serve as an illustration of the REST mapping method.

1. Start WebSphere Commerce Developer.

2. Start the WebSphere Commerce Test Server.

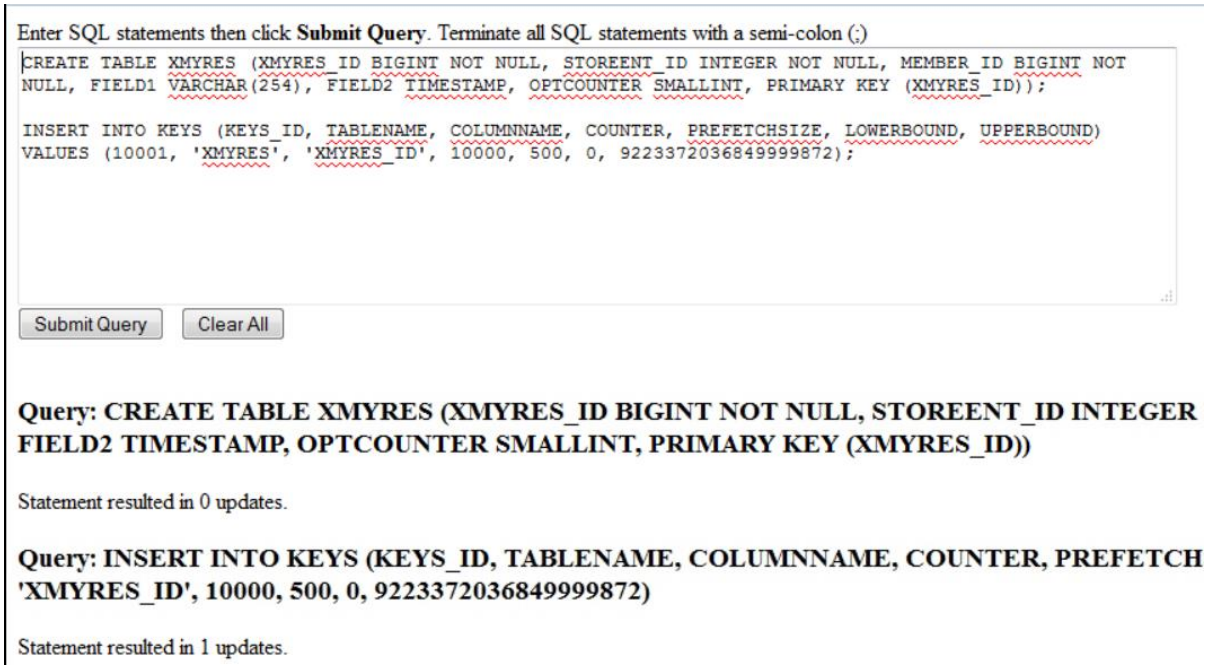
3. Open a web browser and enter the following URL: http://localhost/webapp/wcs/admin/servlet/db.jsp

4. Enter the following SQLs in the text area and click **Submit Query**.

```
CREATE TABLE XMYRES (XMYRES_ID BIGINT NOT NULL, STOREENT_ID
INTEGER NOT NULL, MEMBER_ID BIGINT NOT NULL, FIELD1
VARCHAR(254), FIELD2 TIMESTAMP, OPTCOUNTER SMALLINT, PRIMARY
KEY (XMYRES_ID));

INSERT INTO KEYS (KEYS_ID, TABLENAME, COLUMNNAME, COUNTER,
PREFETCHSIZE, LOWERBOUND, UPPERBOUND) VALUES (10001, 'XMYRES',
'XMYRES_ID', 10000, 500, 0, 9223372036849999872);
```

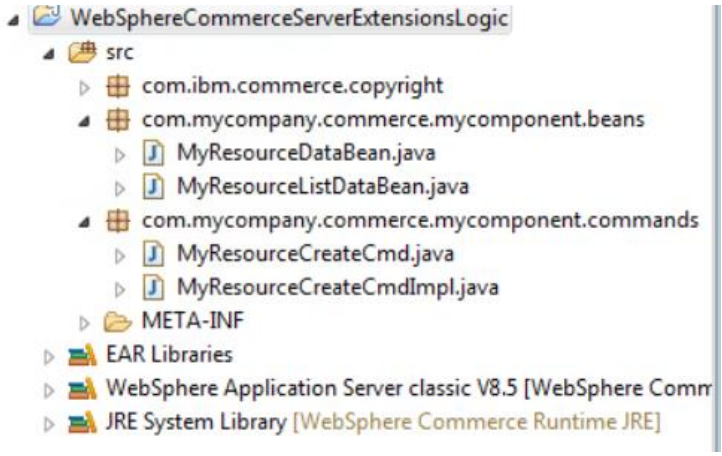
Note: The above SQLs can also be found in **C:\exercises\create-rest-services\MYRES.sql**.

5. Verify that the SQLs execute successfully.




6. In the Java EE perspective, Enterprise Explorer view, expand **WebSphereCommerceServerExtensionsLogic**.
7. Right-click **src** and select **New > Package**.
8. Create a package named **com.mycompany.commerce.mycomponent.beans**.
9. Repeat the previous step to create another package named **com.mycompany.commerce.mycomponent.commands**.
10. In Windows Explorer, navigate to the following directory: **C:\IBM\WCDE80GM\workspace\WebSphereCommerceServerExtensionsLogic\src\com\mycompany\commerce\mycomponent\beans**. Copy **MyResourceDataBean.java** and **MyResourceListDataBean.java** from **C:\exercises\create-rest-services** to this folder.
11. In Windows Explorer, navigate to the following directory: **C:\IBM\WCDE80GM\workspace\WebSphereCommerceServerExtensionsLogic\src\com\mycompany\commerce\mycomponent\commands**. Copy **MyResourceCreateCmd.java** and **MyResourceCreateCmdImpl.java** from **C:\exercises\create-rest-services** to this folder.
12. In Rational Application Developer, right click on **WebSphereCommerceServerExtensionsLogic** and click **Refresh**.

The Java files that you copied in the previous steps should now be visible in the workspace.




Ensure that the classes are compiled and that there are no compilation errors. If needed, you can compile the classes using **Project > Clean**.

---

**Note: MyResourceCreateCmd** provides the default implementation class using the **defaultCommandClassName**.

```
public interface MyResourceCreateCmd extends ControllerCommand {
    public static final String NAME = MyResourceCreateCmd.class.getName();
    public static final String defaultCommandClassName = NAME + "Impl";
```

This means that, for the purposes of this exercise, we do not need to register the implementation class in **CMDREG** table. However, in a real-life scenario, it is highly recommended to register implementation classes in **CMDREG**, and not sure **defaultCommandClassName**. This provides greater flexibility, for example, you can register different implementation classes for the same interface in different stores.

---

13. You can open the data beans and controller command to review the code provided.

    - **MyResourceDataBean** is used to retrieve the data for **field1** and **field2** from the **XMYRES** table, that you created in the database. The **MyResourceListDataBean** contains a list of **MyResourceDataBean** beans.
    - **MyResourceCreateCmd** is used to run actions against the custom table, such as create the data in the custom table **XMYRES**.

## Part 2: Generating Configuration-Based Mapping Files

In this lesson, you use a utility to generate the XML mapping files for your sample data beans and controller command.

WebSphere Commerce uses mapping configuration files to make REST calls to beans and commands. These mapping files are divided into profiles, which use different types of calls for the same bean or command. For example, to specify different output values per profile.

Each configuration file defines the mappings for a single bean or command. The file describes mapping of REST input names to the corresponding bean or command setter methods. For each profile in the file, output mappings are also supplied to describe how the getter-methods from the bean or command must be mapped back to a REST entity output.

WebSphere Commerce provides a utility that is called RESTClassicSampleGen.bat. The utility creates initial mapping configuration files and sample REST resource handler code. The sample code can be used as a base to create your own REST service for your bean or command. The generated code demonstrates best practices, techniques, and conventions to assist with your customization. The utility helps you to generate a complete set of source code package elements.

---

**Note:** The generated files are samples that can be used as a base. Do not expect to use the generated mappings without modifications.

---

1. Open a command prompt and navigate to the following directory: **C:\IBM\WCDE80GM\bin**.

2. Run each of the following command one by one. Ensure that each command runs without errors.

   ---

   **Note:** The commands can also be copied from the following file: **C:\exercises\create-rest-services\REST_Scripts.txt**.

   ---

```
RESTClassicSampleGen.bat
location=com.mycompany.commerce.mycomponent.beans.MyResourceLi
stDataBean outputDir=C:\RESTconfig
additionalClassPath="C:\IBM\WCDE80GM\workspace\WebSphereCommer
ceServerExtensionsLogic\bin"
```


For each execution, you can check the log file **C:\IBM\WCDE80GM\logs\restclassic.log** to verify that the utility completed successfully.

```
RESTClassicSampleGen.bat
location=com.mycompany.commerce.mycomponent.beans.MyResourceDa
taBean outputDir=C:\RESTconfig
additionalClassPath="C:\IBM\WCDE80GM\workspace\WebSphereCommer
ceServerExtensionsLogic\bin"
```


```
RESTClassicSampleGen.bat
location=com.mycompany.commerce.mycomponent.commands.MyResourc
eCreateCmd outputDir=C:\RESTconfig
additionalClassPath="C:\IBM\WCDE80GM\workspace\WebSphereCommer
ceServerExtensionsLogic\bin"
```


Where:

**location** = The fully qualified Java class name of the data bean or controller command, such as com.mycompany.commerce.mycomponent.beans.MyResourceDataBean.

**outputDir =** The output directory. If you do not have an existing one, the output directory is automatically created. Subdirectories are created for each class type generated. In this sample, the **outputDir** is **C:\RESTconfig**.

**additionalClassPath =** specifies extra JAR files and directories to locate classes (and their dependencies). Separate extra JAR files and directories with a semicolon.

The utility generates a set of directories and files for creating your REST service. It contains the following three folders under your outputDir folder:

- **controllerCommandSamples**
- **dataBeanSamples**
- **handlerSamples**

These folders represent the mapping configuration files and handler sample code for **MyResourceDataBean**, **MyResourceListDataBean**, and **MyResourceCreateCmd**.

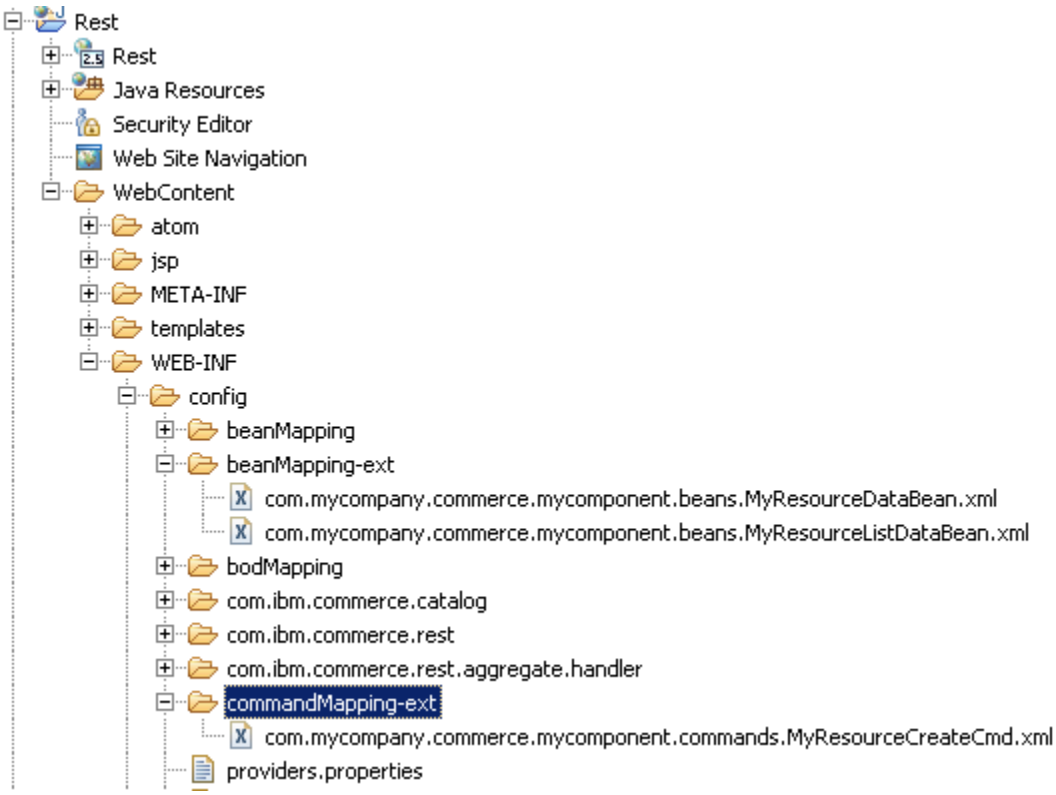
2. In the Enterprise Explorer view, go to **Rest > WebContent > WEB-INF > config**.
3. Create the **beanMapping-ext** and **commandMapping-ext** mapping folders.
    a. Right-click **config**. Click **New** > **Folder**.
    b. In the **Folder name** field, enter **beanMapping-ext**.
    c. Click **Finish**. The **beanMapping-ext** folder is created under the config directory.
    d. Repeat the steps to create the **commandMapping-ext** folder.
4. Copy the sample configuration to the REST configuration folders that you created in the previous step.
    a. Copy **com.mycompany.commerce.mycomponent.beans.MyResourceDataBean.xml** and **com.mycompany.commerce.mycomponent.beans.MyResourceListDataBean.xml** from **C:\RESTconfig\dataBeanSamples** to the **beanMapping-ext** folder.
    b. Copy **com.mycompany.commerce.mycomponent.commands.MyResourceCreateCmd.xml** from **C:\RESTconfig\controllerCommandSamples** to the **commandMapping-ext** folder.

---

**Note:** If you were unable to generate the configuration files using the utility due to errors, and would like to continue with the exercise without spending time in debugging the errors, you can use the files provided in **C:\exercises\create-rest-services**.

---

Your file structure should resemble the following image:




5. Review the sample mapping configuration files.
    a. Go to **Rest** > **WebContent** > **WEB-INF** > **config** > **beanMapping-ext**.
    b. Open the following file: **com.mycompany.commerce.mycomponent.beans.MyResourceDataBean.xml**.

       The following code displays:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 ================================================================================
  The sample contained herein is provided to you "AS IS".

  It is furnished by IBM as a simple example and has not been thoroughly tested
  under all conditions.  IBM, therefore, cannot guarantee its reliability,
  serviceability or functionality.

  This sample may include the names of individuals, companies, brands and products
  in order to illustrate concepts as completely as possible.  All of these names
  are fictitious and any similarity to the names and addresses used by actual persons
  or business enterprises is entirely coincidental.
 ================================================================================
--><bean>
    <profiles>
        <profile name="sample">
            <inputs>
                <input inputName="myResourceId" methodName="setMyResourceId"/>
            </inputs>
            <outputs>
                <output methodName="getField1" outputName="field1"/>
                <output methodName="getField2" outputName="field2"/>
                <output methodName="getMyResourceId" outputName="myResourceId"/>
            </outputs>
        </profile>
    </profiles>
</bean>
```

c. Change the profile name from **sample** to **MyCompany_Databean_Summary**. If you copied the file from **C:\exercises\create-rest-services**, then it will already have the correct profile name.

d. Open the **com.mycompany.commerce.mycomponent.beans.MyResourceListDataBean.xml** file. Change the profile name from **sample** to **MyCompany_DatabeanList_Summary**.

e. Go to **Rest** > **WebContent** > **WEB-INF** > **config** > **commandMapping-ext**.

f. Open the **com.mycompany.commerce.mycomponent.commands.MyResourceCreateCmd.xml** file. Change the profile name from **sample** to **MyCommand_Summary**.

## Part 3: Creating custom REST resource handler to perform mapping

In this lesson, you learn how to create your own REST resource handler for your sample data bean and controller command. A REST call requires the creation of a resource handler. The resource handler represents the entry point for resource requests and is annotated with the @Path, context, and other information that is required to handle a request. Handlers are responsible for coordinating the client request. Handlers intercept calls, run required actions, and convert responses to a form consumable by the client by using standard HTTP protocol elements.

With the configuration-based command and bean mapping framework, a REST call is made to access a data bean (GET) or a controller command (POST). The handler is relatively small, and required to define the annotations for Apache Wink to understand how to make the REST call to the handler and supply the requested information.

1. In File Explorer, navigate to the **C:\RESTconfig\handlerSamples** directory. Review the sample handler code that is generated by the utility for your data beans and controller command.

    a. Three separate handlers have been generated. One for each data bean and controller command.

        i. MyResourceCreateCmdHandler

        ii. MyResourceDataBeanHandler

        iii. MyResourceDataBeanListHandler

    b. Custom resource handlers, that call a data bean or a controller command, should always extend from **AbstractConfigBasedClassicHandler**.

    c. The data bean handler has a single method **processGetRequest** which calls the method **executeConfigBasedBeanWithContext** to activate the data bean.

    d. The command handler has a single method **processPostRequest** which calls the method **executeConfigBasedCommandWithContext** to execute the controller command.

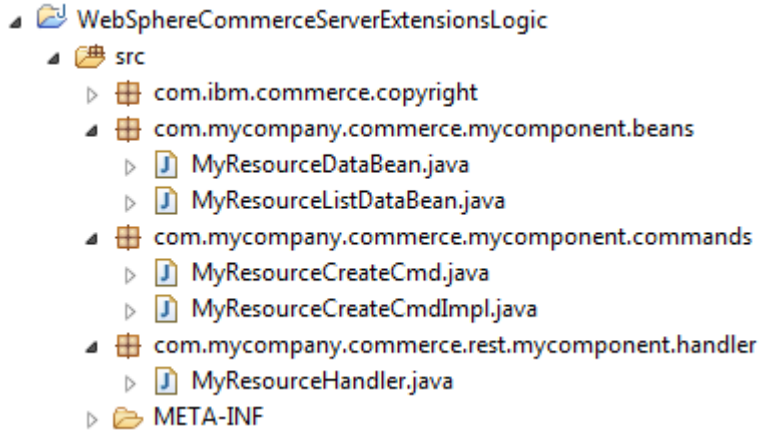
In our case, we will create a single resource handler to call the data beans and command.

2. In Enterprise Explorer view, browse to **WebSphereCommerceServerExtensionsLogic > src**. Right click **src** and select **New > Package**.

3. In the **Name** field, enter **com.mycompany.commerce.rest.mycomponent.handler**. Click **Finish**.

4. In Windows Explorer, navigate to the following directory: **C:\IBM\WCDE80GM\workspace\WebSphereCommerceServerExtensionsLogic\src\com\mycompany\commerce\rest\mycomponent\handler**. Copy **MyResourceHandler.java** from **C:\exercises\create-rest-services** to this folder.

5. In Rational Application Developer, right click on **WebSphereCommerceServerExtensionsLogic** and click **Refresh**.




6. Open **MyResourceHandler.java** and review the code.

   The following methods are implemented:

   **findByMyResourceId**

   This method runs **MyResourceDataBean** based on the input and the configuration mapping profile **MyCompany_Databean_Summary**. It returns the resource data from the **XMYRES** table according to the **myResourceId** you specify. The input parameters for the method are **storeId**, **myResourceId**, and **responseFormat**. The resource path is **store/{storeId}/my_resource/{myResourceId}**.

   **findMyResourceList**

   This method runs **MyResourceListDataBean** based on the input and the configuration mapping profile **MyCompany_DatabeanList_Summary**. It

returns a list of resource data in the form of an array of **MyResourceDataBean** objects. The input parameters for the method are **storeId** and **responseFormat**. The data bean **MyResourceListDataBean** uses the current user's id from the command context to retrieve the list of resources created by the current user. The resource path is **store/{storeId}/my_resource/myResourceListDataBean**.

**create**

This method runs **MyResourceCreateCmd** based on the input and the configuration mapping profile **MyCommand_Summary**. The command inserts new data in to the **XMYRES** table. The input parameters for the REST service are **storeId** and **responseFormat**. The data to be inserted into the table is sent in the body of the POST request. The resource path is **store/{storeId}/my_resource**.

You also can use GET methods to implement queries, specifically find by primary key, find by identifier/name, and find by query. Implement queries by using find by query with internal GET methods. Avoid using GET methods with custom paths as they must be manually created, since they are not generated by the utility.

**initializeQueryRegistry**

This method initializes the new Query registry.

**findByQuery**

This method is to implement the GET method. The input parameter is **storeId** and the resource path is **store/{storeId}/resource_name**, such as **/{storeId}/my_resource**. It uses the query registry framework to map query names to internal GET methods per the REST API Discoverability.

**findSelf**

This method performs the same function as **findMyResourceList**. The resource path of this method is **store/{storeId}/ my_resource?q=self**.

7. Register **MyResourceHandler.java.**
    a. In the Enterprise Explorer view, browse to the **Rest** > **WebContent** > **WEB-INF** > **config** > **resources-ext.properties** file.
    b. Define the new resource handler that you created:

```
com.mycompany.commerce.rest.mycomponent.handler.MyResourceHandler
```

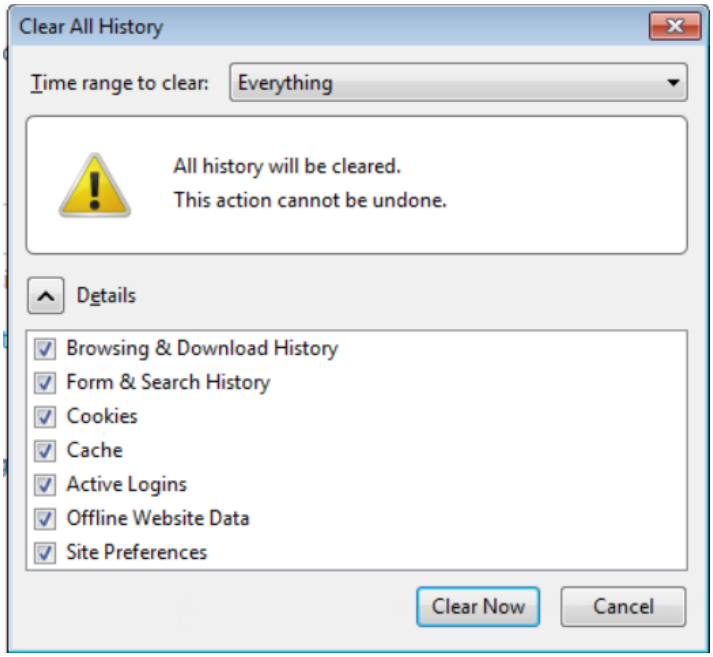
The file will resemble the following screenshot:


c. Save and close the file.

## Part 4: Verifying new REST service call by using the Poster browser plug-in

In this lesson, you verify the new REST calls to the sample data beans and controller commands with the Poster browser plug-in for Firefox.

1. Restart the WebSphere Commerce Test Server. Republish the WC application.
2. Open the Firefox browser. Clear out all cookies and session information, using **Tools > Options > Privacy > clear your recent history**. Clear everything.



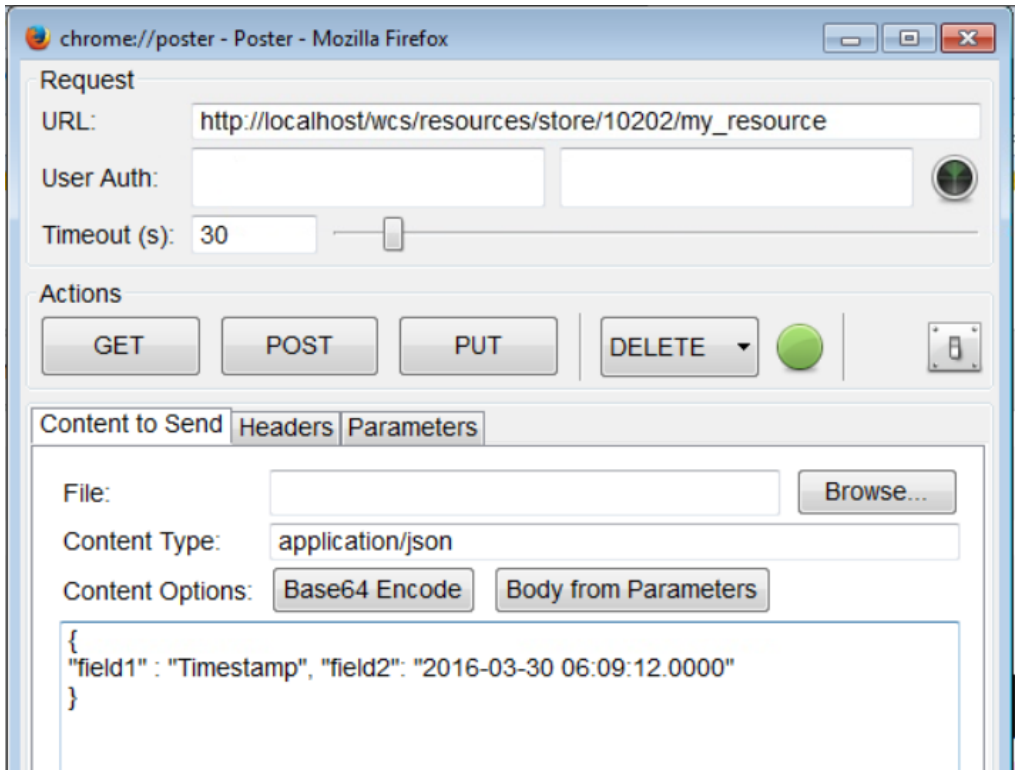

3. Close the browser and open a new Firefox window.
4. Enter the following URL to access the Aurora B2B store:

   http://localhost/webapp/wcs/stores/servlet/en/aurorab2besite

5. Click **Sign In/Register**. Log in to the store as a Site Administrator. For example, **wcsadmin**.
6. Launch Poster for Firefox using **Tools > Poster**.
7. Create data to post into the **XMYRES** table.
   a) In the **URL** field, enter
   http://localhost/wcs/resources/store/10202/my_resource

**Note: 10202** is the store id of **AuroraB2BESite**. If you are using a different environment, this value may be different. You can verify the value by querying the **STOREENT** table.

b) Set **Content Type** as **application/json**.
c) Specify values for field1 and field2 in the body of the **Content to Send** pane:
{ "field1": "Timestamp", "field2": "2016-03-30 06:09:12.0000" }




7. Click **POST** to create the resource data in the table. If the new data is created successfully, the corresponding **myResourceId** is displayed in the response window. For example:

POST on http://localhost/wcs/resources/store/10202/my_resource

Status: 200 OK

{ “myResourceId” : 10001 }


**Troubleshooting:** If you get an “invalid profile” error in response, verify that the profile name given in **com.mycompany.commerce.mycomponent.commands.MyResourceCreateCmd.xml** in **Rest** > **WebContent** > **WEB-INF** > **config** > **commandMapping-ext** is correct. It should be **MyCommand_Summary**.

8. Create more entries in the **XMYRES** table to test. Post the following field data by using Poster:

```
{ "field1": "Timestamp", "field2": "2016-03-30 08:53:19.0000" }
{ "field1": "Timestamp", "field2": "2016-03-30 11:54:22.0000" }
{ "field1": "Timestamp", "field2": "2016-03-30 12:18:02.0000" }
```

   You need to enter the above rows in Poster one by one.

9. Test the **findByMyResourceId** method. In the **URL** field, change the URL to:

   http://localhost/wcs/resources/store/10202/my_resource/10001

10. Within the **Content to Send** pane, remove the values that you specified for **field1** and **field2**.

11. Click **GET**. Using the REST service, the resource data is retrieved through the resource ID number 10001. The response window with the corresponding information displays.

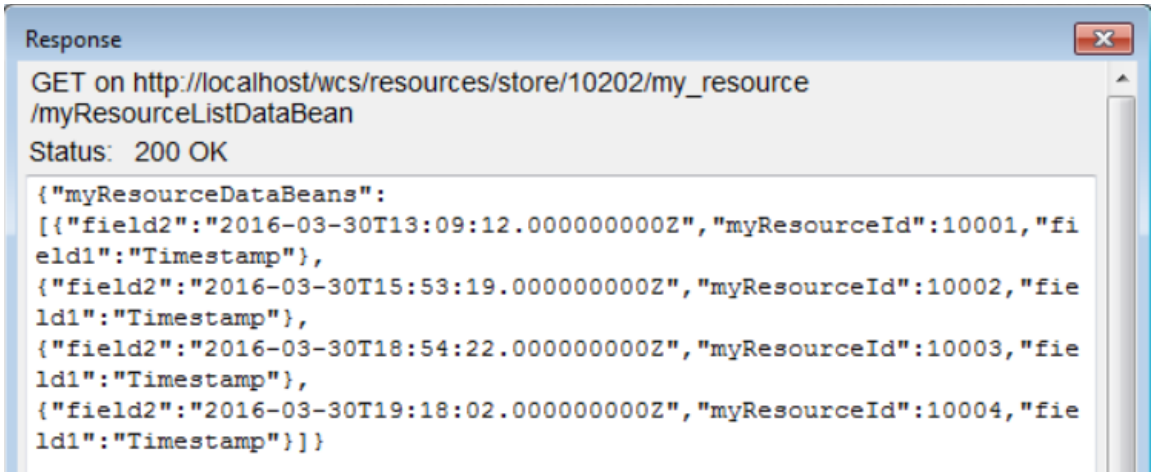
    For example:

```
GET on https://localhost/wcs/resources/store/10202/my_resource/10001
Status: 200 OK

{"myResourceId":10001,"field1":"Timestamp","field2":"2016-03-30T13:09:12.000000000Z"}
```

12. Test the **findMyResourceList** method. In the **URL** field, change the URL to:

http://localhost/wcs/resources/store/10202/my_resource/myResourceListDataBean

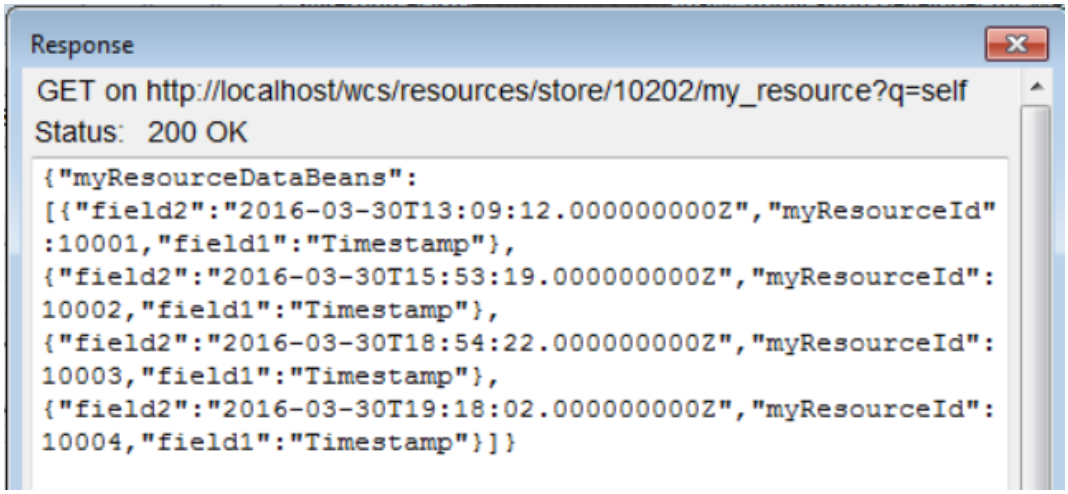
13. Click **GET**. The list of resource data is retrieved by using the REST service. The response window with the corresponding information is shown.




14. Test the **findSelf** query method. In the **URL** field, change the URL to:

    http://localhost/wcs/resources/store/10202/my_resource?q=self

15. Click **GET**. The list of resource data is retrieved by using the REST service. The response window with the corresponding information is shown.




16. Finally, we can verify the data in the XMYRES table by querying the database.

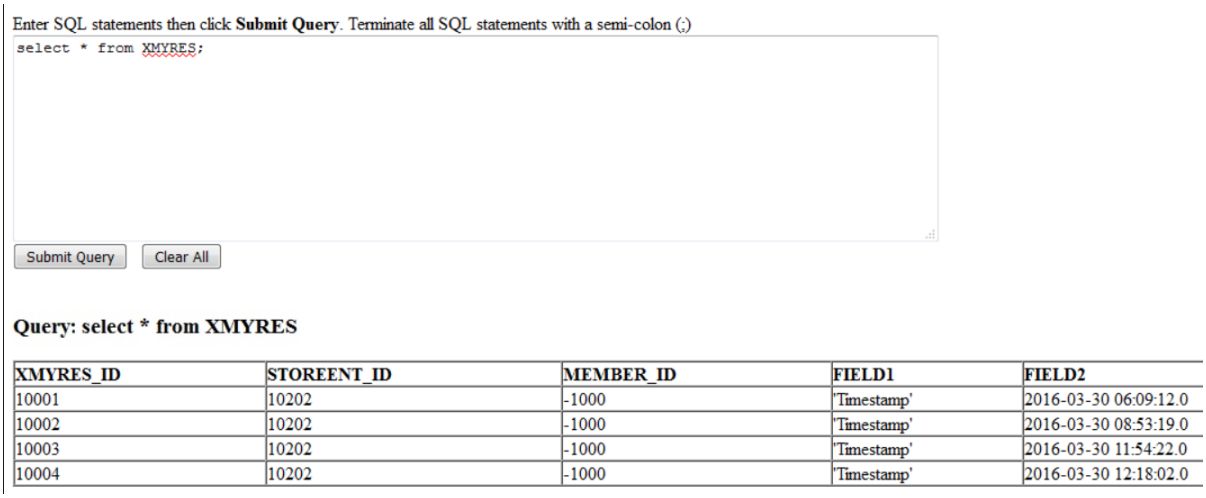
    a) Enter the following URL in the browser:

       http://localhost/webapp/wcs/admin/servlet/db.jsp

    b) In the text area, enter the following query and click **Submit Query**.

```
select * from XMYRES;
```

The data from the table is returned.




**Note:** If no Coordinated Universal Time relation information is given with a time representation, the time is in local time. It is ambiguous in communicating across different time zones. The Timestamp value you input through Poster is processed by the configuration-based command and data bean mapping framework. All formats are ISO 8601 compliant and everything ends in 'Z' except an SQL Date. A pure date value has no time zone.

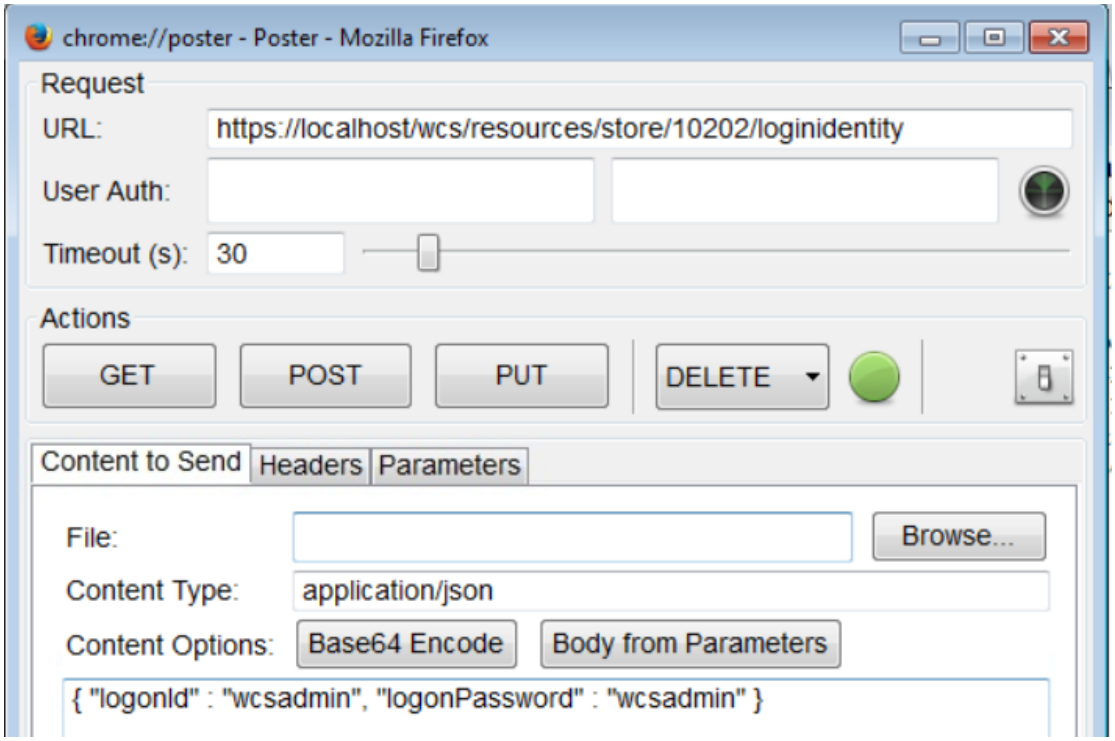
**Note:** The REST APIs inherit the access control of the underlying commands and data beans. Since this exercise does not cover setting up of access control policies for the sample command and data beans, you must sign in as **wcsadmin** first.

In this exercise, we have logged in to the Aurora B2B store as **wcsadmin**, before executing the REST calls. Alternatively, you can log in using the **loginidentity** REST service which returns a **WCToken** and a **WCTrustedToken** value.

https://localhost/wcs/resources/store/10202/loginidentity

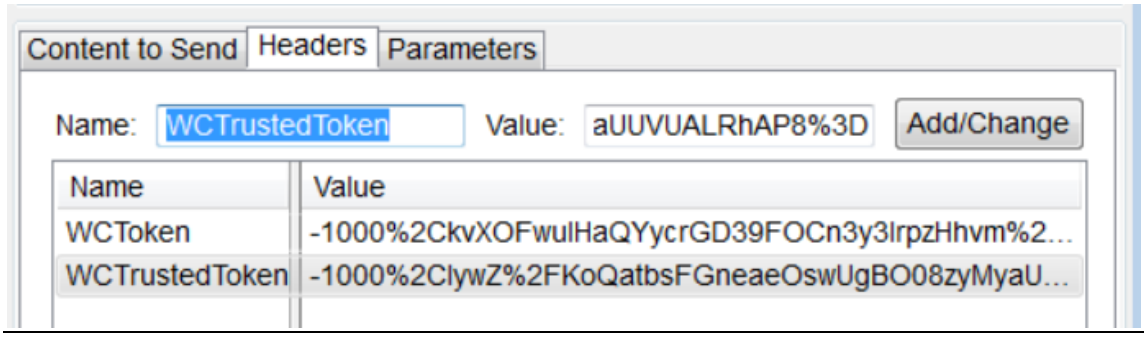
Request:

```
{ "logonId" : "wcsadmin", "logonPassword" : "wcsadmin" }
```




Response:


These values should then be included in the **Headers** pane in **Poster** in the subsequent REST calls.




**Troubleshooting:** If you get invalid cookie or authorization errors, log in and out of the store multiple times. If necessary, log in to the store, then authenticate by using the **loginidentity** service in Poster:

https://localhost/wcs/resources/store/10202/loginidentity

and interact with the storefront to error out the browser user login. Then, try posting the content again.

If you are still having problems with testing - Login into poster using **loginidentity**, and from the Firefox browser logon to the store as **wcsadmin**:

https://localhost/webapp/wcs/stores/servlet/en/aurorab2besite

Now, do not logout from the store and continue the next steps without having to enter the **WCToken** and **WCTrustedToken**.

## Part 5: Customizing an existing configuration-based data bean mapping to return more data

In this lesson, you customize an existing configuration-based data bean mapping to return extra data. By default, WebSphere Commerce uses REST calls for some beans and commands. Configured with mapping profiles, REST calls map REST input parameters to bean or command setter-methods, and bean/command getter-methods to REST output parameters. In some situations, customers might want to use the default REST beans or commands, but want an augmented set of response data. The configuration-based command and data bean mapping framework allows customizations where one REST mapping profile can extend another. For example, **RequisitionListDataBean** can be customized to return extra data.

A requisition list is a list of items that are used to create future orders. Users can create requisition lists of items they frequently order, and use the list to place reorders. The underlying implementation of a requisition list is an order.
The **STATUS** field in the **ORDERS** table is used to determine whether an order is a requisition list. Each item on a requisition list is associated with a catalog entry and has the following attributes: Quantity, Last update time, Owner, Store, and Type (private or shared). The **RequisitionDataBean** retrieves the data for the requisition list item, while **RequisitionListDataBean** is the object that contains a list of **RequisitionDataBeans**. **RequisitionDataBeans** can retrieve the list of data for the Requisition list items.

1. In the Java EE perspective, Enterprise Explorer view, go to **Rest** > **WebContent** > **WEB-INF** > **config** > **beanMapping**.
2. Select and open the **com.ibm.commerce.order.beans.RequisitionListDataBean.xml** file.

   Review the code. You will find that the profile called **IBM_Store_Summary** is used. By default, it returns set of REST outputs, such as comment, description, and lastUpdate. **RequisitionDataBean** can return more data than what is included in the XML by default. For more information, see RequisitionDataBean.

   Next, we will extend the default profile for the bean to return more data.

3. Go to **Rest** > **WebContent** > **WEB-INF** > **config** > **beanMapping-ext**.
4. Copy the **com.ibm.commerce.order.beans.RequisitionListDataBean.xml** file from the **beanMapping** folder to the **beanMapping-ext** folder.
5. Replace the original code with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>

<bean>
   <profiles>
<!-- Profile extends the IBM_Store_Summary profile.
It will look for IBM_Store_Summary first in the extension file, and
subsequently the base if not found. -->
<profile name="MyCompany_Store_Summary">
<extends>
   <extend profileName="IBM_Store_Summary" />
</extends>
<outputs>
<!-- Extend a nested output element for getRequisitionDataBeans. -->
     <output methodName="getRequisitionDataBeans"
outputName="resultList">
            <output methodName="getTotalTax" outputName="totalTax" />
            <output methodName="getEstimatedShipDate"
outputName="shipDate" />
            <output methodName="getOwner" outputName="owner" />
    </output>
   </outputs>
</profile>
</profiles>
</bean>
```

The custom **MyCompany_Store_Summary** profile extends from the default **IBM_Store_Summary**. The custom profile replaces the default **IBM_Store_Summary** profile. The output parameters defined by the custom profile and the default profile are merged together and included in the REST response.

6. Save the file.
7. Restart the WebSphere Commerce Test Server.

## Part 6: Verifying customized REST service call using the Poster browser plug-in

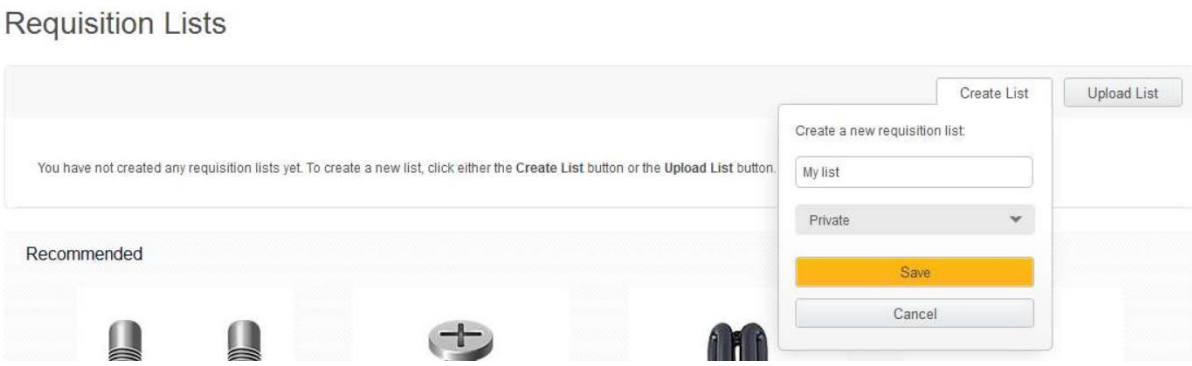
In this lesson, you verify the customized REST call to the **RequisitionListDataBean** using Poster.

First, we will create a requisition list that we can use for testing.

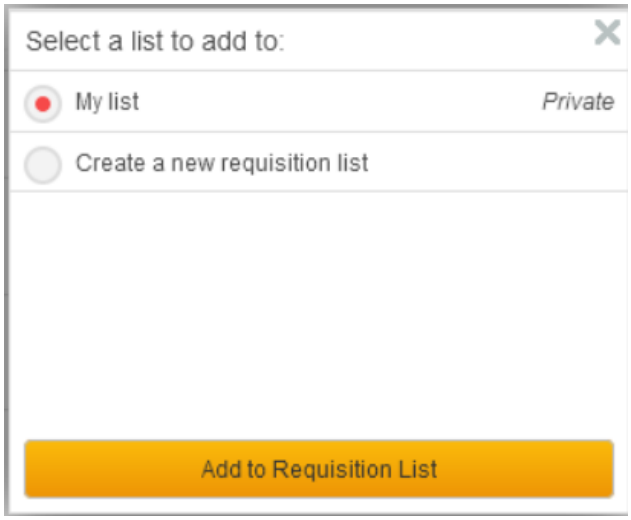
1. Open the Aurora B2B starter store in Firefox:

https://localhost/webapp/wcs/stores/servlet/en/aurorab2besite

2. Click **Sign In/Register** and log in to the store as a Site Administrator. For example, wcsadmin.
3. Click **My Account** to load the My Account page.
4. On the left pane, click **Requisition Lists** tab under the **Orders** section.
5. On the right side, click **Create List**. A small window pane opens. Type in a name and select a type (shared/private) to create a requisition list.

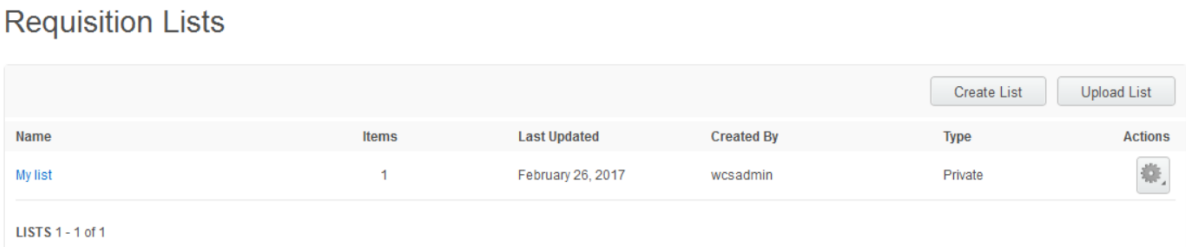



6. Save your new data. A message window displays the successfully created requisition list. Close the prompt.
7. Select a product under the **Recommended** section. Go to the Product Details page.
8. Enter a quantity for the product. Click **Add to Requisition List**.
9. Select the requisition list you just created and click **Add to Requisition List**.




10. Select **Continue Shopping** and go back to the **Requisition Lists** page.

    The page shows the requisition list in the table:




Stay logged in to the store. Next, we will test the customized REST service call using Poster.

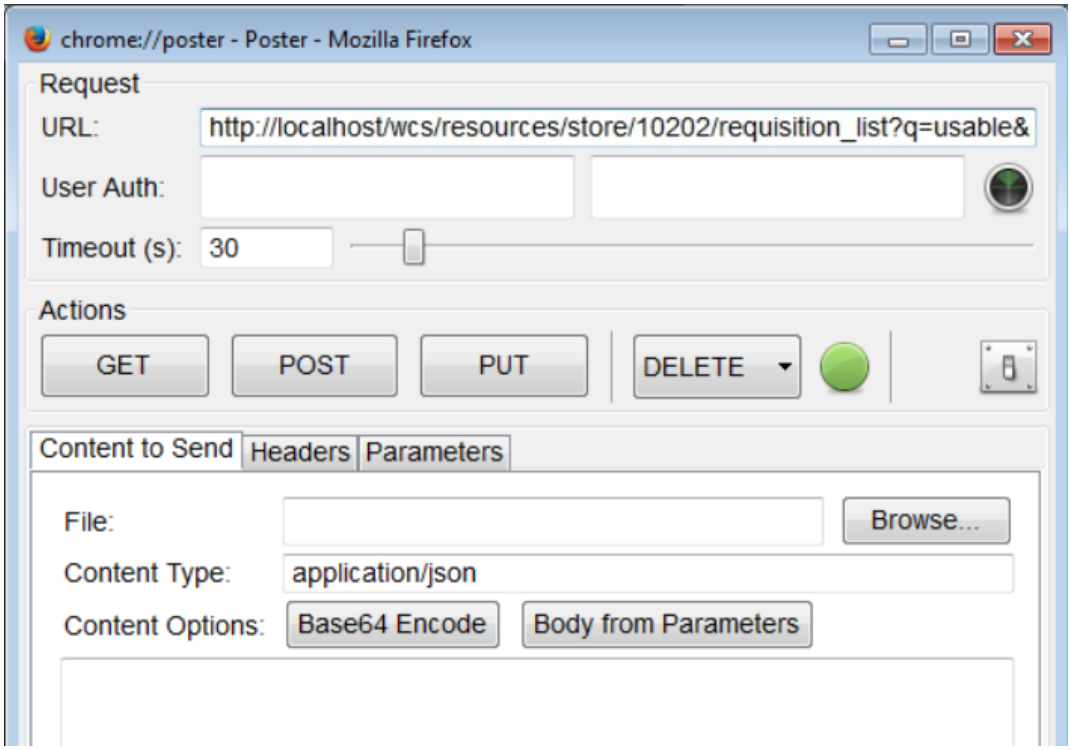
1. Launch Poster for Firefox.

2. In the **URL** field, enter the URL for the GET action to retrieve the data from the requisition list:

http://localhost/wcs/resources/store/10202/requisition_list?q=usable&profileName=MyCompany_Store_Summary

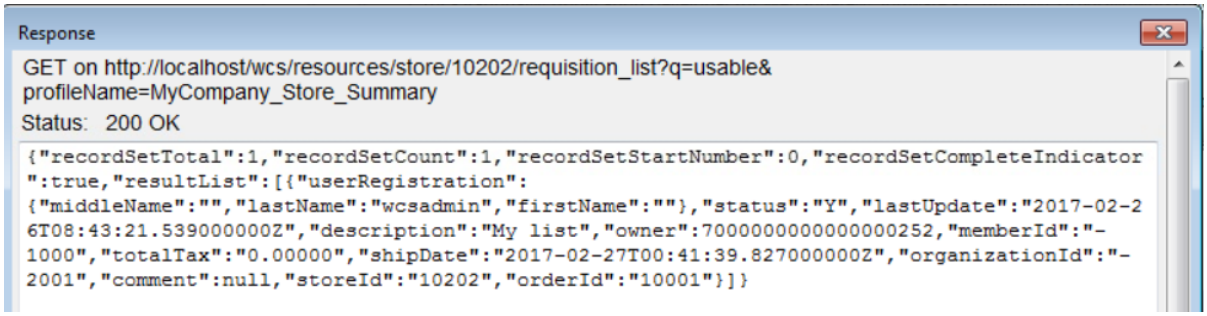
Here we have specified the custom profile in the URL. If we do not include any profile in the URL, then the default profile will be used.

3. Set **Content Type** as **application/json**.




4. Click **GET**.

The REST service retrieves the corresponding data. The response window with the corresponding information displays.




Review the results, noting the information is available within it for the first requisition list item. Extra data such as owner, totalTax, and shipdate is also retrieved successfully.

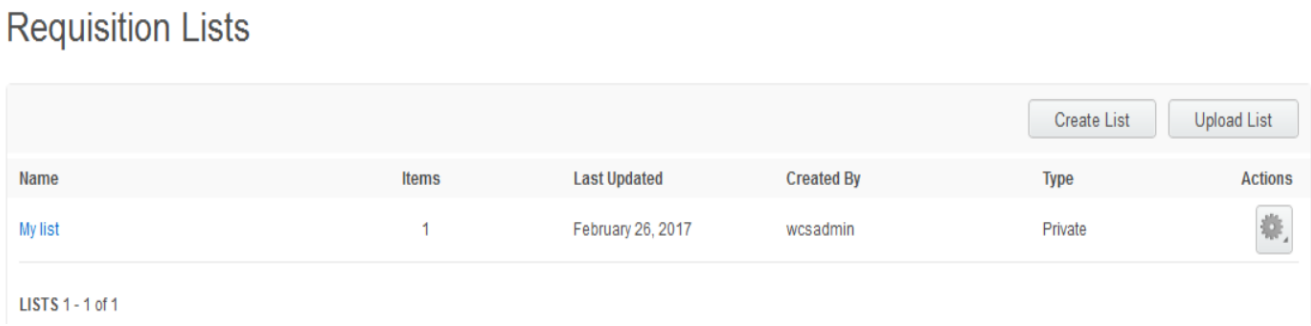
Next, customize the store widget to use the additional data.

## Part 7: Customizing the store widget to use the additional data

In this lesson, you learn how to customize the store widget to use the new data that is returned by the custom profile that you created in the previous lesson.

1. Examine the storefront by using Firebug.
    a. Open the Mozilla Firefox browser.
    b. Open the Aurora B2B starter store in a web browser:

       https://localhost/webapp/wcs/stores/servlet/en/aurorab2besite
    c. Click **Sign In/Register** and log in to the store as a Site Administrator. For example, wcsadmin.
    d. Click **My Account** to load the My Account page.
    e. On the left pane, click the **Requisition List** tab under the **Orders** section. The values in the Requisition list columns are returned by the default **IBM_Store_Summary** profile by using REST service calls.

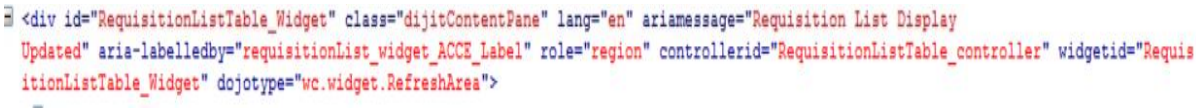



You can now replace the default table column. For example, we will replace **Last Updated**, with new custom data such as **Owner**, which is returned by the **MyCompany_Store_Summary** profile.

    f. Click the **Firebug** icon ( ) to open the Firebug pane.

    a. In the left pane, click the **Inspect** icon ( ) to inspect an element in the page.
    b. Highlight the requisition list table area. By referring to the debug pane, you can see that this table is managed by a widget named **RequisitionListTable_Widget**. This corresponds to the **com.ibm.commerce.store.widgets.RequisitionLists** widget, which is

**Stores/Widgets_701/com.ibm.commerce.store.widgets.RequisitionLists**.




2. Update the storefront to display the customized information.
    a. In Java EE perspective, Enterprise Explorer view, go to **Stores\WebContent\Widgets_701\com.ibm.commerce.store.widgets.RequisitionLists**.
    b. Select and open the **RequisitionLists_Data.jspf** file.
    c. Locate the following code snippet:

```
<wcf:rest var="response" url="store/{storeId}/requisition_list">
   <wcf:var name="storeId" value="${storeId}" encode="true"/>
   <wcf:param name="q" value="usable"/>
   <wcf:param name="pageNumber" value="${currentPage}"/>
   <wcf:param name="pageSize" value="${pageSize}"/>
   <wcf:param name="orderBy" value="D-lastUpdate"/>
</wcf:rest>
```

    d. Add the following line of code to the end to specify your custom profile:

```
<wcf:param name="profileName" value="MyCompany_Store_Summary"/>
```

The REST call should resemble the following screenshot:

```
<wcf:rest var="response" url="store/{storeId}/requisition_list">
    <wcf:var name="storeId" value="${storeId}" encode="true"/>
    <wcf:param name="q" value="usable"/>
    <wcf:param name="pageNumber" value="${currentPage}"/>
    <wcf:param name="pageSize" value="${pageSize}"/>
    <wcf:param name="orderBy" value="D-lastUpdate"/>
    <wcf:param name="profileName" value="MyCompany_Store_Summary"/>
</wcf:rest>
```

    e. Locate the following code snippet:

```
<wcf:set target="${reqListMap}" key="updated"
value="${formattedReqLastUpdate}"/>
```

    f. Comment out the original line and add the following code snippet:

```
<wcf:set target="${reqListMap}" key="updated"
value="${requisitionList.owner}"/>
```

The section should resemble the following screenshot:

```
<%--<wcf:set target="${reqListMap}" key="updated" value="${formattedReqLastUpdate}"/>--%>
<wcf:set target="${reqListMap}" key="updated" value="${requisitionList.owner}"/>
```

g. Save your changes and close the file.

h. Go to **Stores\WebContent\Widgets_701\Properties**.

i. Open the **widgettext_B2B_en_US.properties** and **widgettext_B2B.properties** files.

j. Locate the following code snippet:

```
REQUISITIONLIST_UPDATED = Last Updated
```

k. Replace it with the following code snippet:

```
REQUISITIONLIST_UPDATED = Owner
```

l. Save your changes and close the files.

m. Restart the test server.

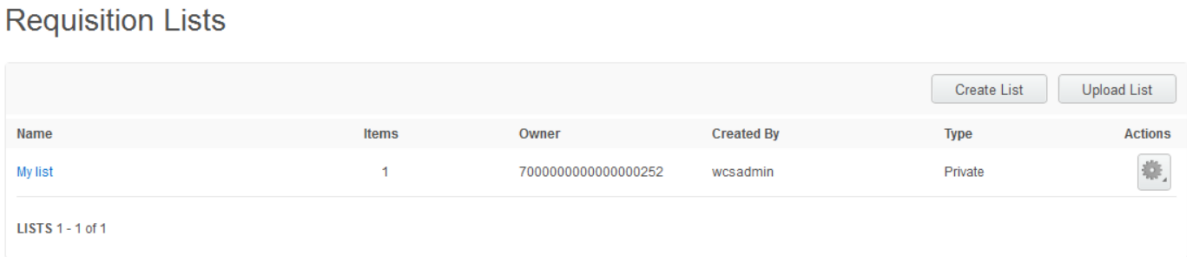
Next, verify your storefront customization.

## Part 8: Verifying the store widget customization in the storefront

1. Open the Aurora B2B starter store in a web browser:

   https://localhost/webapp/wcs/stores/servlet/en/aurorab2besite

2. Click **Sign In/Register** and log in to the store as a Site Administrator. For example, wcsadmin.
3. Click **My Account** to load the My Account page.
4. On the left pane, click the **Requisition Lists** tab under the **Order** section.
5. Review the requisition lists. The **Last Updated** table column is now replaced by **Owner**:



## Part 9: Annotating your REST resource handler API to use the Swagger user interface

In this lesson, you learn how to annotate your custom REST handler (created in the previous lesson) to expose the API documentation through the Swagger user interface (UI).

WebSphere Commerce uses the Swagger UI to provide an interactive representation of your REST API.

By default, the WebSphere Commerce REST services are annotated to use the REST Discovery API. This API generates resource listings and API declarations for all the REST services that are registered in your environment. These listings and declarations conform to Swagger specification 1.2. You can view an interactive version of the REST API in a web browser by using the Swagger user interface.

Annotate your custom REST services to use WebSphere Commerce REST Discovery API. This annotation infrastructure documents all of the details of your custom REST API. When you are using this infrastructure, you can discover the available REST resources and methods, and test the API from inside the Swagger user interface.

1. In Java EE perspective, Enterprise Explorer view, open **MyResourceHandler.java**.

    a. Browse to **WebSphereCommerceServerExtensionsLogic** > **src** > **com.mycompany.commerce.rest.mycomponent.handler**.

    b. Open the **MyResourceHandler.java** file.

    The resource handler already includes the required annotation. We will review it now.

2. At the beginning of the code section, the strings for each method's description are defined. Locate the following lines of code:

```
private static final String FIND_BY_MY_RESOURCE_ID_DESCRIPTION =
"Finds my resource by its ID.";
      private static final String
FIND_BY_MY_RESOURCE_LIST_DESCRIPTION = "Finds my resource by My
resource list.";
      private static final String
FIND_MY_RESOURCE_BY_QUERY_DESCRIPTION = "Finds my resources by a
query. See each query for details on input and output.";
      private static final String SELF_QUERY_DESCRIPTION = "Finds
my resources created by the current user.";
      private static final String CREATE_ACTION_DESCRIPTION =
"Creates my resource.";
```

3. Each resource is annotated to contain a description. Doing so ensures that a description is displayed for your resource on the Swagger UI landing page that lists of all the available classes and their descriptions.

   Find the following snippet:

   ```
   @Path("store/{storeId}/my_resource")
   @Description("My resource.")
   ```

4. The **findByMyResourceId** method is annotated. For more information about the available annotations types, see WebSphere Commerce REST.

   Locate the following lines of code:

   ```
   @Produces({ MediaType.APPLICATION_JSON,
   MediaType.APPLICATION_XML, MediaType.APPLICATION_XHTML_XML,
   MediaType.APPLICATION_ATOM_XML })
   @Description(FIND_BY_MY_RESOURCE_ID_DESCRIPTION)
   @ResponseSchema(valueClass = MyResourceListDataBean.class,
   valueClassType = ClassType.Bean,
     profile = PROFILE_NAME_MY_COMPANY_DATABEAN_SUMMARY,
   responseCodes = {
       @ResponseCode(code = 200, reason = RESPONSE_200_DESCRIPTION),
       @ResponseCode(code = 400, reason = RESPONSE_400_DESCRIPTION),
       @ResponseCode(code = 401, reason = RESPONSE_401_DESCRIPTION),
       @ResponseCode(code = 403, reason = RESPONSE_403_DESCRIPTION),
       @ResponseCode(code = 500, reason = RESPONSE_500_DESCRIPTION)
   })
   ```

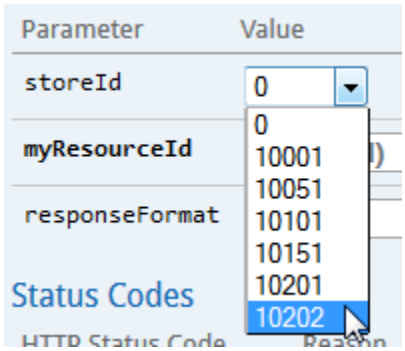
   - **@Description** documents the short description of a REST API method.
   - **@ResponseSchema** documents the response schema of a REST API.

5. In each GET method (e.g. **findByMyResourceId**) and POST method (e.g. create) **@ParameterDescription** with store id is included, so you can select your store ID from a drop-down list in Swagger.

   ```
   @PathParam(PARAMETER_STORE_ID) @ParameterDescription(description
   = PARAMETER_STORE_ID_DESCRIPTION,
   valueProviderClass = StoreIdProvider.class, required = true)
   String storeId,
   ```

   a. **@ParameterDescription** describes a parameter that is used in a REST API.

   You will be able to select your store ID for the GET method from a drop-down list in Swagger:




6. The remaining GET methods are also annotated using the same convention as the **findByMyResourceId** method.

```
public Response findMyResourceList
@Description(FIND_BY_MY_RESOURCE_LIST_DESCRIPTION)
```

```
public Response findByQuery
@Description(FIND_MY_RESOURCE_BY_QUERY_DESCRIPTION)
```

```
public Response findSelf
@Description(SELF_QUERY_DESCRIPTION)
```

7. The **CREATE** method is annotated.

```
@InternalProduces({ MediaType.APPLICATION_JSON,
MediaType.APPLICATION_XML,
 MediaType.APPLICATION_XHTML_XML, MediaType.APPLICATION_ATOM_XML
})
@Description(CREATE_ACTION_DESCRIPTION)
@AdditionalParameterList({
        @ParameterDescription(name =
ParameterDescription.BODY_NAME, paramType =
ParameterDescription.BODY_TYPE,
          description = ParameterDescription.BODY_DESCRIPTION,
typeClass = ResourceCreateRequest.class) })
    @ResponseSchema(valueClass = MyResourceCreateResponse.class,
valueClassType = ClassType.Command,
      profile =  PROFILE_NAME_MY_COMMAND_SUMMARY,responseCodes =
{
    @ResponseCode(code = 200, reason = RESPONSE_200_DESCRIPTION),
    @ResponseCode(code = 201, reason = RESPONSE_200_DESCRIPTION),
    @ResponseCode(code = 400, reason = RESPONSE_400_DESCRIPTION),
    @ResponseCode(code = 401, reason = RESPONSE_401_DESCRIPTION),
    @ResponseCode(code = 403, reason = RESPONSE_403_DESCRIPTION),
    @ResponseCode(code = 500, reason = RESPONSE_500_DESCRIPTION)
})
```

a. **@AdditionalParameterList** adds parameters to a REST API method without adding them to the Java method signature. This **@AdditionalParameterList** contains a

**@ParameterDescriptionannotation** for the REST request body. When you view this API from the Swagger UI, this parameter adds a text-box in which you can input request body content for testing purposes.

8. Near the top of the file, locate the definition of **ResourceCreateRequest** class and **MyResourceCreateResponse** class we used in the previous step.

```
private static class ResourceCreateRequest {
@ParameterDescription(description = "The content of field 1 for
my resource .")
private String field1;
@ParameterDescription(description = "The content of field 2 for
my resource.")
private Timestamp field2;
}
private static class MyResourceCreateResponse {
@ParameterDescription(description = "The ID of my resource.")
private long myResourceId;
}
```

## Part 9: Verify your custom REST resource handler API through the Swagger user interface

In this lesson, you use the Swagger user interface to view interactive documentation of your REST API.

1. Log in to your Aurora B2B starter store as a Site Administrator.

   **Note:** Logging in to the store sets up the security tokens so that you can make REST calls from Swagger. Although you can still view the REST resources in Swagger, failing to log in to the store with the appropriate permissions might prevent you from running any REST calls from Swagger.

   a. Open the Aurora B2B starter store in a web browser:

      https://localhost/webapp/wcs/stores/servlet/en/aurorab2besite

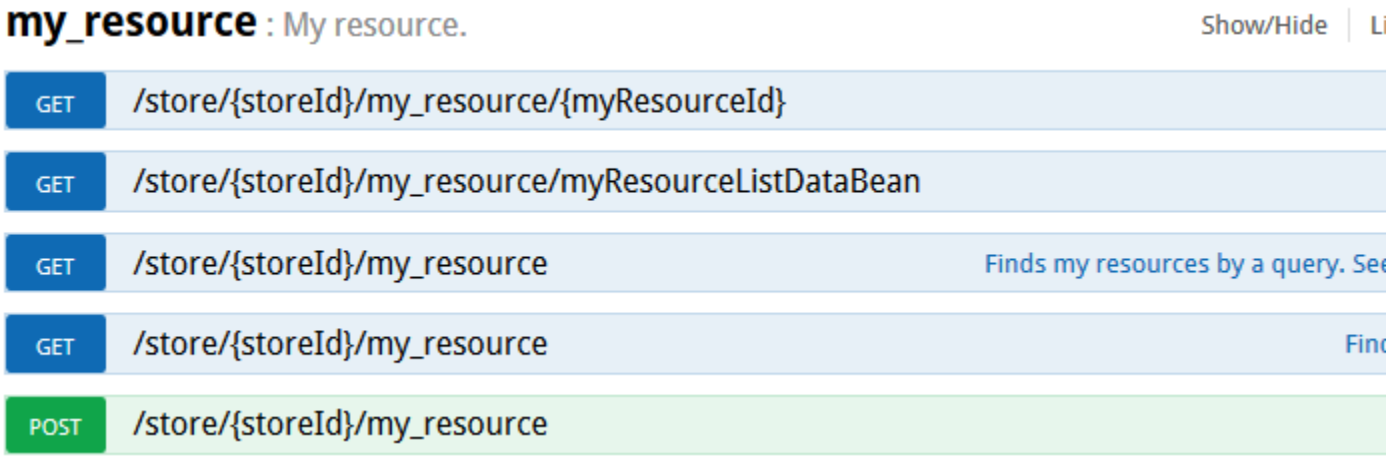
   b. Click **Sign In/Register** and log in to the store as a Site Administrator. For example, wcsadmin.

2. After you log in, go to:

   https://localhost/webapp/wcs/stores/servlet/swagger/index.html

   to access the Swagger user interface.

3. Scroll through the REST resource handlers, which are listed alphabetically, to find **my_resource**. Click to expand the resource in Swagger.




   Swagger displays the new methods that you created earlier. Notice the method descriptions on the right side.

4. Test your available methods. Rather than use the Poster plug-in, you can use the Swagger user interface to test your available methods. Start by testing the POST method.

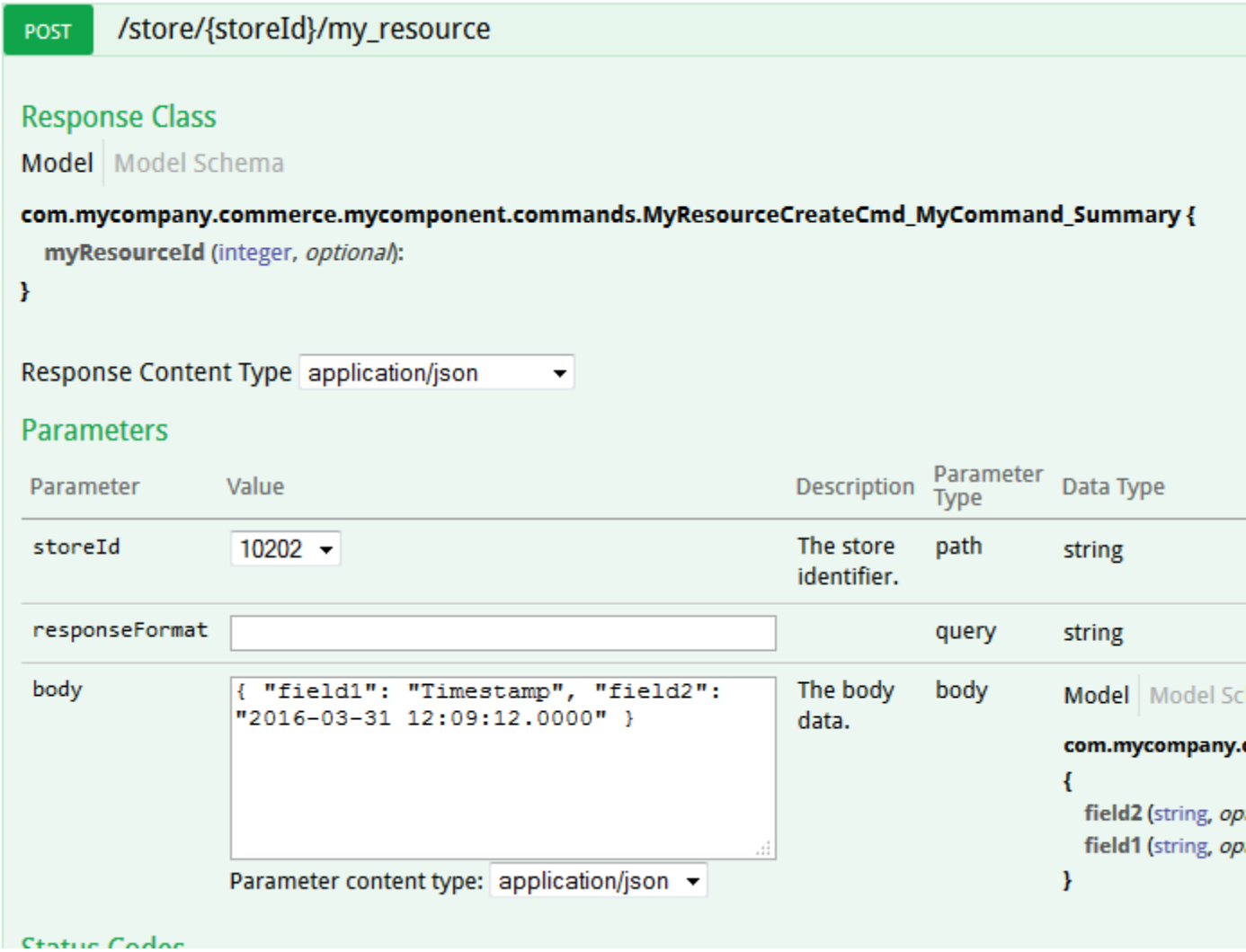
   a. Select the POST method to expand it.

b. Choose **storeId** 10202 from the drop-down list.

c. Copy the following values into the **body** text box:

```
{ "field1": "Timestamp", "field2": "2016-03-31
12:09:12.0000" }
```

Your **Paramaters** section should resembles the following screen capture:




d. Click the **Try it out!** button.

This action adds resource data into the **XMYRES** table. If the new data is created successfully, the corresponding **myResourceID** is displayed in the **Response Body** section:

Request URL

```
https://localhost:443/wcs/resources/store/10202/my_resource
```

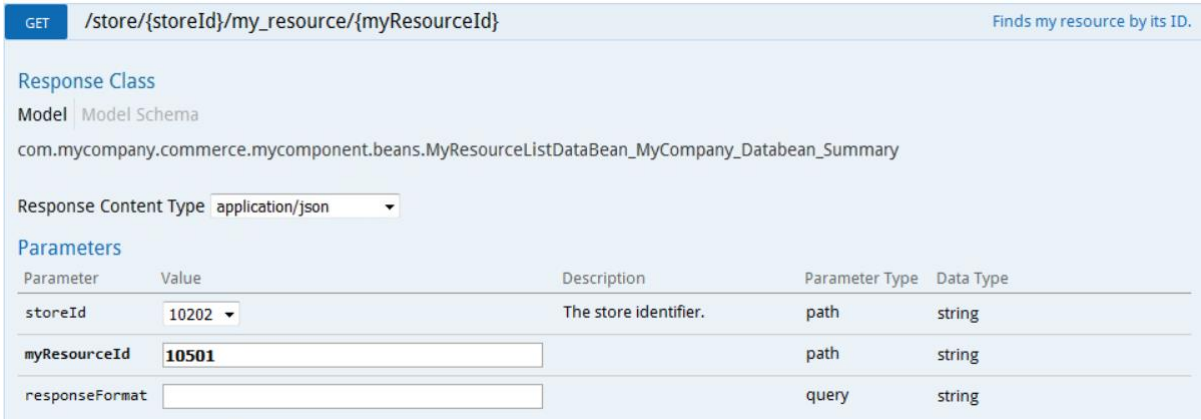
Response Body

```
{
  "myResourceId": 10501
}
```

Response Code

```
200
```

5. Use a GET method to retrieve the data you created in the previous step.
    a. Click the first **GET** method to find a resource by its ID.
    b. Select the **storeId** 10202 from the drop-down list.
    c. Specify the value for **myResourceId** that was received in the response body in the preceding step, as shown in the following screen capture:




    d. Click the **Try it out!** button.

       The resource data is retrieved by using the resource ID number 10501, and then displayed in the response window:


Similarly, you can test the **findMyResourceList** method.

## What you did in this exercise

In this tutorial, you created a REST handler for mapping the custom data bean and controller command, and customized an existing configuration-based data bean mapping to return more data.

You also reviewed the annotations in the custom REST handler, which enabled Swagger to display the available REST API.

## End of exercise

# Exercise 6: WebSphere Commerce Build and Deployment

## Estimated time

01:30 hours

## What this exercise is about

The objective of this exercise is to use the WebSphere Commerce Build and Deployment tool (WCBD) to configure and run the build process to create a deployment package and then deploy the package to the Toolkit environment.

## What you should be able to do

After completing this exercise, you will be able to:

- Configure and run the Build process.
- Configure and run the Deploy process.

## Introduction

This exercise demonstrates the use of the WebSphere Commerce Build and Deployment Tool. Using WCBD, you configure and perform a build to create a deployment package. You then deploy the package to the Toolkit environment.

## Requirements

- WebSphere Commerce Developer V8 Enterprise
- Extended sites store archive published
- AuroraESite store created

## Exercise instructions

In this exercise, we will make perform a minor JSP code change, build a deployment package which will include the updated JSP and then deploy the package so that the updated JSP is deployed.

## Part 1: Perform a Build to create a deployment package by using the WCBD tool

In this part of the exercise, you perform a build and create a deployment package. For simplicity, this exercise uses a directory on the file system as a repository instead of a full source control system.

1. Navigate to the repository and examine the files.
    a. Using Windows Explorer, navigate to **C:\exercises\wcbd\Repository**.
    b. Expand the **workspace** folder. The directory structure under the workspace folder corresponds to the WCBD default repository structure. You can find a description of the repository structure in the WC Information Center:

    https://www.ibm.com/support/knowledgecenter/SSZLC2_8.0.0/com.ibm.commerce.developer.doc/concepts/cdewcbdrepo.htm

    The **Stores** directory contains the updated JSP that will be deployed. In **TopCategoriesDisplay.jsp**, we have updated the <title> to include the text “Success!”:

    ```
    <title><c:out
    value="${pageTitle}"/> Success!</title>
    ```

2. Prepare to build the deployment package. The WCBD tool provides sample configuration files that can be used as templates or starting points to configure the tool for the local environment. In the following steps, you set your local file system paths in the WCBD configuration files.
    a. Using Windows Explorer, navigate to **C:\IBM\WCDE80GM\wcbd\extract** and copy the **wcbd-sample-extract-local.xml** file into the **C:\IBM\WCDE80GM\wcbd** folder. Rename the file to **extract-local.xml**.
    b. Using Windows Explorer, navigate to **C:\IBM\WCDE80GM\wcbd\extract** and copy the **wcbd-sample-extract-local.properties** file into the **C:\IBM\WCDE80GM\wcbd** folder. Rename the file to **extract-local.properties**.

c. Open **extract-local.xml** from the **C:\IBM\WCDE80GM\wcbd** folder by **right-clicking** the file in Windows Explorer and selecting **Edit** to open the file in Notepad.

Change the following line:

```
<project name="wcbd-sample-extract-local" default="all">
```

to:

```
<project name="extract-local" default="all">
```

d. Save the changes and close the file.

---

**Note:** Instead of making changes manually, you can use the **extract-local.xml** file provided in **C:\exercises\wcbd\build**.

---

e. Open **extract-local.properties** with Notepad from the

**C:\IBM\WCDE80GM\wcbd** folder by double-clicking the file in Windows Explorer.

f. Set the value of the **local.extract.dir** to **C:/exercises/wcbd/Repository**. The properties files use the forward slash '/' as the file separator, even for Windows systems.

g. Save the changes and close the file.

---

**Note:** Instead of making changes manually, you can use the **extract-local.properties** file provided in **C:\exercises\wcbd\build**.

---

h. Make copies and then rename the following files that are in the **C:\IBM\WCDE80GM\wcbd** folder:

***wcbd-build.properties.template*** to **build-local.properties**

***wcbd-build.private.properties.template*** to **build-local.private.properties**

***wcbd-setenv.bat.template*** to **setenv.bat**

i. Apache Ant is a prerequisite for WCBD. For your convenience, we have already downloaded it and made it available at **C:\exercises\wcbd\apache-ant-1.9.9**. You can also download it from the Apache Ant website:

http://ant.apache.org

j. Copy the **ant-contrib-1.0b3.jar** file from **C:\exercises\wcbd** and add it to the **C:\IBM\WCDE80\wcbd\lib** directory. You can also download it from the Apache Ant Contrib website:

https://sourceforge.net/projects/ant-contrib/files/ant-contrib/1.0b3/

k. Open the file **setenv.bat** in Notepad by right-clicking the file in Windows Explorer and select **Edit**.

l. Locate the following properties and set the values as indicated. (It is important to notice that for this file, you use the "\" character).

```
ANT_HOME=C:\exercises\wcbd\apache-ant-1.9.9

WAS_HOME=C:\IBM\AppServer
```

m. Save the changes and close the file.

---

**Note:** Instead of making changes manually, you can use the **setenv.bat** file provided in **C:\exercises\wcbd\build**.

---

n. Open the file **build-local.properties** by double-clicking on the file. If prompted to choose a program to open the file, select **Notepad**.

o. Locate the properties that are listed in the following table and set the values as indicated. (It is important to notice that for this file, you use the **"/"** character.)

| Property name | Value |
|---|---|
| was.home | C:/IBM/AppServer |
| wc.home | C:/IBM/WCDE80GM |
| extract.ant.file= | ${basedir}/extract-local.xml |
| web.module.list= | Stores |

p. Save the changes and close the file.

---

**Note:** Instead of making changes manually, you can use the **build-local.properties** file provided in **C:\exercises\wcbd\build**.

---

3. Run the build process to create the deployment package.

a. Open a command prompt and go to the directory **C:\IBM\WCDE80GM\wcbd**.

b. Run the following command:

```
wcbd-ant.bat -buildfile wcbd-build.xml -Dbuild.label=StorePackage -
Dbuild.type=local
```

c. Wait for the build process to complete. Ensure that you see a **Build Successful** message in the command prompt.

4. Extract the deployment package and examine its contents.

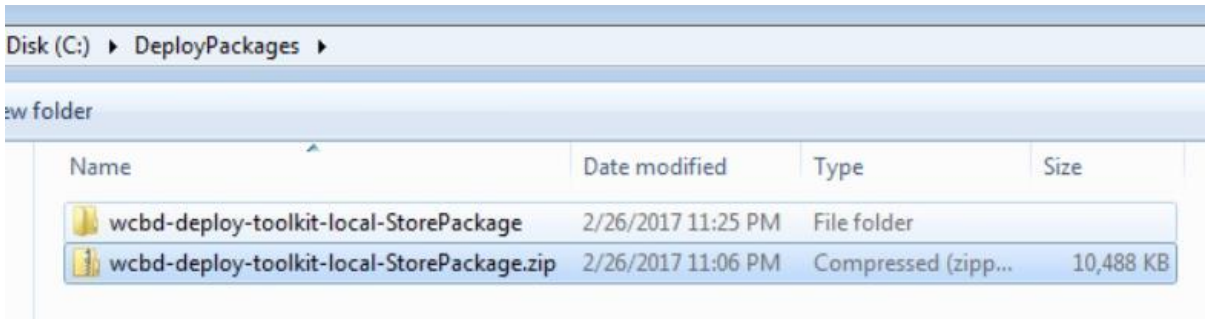
a. Create a folder named **C:\DeployPackages**.

b. Navigate to **C:\IBM\WCDE80GM\wcbd\dist** in Windows Explorer.

   Two folders, **server** and **toolkit**, exist in the **dist** directory. The package inside the **server** directory is to be deployed in a server environment, while the package inside the **toolkit** directory is to be deployed in a toolkit environment.

c. Copy **wcbd-deploy-toolkit-local-StorePackage.zip** to **C:\DeployPackage.**

d. Unzip **wcbd-deploy-toolkit-local-StorePackage.zip**. The **wcbd-deploy-toolit-local-StorePackage\source** folder contains the source code changes that will be deployed. You can open it and review it. You should find the updated **TopCategoriesDisplay.jsp** inside it.




You have created a deployment package that can be handed over to a System Administrator or Deployment Manager that they can use to deploy to a Server environment. This deployment package can be used to deploy to the Dev, QA, and Test environments.

## Part 2: Perform a Deployment with the WCBD tool

In this part of the exercise, you configure and run a deployment of the package was that created in Part 1.

1. In Windows Explorer, navigate to **C:\DeployPackages\wcbd-deploy-toolkit-local-StorePackage** directory.
2. Copy and rename the following files that are in the directory:

   **wcbd-deploy.properties.template to deploy-toolkit.properties**

   **wcbd-deploy.private.properties.template to deploy-toolkit.private.properties**

   **wcbd-setenv.bat.template to setenv.bat**
3. Open **setenv.bat** for editing. Set the property **WCDE_HOME** to **C:\IBM\WCDE80GM**.

   **Note:** Instead of making changes manually, you can use the **setenv.bat** file provided in **C:\exercises\wcbd\deploy**.
4. Open **deploy-toolkit.properties** and set:

   ```
   web.module.list=Stores
   ```

When you perform a deployment to the toolkit, it is important to ensure that the **workspace** directory under the WebSphere Commerce Developer installation directory is backed up. In case there a deployment failure, the backup can be used to restore a functioning workspace. The deployment script can automatically create a backup before doing the deployment. Alternatively, you can manually take a backup. The backup can take several minutes depending on the size of the workspace directory. For demonstration purposes, we will disable automatic backup and take a manual backup instead.

5. Take a backup of **C:\IBM\WCDE80GM\workspace**. Make sure that WebSphere Commerce Developer and the test server are not started, otherwise some files may be locked and may not be copied. They should also be stopped before running the deployment process.
6. In **deploy-toolkit.properties**, set:

   ```
   run.backup=false
   ```
7. Set up the database related properties in **deploy-toolkit.properties**:

   - `db.name=${wc.home}/db/mall`
   - `db.schema.name=APP`

- `jdbc.url=jdbc:derby:${db.name}`
- `jdbc.driver=org.apache.derby.jdbc.EmbeddedDriver`
- `jdbc.driver.path=${wc.home}/lib/derby.jar`

**Note:** Instead of making changes manually, you can use the **deploy-toolkit.properties** file provided in **C:\exercises\wcbd\deploy**.

8. Set up the following properties in **deploy-toolkit.private.properties**:
   - `db.user.name=APP`
   - `db.user.password=afacWLqg1trlbNupQsppiw==`

**Note:** The user name and password given here are the default credentials for the database when Apache Derby is used as the development database. These credentials will remain the same in every WebSphere Commerce Developer V8 environment and are given in **wc-server.xml**.

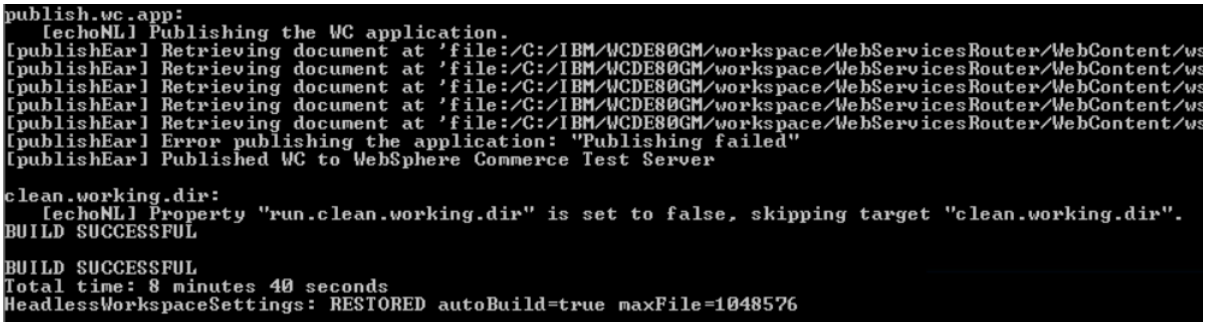
**Note:** Instead of making changes manually, you can use the **deploy-toolkit.private.properties** file provided in **C:\exercises\wcbd\deploy**.

9. Ensure that WebSphere Commerce Developer and the test server are shut down.
10. Open a command prompt and navigate to the **C:\DeployPackages\wcbd-deploy-toolkit-local-StorePackage** directory.
11. Run the following command:

```
wcbd-rad-ant.bat -buildfile wcbd-deploy.xml -Dtarget.env=toolkit
```

12. Wait for the deployment to finish. This will take several minutes.

    Ensure that the **Build Successful** message is displayed.



**Note:** If you see a “Publishing failed” error, you can ignore it. You can manually publish the application from RAD.

13. Start WebSphere Commerce Developer.
14. Expand **Stores > Web Content > AuroraStorefrontAssetStore > ShoppingArea > CatalogSection > CategorySubsection**.
15. Double-click **TopCategoriesDisplay.jsp** to open it in the JSP Editor.
16. Search for **<title** in the JSP. The <title> should look like this:

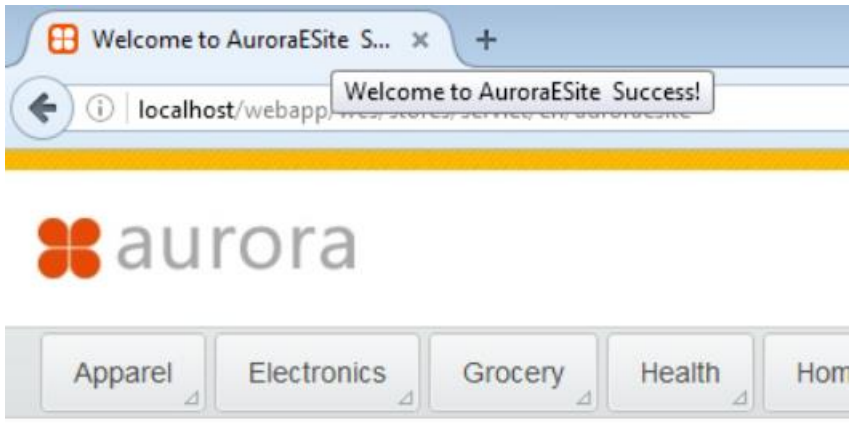
```
<title><c:out value="${pageTitle}"/> Success!</title>
```

This is the change that we had included in the local repository to be deployed.

17. Start WebSphere Commerce Test Server. Republish the WC application.
18. Open the Aurora B2C store in a browser window:

    http://localhost/webapp/wcs/stores/servlet/auroraesite

    The title in the store should display as expected.



## What you did in this exercise

In this exercise, you learned how to use the WCBD tool to configure and run the build process to create a deployment package. You then deployed the package to your local Toolkit environment.

## End of exercise