IBM WebSphere Commerce

Programming Guide and Tutorials

Version 5.5

IBM WebSphere Commerce

Programming Guide and Tutorials

Version 5.5

Note:
Before using this information and the product it supports, be sure to read the information under “Notices” on page 385.

First Edition, Second Revision (December 2003)

This edition applies to IBM WebSphere Commerce Business Edition Version 5.5, IBM WebSphere Commerce - Express Version 5.5, IBM WebSphere Commerce Professional Edition Version 5.5 (product number 5724-A18), and to all subsequent releases and modifications until otherwise indicated in new editions. Ensure that you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality.

IBM welcomes your comments. You can send your comments by using the online IBM WebSphere Commerce documentation feedback form, available at the following URL:

http://www.ibm.com/software/webservers/commerce/rcf.html

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2000, 2003. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Before you begin

The *WebSphere Commerce Programming Guide and Tutorials* provides information about the WebSphere® Commerce architecture and programming model. In particular, it provides details on the following topics:

- Component interactions
- Design patterns
- Persistent object model
- Access control
- Error handling and messages
- Command implementation
- Development tools
- Deployment of customized code

In addition, this book includes the following tutorials:

- Creating new business logic
- Modifying an existing controller command
- Extending the object model and modifying an existing task command
- Extending an existing WebSphere Commerce entity bean

Updates to this book

The most recent version of this document is available as a PDF file from the Technical Library section of the WebSphere Commerce Web site:

http://www.ibm.com/software/commerce/library/

Updated versions of this document are also available from the WebSphere Commerce Zone at WebSphere Developer Domain:

http://www.ibm.com/websphere/developer/zones/commerce

For additional support information, see the WebSphere Commerce Support site:

http://www.ibm.com/software/commerce/support/

* Updates from earlier versions of this document are identified in the text of the
* document as follows:

© Copyright IBM Corp. 2000, 2003

- The asterisk character (*) in the margin identifies updates that have been made in the current revision of the document.
- The Express icon (see "Conventions used in this book") identifies updates that were made in the previous revision.

Conventions used in this book

This book uses the following highlighting conventions:

Boldface type indicates commands or graphical user interface (GUI) controls such as names of fields, buttons, or menu choices.

`Monospaced type` indicates examples of text you enter exactly as shown, as well as directory paths.

Italic type is used for emphasis and variables for which you substitute your own values.

This icon marks a Tip — additional information that can help you complete a task.

400 indicates information specific to WebSphere Commerce for the IBM® @server iSeries™ 400® (formerly called AS/400®).

AIX indicates information that is specific to WebSphere Commerce for AIX®.

Linux indicates information that is specific to WebSphere Commerce for Linux.

Solaris indicates information that is specific to WebSphere Commerce for Solaris Operating Environment software.

Windows indicates information that is specific to WebSphere Commerce for Windows® 2000.

DB2 indicates information specific to the DB2 Universal Database™ family (hereafter referred to as simply as DB2®).

DB2 390 z/OS indicates information specific to DB2 Universal Database for OS/390® and z/OS®, Version 7 (hereafter referred to as DB2 for OS/390 and z/OS).

Oracle indicates information specific to Oracle. You can use Oracle as your database management system if you are using WebSphere Commerce Business Edition or WebSphere Commerce Professional Edition.

Business indicates information specific to IBM WebSphere Commerce Business Edition.

Express indicates information specific to IBM WebSphere Commerce - Express.

Professional indicates information specific to IBM WebSphere Commerce Professional Edition.

Developer indicates information specific to the WebSphere Commerce development environment. For WebSphere Commerce Business Edition and WebSphere Commerce Professional Edition, your development environment is WebSphere Commerce Studio, Version 5.5.

For WebSphere Commerce - Express, the development environment is WebSphere Commerce - Express Developer Edition, Version 5.5

Knowledge requirements

This book should be read by Store Developers that need to understand how to customize a WebSphere Commerce application. Store Developers that are performing programmatic extensions should have knowledge in the following areas:

- Java™
- Enterprise JavaBeans component architecture
- JavaServer Pages technology
- HTML
- Database technology
- Business Professional WebSphere Studio Application Developer, Version 5

  Express WebSphere Studio Application Developer, Version 5.1

Path variables

This guide uses the following variables to represent directory paths:

WC_installdir
: This is the installation directory for WebSphere Commerce. The following are the default installation directories for WebSphere Commerce on various operating systems:

- 400 /QIBM/ProdData/CommerceServer55
- AIX /usr/WebSphere/CommerceServer55
- Linux /opt/WebSphere/CommerceServer55
- Solaris /opt/WebSphere/CommerceServer55
- Windows C:\Program Files\WebSphere\CommerceServer55

WC_userdir
: The directory for all the data used by WebSphere Commerce which can be modified or needs to be configured by the user.

  - 400 /QIBM/UserData/CommerceServer55

WAS_installdir
: This is the installation directory for WebSphere Application Server. The following are the default installation directories for WebSphere Application Server on various operating systems:

  - 400 /QIBM/ProdData/WebAs5/Base
  - AIX /usr/WebSphere/AppServer
  - Linux /opt/WebSphere/AppServer
  - Solaris /opt/WebSphere/AppServer
  - Windows C:\Program Files\WebSphere\AppServer

WAS_userdir
: The directory for all the data used by WebSphere Application Server which can be modified or needs to be configured by the user.

  - 400 QIBM/UserData/WebAS5/Base/*WAS_instancename* and *WAS_instance_name* represents the name of the WebSphere Application Server with which your WebSphere Commerce instance is associated.

WCDE_installdir
: The installation directory for the WebSphere Commerce development environment. For WebSphere Commerce Business Edition and WebSphere Commerce Professional Edition, your development environment is WebSphere Commerce Studio, Version 5.5. The following is the default installation directory: C:\WebSphere\CommerceStudio55.

  For WebSphere Commerce - Express, the development environment is WebSphere Commerce - Express Developer Edition, Version 5.5. The following is the default installation directory: C:\WebSphere\CommerceDev55

Where to find more information

For more information related to WebSphere Commerce, refer to the following Web site:

http://www.ibm.com/software/commerce/library/

Contents

Before you begin iii
Updates to this book iii
Conventions used in this book iv
Knowledge requirements v
Path variables. v
Where to find more information vii

Part 1. Concepts and architecture . 1

Chapter 1. Overview 3
WebSphere Commerce software components . 3
WebSphere Commerce application architecture 4
WebSphere Commerce run-time architecture . 7
Servlet engine. 9
Protocol listeners. 9
Adapter manager 9
Adapters 10
Web controller 11
Commands 12
WebSphere Commerce entity beans . . . 13
Data beans 14
Data bean manager 14
JavaServer Pages templates 14
instance_name.xml configuration file . . . 14
Summary for a request 15

Part 2. Programming model . . . 17

Chapter 2. Design patterns 19
Model-View-Controller design pattern . . . 19
Command design pattern 20
Command framework. 21
Command factory 23
Command flow 26
Command registration framework . . . 28
Display design pattern 37
JSP templates and data beans 38
Types of data beans 38
Invoking controller commands from within a JSP template 42
Lazy fetch data retrieval 42
Setting JSP attributes - overview 43
Required property settings 45

Chapter 3. Persistent object model . . . 47
Implementation of WebSphere Commerce entity beans 47
WebSphere Commerce entity beans - overview 47
Deployment descriptors for WebSphere Commerce enterprise beans 48
Extending the WebSphere Commerce object model 50
Object life cycles 77
Transactions 77
Other considerations for entity beans . . 78
Using entity beans 82
Database considerations 82
Database schema object naming considerations 83
Database column data type considerations 85
* Data type differences among databases . . 86

Chapter 4. Access control. 89
Understanding access control 89
Overview of resource protection in WebSphere Application Server 89
Security consideration for URL parameters 91
Introduction to WebSphere Commerce access control policies. 92
Types of access control 99
Access control interactions 101
Protectable interface 104
Groupable interface 104
Finding more information about access control 105
Implementing access control 105
Identifying protectable resources 105
Implementing access control in enterprise beans 106
Implementing access control in data beans 108
Implementing access control in controller commands 110
Implementing access control policies in views 113
Modifying access control on existing WebSphere Commerce resources 114

© Copyright IBM Corp. 2000, 2003

Adding a new relationship to an existing WebSphere Commerce entity bean . . . 114
Adding access control to an existing WebSphere Commerce entity bean that is not already protected 116
Understanding the access control implications when a controller command is extended 116
Sample access control policies for development purposes 119
Sample access control policy for new views 119
Sample command-level access control policy for new controller commands . . 119
Sample resource-level access control policy for a new command and enterprise bean 121

Chapter 5. Error handling and messages 123
Command error handling 123
Types of exceptions 123
Error message properties files. 124
Exception handling flow 124
Exception handling in customized code 126
Creating messages 128
Execution flow tracing 130
JSP template error handling 130

Chapter 6. Command implementation . . 133
New commands - introduction 133
Packaging customized code 136
Command context 137
Temporary changes to contextual information for URL commands 138
New controller commands 139
isGeneric method 139
isRetriable method 140
setRequestProperties method 140
validateParameters method 141
getResources method 141
performExecute method. 141
Long-running controller commands . . . 142
Formatting of input properties to view commands 143
Flattening input parameters into a query string for HttpRedirectView 143
Handling a limited length redirect URL 143
Setting attributes in the HttpServletRequest object for HttpForwardView. 145
Database commits and rollbacks for controller commands. 145
Example of transaction scope with a controller command 146
New task commands. 148
Customization of existing commands . . . 149
Customizing existing controller commands 149
Customizing existing task commands . . 154
Data bean customization 156

Chapter 7. Trading agreements and business policies (Business Edition) . . 157
Introduction 157
Business policy objects and commands. . . 158
ToolTech sample contract data 160
CONTRACT table sample data 160
TERMCOND table sample data 161
POLICYTC table sample data 161
POLICY table sample data 162
TRADEPOSCN table sample data . . . 162
SHIPMODE table sample data 162
Extending the existing contract model . . . 162
Creating a new business policy 163
Creating a new business policy type . . 164
Writing the new business policy command 165
Registering the new business policy and business policy command 168
Relating a terms and conditions object to a new business policy 168
Creating new terms and conditions . . . 169
Invoking the new business policy 184
Creating a contract 185
Contract customization scenarios. 185
Rebate scenario 185

Part 3. Development environment 195

Chapter 8. Development environment . . 197
Typical development environment 197
WebSphere Studio Application Developer 198
Development environment for iSeries . . . 198
* Using different database management
* systems for development and production . . 199
Overview of the WebSphere Commerce enterprise bean conversion tool 199
Payment options within the development environment 199

Chapter 9. Deployment details 201
User permission requirements for deployment steps 201
Incremental deployment 202
Deploying enterprise beans 202
Creating the EJB JAR file 202
Updating the EJB JAR file on the target WebSphere Commerce Server 206
Deploying commands and data beans . . . 208
Creating the JAR file 209
Updating the JAR file on the target WebSphere Commerce Server 209
Deploying store assets 210
Exporting store assets 210
Transferring store assets 211
Updating the target database 212
Access control updates 212

Part 4. Tutorials 215

Chapter 10. Tutorial: Creating new business logic 217
Locating the sample code 218
Preparing your workspace 218
Creating a new view 222
Registering MyNewView 222
Creating a properties file for the tutorial 224
Creating MyNewJSPTemplate 225
Creating and loading access control policies for MyNewView 227
Testing MyNewView 228
Creating a new controller command . . . 230
Registering MyNewControllerCmd . . . 230
Creating the MyNewControllerCmd interface 232
Creating the MyNewControllerCmdImpl implementation class 232
Creating and loading access control policies for the command 233
Testing MyNewControllerCmd 234
Passing information from MyNewControllerCmd to MyNewView . . 235
Passing information using a TypedProperties object 235
Passing information using a data bean 238
Parsing and validating URL parameters in MyNewControllerCmd 244
Adding new fields to MyNewControllerCmd 244
Passing URL parameters to the view . . 245
Catching missing parameters and validating values 246
Adding new fields to MyNewDataBean 247
Modifying MyNewJSPTemplate to display the URL parameters 248
Testing URL parameter values 248
Creating a new task command 252
Creating MyNewTaskCmd 253
Calling the task command 255
Modifying MyNewJSPTemplate to add the greetings message 257
Testing MyNewTaskCmd 257
Modifying MyNewTaskCmd 258
Modifying MyNewControllerCmdImpl to create an object for the task command . . 259
Modifying the new task command for user name validation. 260
Modify MyNewJSPTemplate for user name validation 261
Testing user name validation 262
Creating a new entity bean 264
Creating the XBONUS table 264
Creating the BonusBean entity bean. . . 265
Integrating the Bonus entity bean with MyNewControllerCmd 275
Deploying the bonus points logic 289
Creating the command and data bean JAR file 289
Creating the EJB JAR file 290
Exporting store assets 291
Packaging access control policies. . . . 292
Transferring assets to your target WebSphere Commerce Server 292
Stopping your target WebSphere Commerce Server 292
Updating the database on your target WebSphere Commerce Server 292
Updating store assets on your target WebSphere Commerce Server 298
Updating the command and data bean JAR file on your target WebSphere Commerce Server 298
Updating the EJB JAR file on your target WebSphere Commerce Server 299
Verifying bonus points logic on the target WebSphere Commerce Server 300

Chapter 11. Tutorial: Modifying an existing controller command 303
Prerequisites 303

Creating the new MyOrderItemAddCmdImpl class 304
Creating message information. 306
Modifying the command registry 308
Testing the MyOrderItemAddCmdImpl command 309
Deploying MyOrderItemAddCmdImpl. . . 311
Creating the command JAR file 312
Exporting the message properties file . . 313
Transferring assets to the target WebSphere Commerce Server 313
Stopping your target WebSphere Commerce Server 313
Updating the database on the target WebSphere Commerce Server 314
Updating the command JAR file on the target WebSphere Commerce Server. . . 315
Updating the message properties on the target WebSphere Commerce Server. . . 315
Verifying the MyOrderItemAddCmdImpl logic on the target WebSphere Commerce Server. 315

Chapter 12. Tutorial: Extending the object model and modifying an existing task command 319
Prerequisites 320
Creating and populating the XORDGIFT table 320
Creating the OrderGift entity bean 321
Integrating the OrderGift entity bean into the shopping flow 334
Creating the OrderGiftDataBean 334
Creating the MyExtOrderProcessCmdImpl class 335
Compiling changes 337
Modifying display pages for gift messages. 337
Testing the new gift message functionality 340
Deploying the gift message functionality . . 343
Creating the command and data bean JAR file 343
Creating the EJB JAR file 344
Exporting store assets 345
Transferring assets to your target WebSphere Commerce Server 346
Stopping your target WebSphere Commerce Server 346
Updating the database on your target WebSphere Commerce Server 346
Updating store assets on your target WebSphere Commerce Server 348
Updating the command and data bean JAR file on your target WebSphere Commerce Server 349
Updating the EJB JAR file on your target WebSphere Commerce Server 349
Verifying the gift message functionality on the target WebSphere Commerce Server. 350

Chapter 13. Tutorial: Extending an existing WebSphere Commerce entity bean 353
Prerequisites 353
Creating and populating the XHOUSING table 353
Adding new fields to the User entity bean 354
Updating the schema and table mapping information 356
Creating the table definition for the XHOUSING table 356
Creating the XHOUSING table map. . . 358
Updating the mapping file 358
Generating the access beans and deployed code 359
Creating the MyPostUserRegistrationAddCmdImpl implementation 359
Modifying the command registry . . . 362
Modifying JSP templates to collect and display housing information 363
Testing the modified code 365
Deploying the housing survey logic. . . . 367
Creating the command JAR file 367
Creating the EJB JAR file 368
Exporting store assets 369
Transferring assets to your target WebSphere Commerce Server 369
Stopping your target WebSphere Commerce Server 370
Updating the database on your target WebSphere Commerce Server 370
Updating store assets on your target WebSphere Commerce Server 372
Updating the command JAR file on your target WebSphere Commerce Server. . . 372
Updating the EJB JAR file on your target WebSphere Commerce Server 373

Verifying the housing survey logic on the target WebSphere Commerce Server. . . . 374

Part 5. Appendixes 377

Appendix A. Configuring WebSphere Commerce component tracing in the WebSphere Commerce development environment 379
Output file 379

Appendix B. Where to find more information 381
WebSphere Commerce development environment information 381
WebSphere Commerce development environment online help 381
WebSphere Commerce Web site 382
WebSphere Developer Domain 382
IBM Redbooks™ 382
WebSphere Studio Application Developer information 382
WebSphere Studio Application Developer online help 382
WebSphere Studio Application Developer Web site 382
WebSphere Developer Domain 383
IBM Redbooks 383

Notices 385
Trademarks and service marks 388

Index 389

Part 1. Concepts and architecture

© Copyright IBM Corp. 2000, 2003

Chapter 1. Overview

WebSphere Commerce software components

Before examining how the WebSphere Commerce Server functions, it is useful to look at the larger picture of the software components that relate to WebSphere Commerce. The following diagram shows a simplified view of these software products:

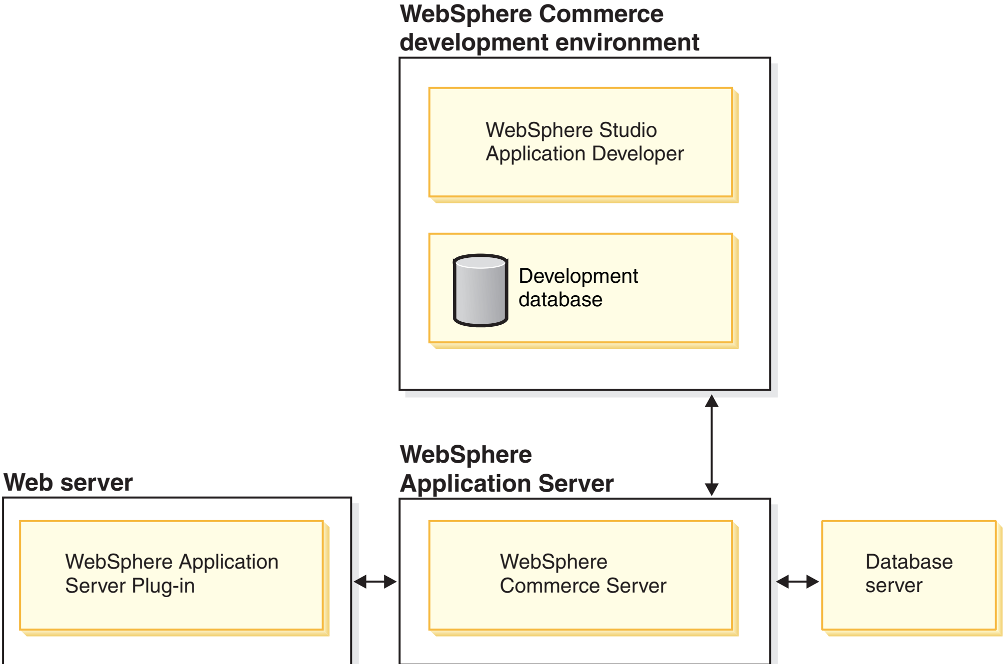


Figure 1.

The Web server is the first point of contact for incoming HTTP requests for your e-commerce application. In order to interface efficiently with the WebSphere Application Server, it uses the WebSphere Application Server plug-in.

The WebSphere Commerce Server runs within the WebSphere Application Server, allowing it to take advantage of many of the features of the application server. The database server holds most of your application's data, including product and shopper data. In general, extensions to your application are made by modifying or extending the code for the WebSphere

© Copyright IBM Corp. 2000, 2003

Commerce Server. In addition, you may have a need to store data that falls outside of the realm of the WebSphere Commerce database schema within your database.

Developers use WebSphere Studio Application Developer to perform the following tasks:

- create and customize store front assets such as JSP templates and HTML pages
- create new business logic in Java
- modify existing business logic in Java
- test code and store front assets

The WebSphere Commerce development environment uses a development database. Developers can use their preferred database tools (including WebSphere Studio Application Developer) to make database modifications.

WebSphere Commerce application architecture

Now that you have seen how the various software components related to WebSphere Commerce fit together, it is important to understand the application architecture. This will help you to understand which parts are foundation layers and which parts you can modify. The following diagram shows the various layers that comprise the application architecture:

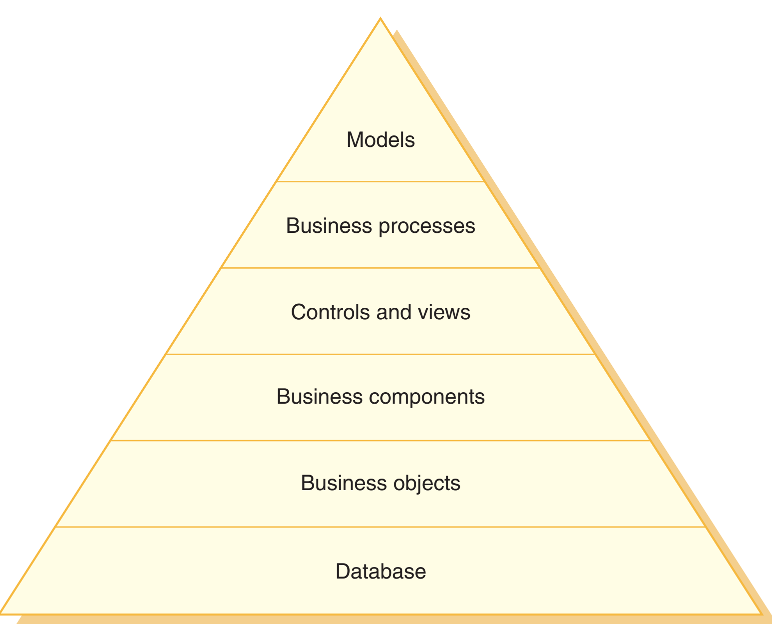


Figure 2.

Each layer of the application architecture is described below:

Database
WebSphere Commerce uses a database schema designed specifically for e-commerce applications and their data requirements. The following are examples of tables in this schema:

- USERS
- ORDERS
- INVENTORY

Business objects

Business objects represent entities within the commerce domain and encapsulate the data-centric logic required to extract or interpret information contained within the database. These entities comply with the Enterprise JavaBeans™ specification.

These entity beans act as an interface between the business components and the database. In addition, the entity beans are easier to comprehend than complex relationships between columns in database tables.

Business components
Business components are units of business logic. They perform coarse-grained procedural business logic. The logic is implemented

using the WebSphere Commerce model of controller commands and task commands. An example of this type of component is the OrderProcess controller command. This particular command encapsulates all of the business logic required to process a typical order. The e-commerce application calls the OrderProcess command, which in turn, calls several task commands to perform individual units of work. For example, individual task commands ensure that enough inventory is available to meet the requirements of the order, process the payment, update the status of the order and when the process has completed, decrement the inventory by the appropriate amount.

Controls and views

A Web controller determines the appropriate controller command implementation and view to be used. Implementations can be store specific.

Views display the results of commands and user actions. They are implemented using JSP templates. Examples of views include ProductDisplayView (returns a product page showing relevant information for the shopper's selected product) and OrderCancelView.

Business processes

Sets of business components and views together create workflow and siteflow processes that are known as business processes. Examples of business processes include:

Create an e-mail campaign

This business process includes the business components and views related to all steps involved in the process of creating e-mail campaigns.

Prepare an online catalog

This business process includes the business components and sub-processes related to creating an online catalog. This includes designing the catalog, loading catalog data, creating merchandising associations, and setting pricing information.

Models

When gathered together, the lower layers of the diagram make up e-commerce business models. One example of an e-commerce business model is the consumer direct model that is displayed in the FashionFlow sample store. Business Another example is the B2B direct model that is displayed in the ToolTech sample store.

WebSphere Commerce run-time architecture

The previous section introduced the application architecture, which depicts the various layers in the WebSphere Commerce application, from a business application point-of-view. This section describes how the run-time architecture is implemented.

The major components of the WebSphere Commerce run-time architecture are:

- Servlet engine
- Protocol listeners
- Adapter manager
- Adapters
- Web controller
- Commands
- Entity beans
- Data beans
- Data bean manager
- Display pages
- XML files

The interactions between WebSphere Commerce components are shown in the following diagram. More detail on each component can be found in subsequent sections.

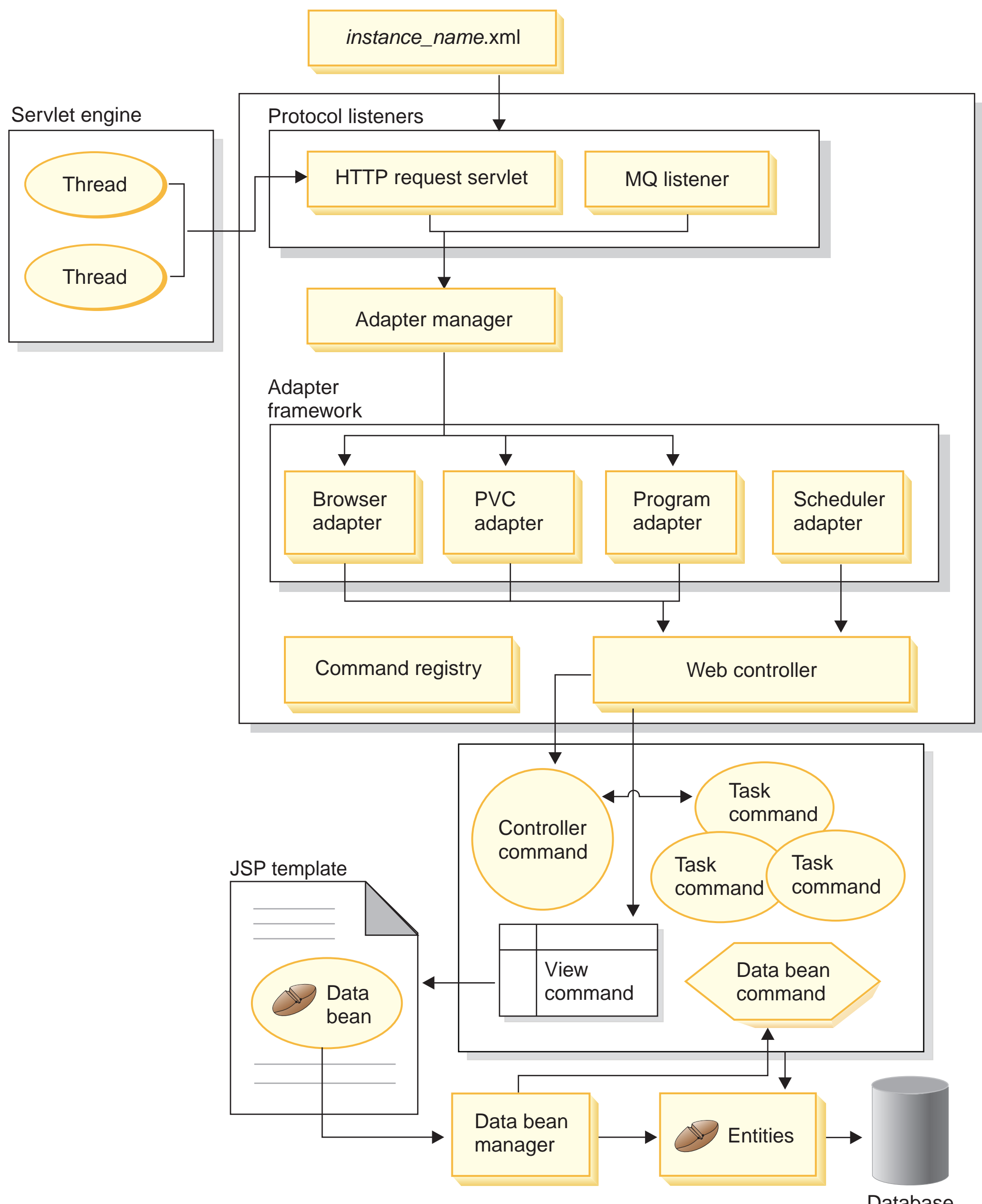


Figure 3.

Servlet engine

The servlet engine is the part of the WebSphere Application Server run-time environment that acts as a request dispatcher for inbound URL requests. The servlet engine manages a pool of threads to handle requests. Each inbound request is executed on a separate thread.

Protocol listeners

WebSphere Commerce commands can be invoked from various devices. Examples of devices that can invoke commands include:

- Typical Internet browsers
- Mobile phones using Internet browsers
- Business-to-business applications sending XML messages using MQSeries®
- Procurement systems sending requests using XML over HTTP
- The WebSphere Commerce scheduler that executes a background job

Devices can use a variety of communication protocols. A protocol listener is a run-time component that receives inbound requests from transports and then dispatches the requests to the appropriate adapters, based upon the protocol used. The protocol listeners include:

- Request servlet
- MQSeries listener

When the request servlet receives a URL request from the servlet engine, it passes the request to the adapter manager. The adapter manager then queries the adapter types to determine which adapter can process the request. Once the specific adapter is determined, the request is passed to the adapter.

When the request servlet is initialized, it reads the *instance_name*.xml configuration file (where *instance_name* is the name of the WebSphere Commerce instance). One of the configuration blocks in the XML file defines all of the adapters. The init() method of the request servlet initializes all defined adapters.

The MQSeries listener receives XML-based MQSeries messages from remote programs and dispatches the requests to the non-HTTP adapter manager.

The Job Scheduler does not require a protocol listener.

Adapter manager

The adapter manager determines which adapter is capable of handling the request and then forwards the request to that adapter.

Adapters

WebSphere Commerce adapters are device-specific components that perform processing functions before passing a request to the Web controller. Examples of processing tasks performed by an adapter include:

- Instructing the Web controller to process the request in a manner specific to the type of device. For example, a pervasive computing (PvC) device adapter can instruct the Web controller to ignore HTTPS checking in the original request.
- Transforming the message format of the inbound request into a set of properties that WebSphere Commerce commands can parse.
- Providing device-specific session persistence.

The following diagram shows the implementation class hierarchy for the WebSphere Commerce adapter framework.

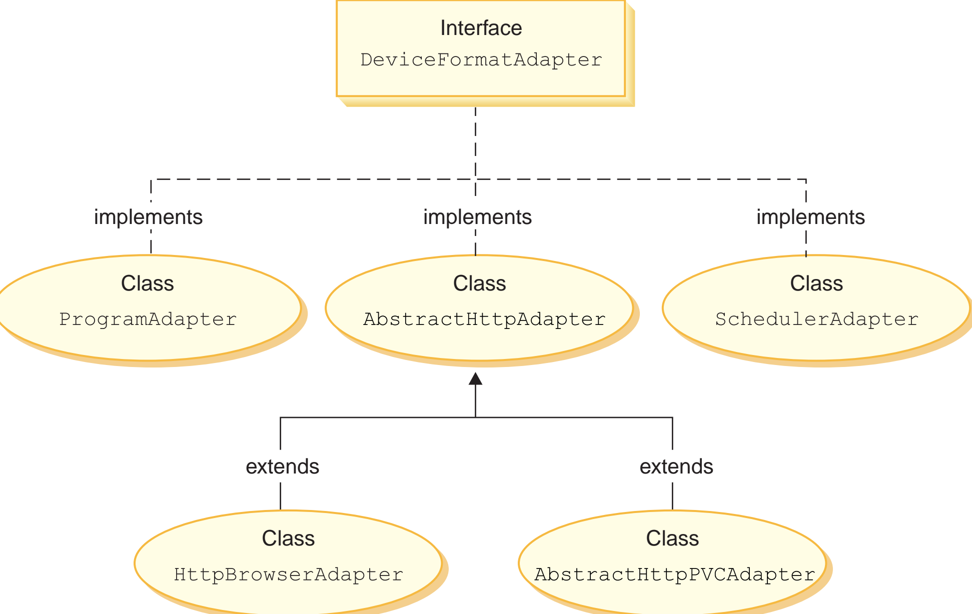


Figure 4.

As displayed in the preceding diagram, all adapters implement the DeviceFormatAdapter interface. The following are the adapters that are used by the WebSphere Commerce run-time environment:

Program adapter
The program adapter provides support for remote programs invoking WebSphere Commerce commands. The program adapter receives requests and uses a message mapper to convert the request into a

CommandProperty object. After the conversion, the program adapter uses the CommandProperty object and executes the request.

Scheduler adapter
The scheduler adapter provides support for WebSphere Commerce commands that are run as background jobs.

HTTP browser adapter
The HTTP browser adapter provides support for requests to invoke WebSphere Commerce commands that are received from HTTP browsers.

HTTP PvC adapter
This is an abstract adapter class that can be used to develop specific PvC device adapters. For example, if you needed to develop an adapter for a particular cellular phone application, you would extend from this adapter.

If required, the adapter framework can be extended in the following two ways:

- Create an adapter for a specific PvC device (for example, create an HttpIModePVCAdapterImpl class to provide support for i-mode devices). An adapter of this type must extend the AbstractHttpAdapterImpl class.
- Create a new adapter that connects to a new protocol listener. This new adapter must implement the DeviceFormatAdapter interface.

Web controller

The WebSphere Commerce Web controller is an application container that follows a design pattern similar to that of an EJB container. This container simplifies the role of commands, by providing such services as session management (based upon the session persistence established by the adapter), transaction control, access control and authentication.

The Web controller also plays a role in enforcing the programming model for the commerce application. For example, the programming model defines the types of commands that an application should write. Each type of command serves a specific purpose. Business logic must be implemented in controller commands and view logic must be implemented in view commands. The Web controller expects the controller command to return a view name. If a view name is not returned, an exception is thrown.

For HTTP requests, the Web controller performs the following tasks:

- Begins the transaction using the UserTransaction interface from the javax.transaction package.
- Gets session data from the adapter.

- Determines whether the user must be logged on before invoking the command. If required, it redirects the user's browser to a logon URL.
- Checks if secure HTTPS is required for the URL. If it is required but the current request is not using HTTPS, it redirects the Web browser to an HTTPS URL.
- Invokes the controller command and passes it the command context and input properties objects.
- If a transaction rollback exception occurs and the controller command can be retried, it retries the controller command.
- A controller command normally returns a view name when there is a view command to be sent back to the client. The Web controller invokes the view command for the corresponding view. There are a number of ways to form a response view. These include redirecting to a different URL, forwarding to a JSP template or writing an HTML document to the response object.
- Saves the session data.
- Commits the session data.
- Commits the current transaction if it is successful.
- Rolls back the current transaction in case of failure (depending upon circumstances).

Commands

WebSphere Commerce commands are beans that contain the programming logic associated with handling a particular request.

There are four main types of WebSphere Commerce commands:

Controller command
A controller command encapsulates the logic related to a particular business process. Examples of controller commands include the OrderProcessCmd command for order processing and the LogonCmd that allows users to log on. In general, a controller command contains the control statements (for example, if, then, else) and invokes task commands to perform individual tasks in the business process. Upon completion, a controller command returns a view name. The Web controller then determines the appropriate implementation class for the view command and executes the view command.

Task command
A task command implements a specific unit of application logic. In general, a controller command and a set of task commands together implement the application logic for a URL request. A task command is executed in the same container as the controller command.

Data bean command
A data bean command is invoked by the data bean manager when a

data bean is instantiated. The primary function of a data bean command is to populate the data bean with data.

View command
A view command composes a view as a response to a client request. There are three types of view commands:

Redirect view command
This view command sends the view using a redirect protocol, such as the URL redirect. A controller command should return a view command in this view type to return a view using a redirect protocol. When a redirect protocol is used, it changes the URL stacks in the browser. When a reload key is entered, the redirected URL executes instead of the original URL.

Direct view command
This view command sends the response view directly to the client.

Forward view command
This view command forwards the view request to another Web component, such as a JSP template.

There are three ways in which a view command can be invoked:

- A controller command specifies a view command name when the request has successfully completed.
- A client requests a view directly.
- A command detects an error and an error task must be executed to process the error. The command throws an exception with a view command name. When the exception propagates to the Web controller, it executes the error view command and returns the response to the client.

WebSphere Commerce entity beans

Entity beans are the persistent, transactional commerce objects provided by WebSphere Commerce. If you are familiar with the commerce domain, entity beans represent WebSphere Commerce data in an intuitive way. That is, rather than having to understand the whole the database schema, you can access data from an entity bean which more closely models concepts and objects in the commerce domain. You can extend existing entity beans. In addition, for your own application-specific business requirements, you can deploy entirely new entity beans.

Entity beans are implemented according to the Enterprise JavaBeans (EJB) component model.

For more information about entity beans, refer to "Implementation of WebSphere Commerce entity beans" on page 47.

Data beans

Data beans are Java beans that are primarily used by Web designers. Most commonly, they provide access to a WebSphere Commerce entity. A Web designer can place these beans on a JSP template, allowing dynamic information to be populated on the page at display time. The Web designer need only understand what data the bean can provide and what data the bean requires as input. Consistent with the theme of separating display from business logic, there is no need for the Web designer to understand how the bean works.

Data bean manager

WebSphere Commerce data beans inserted into JSP templates allow for the inclusion of dynamic content in the page. The data bean manager activates the data bean so that its values are populated when the following line of code is inserted into the page:

```
com.ibm.commerce.beans.DataBeanManager.activate(data_bean, request)
```

where *data_bean* is the data bean to be activated and *request* is an HTTPServletRequest object.

JavaServer Pages templates

JSP templates are specialized servlets that are typically used for display purposes. Upon completion of a URL request, the Web controller invokes a view command that invokes a JSP template. A client can also invoke a JSP template directly from the browser without an associated command. In this case, the URL for the JSP template must include the request servlet in its path, so that all of the data beans required by a JSP template can be activated within a single transaction. The request servlet can forward a URL request to a JSP template and execute the JSP template within a single transaction.

The data bean manager rejects any URL for a JSP template that does not include the request servlet in its path. For more information about protecting JSP templates and other resources, refer to Chapter 4, "Access control," on page 89.

instance_name.xml configuration file

The *instance_name*.xml configuration file (where *instance_name* is the name of the WebSphere Commerce instance) sets configuration information for the WebSphere Commerce instance. It is read when the request servlet is initialized.

Summary for a request

This section provides a summary of the interaction flow between components when forming a response to a request.

A description of each of the steps follows the diagram.

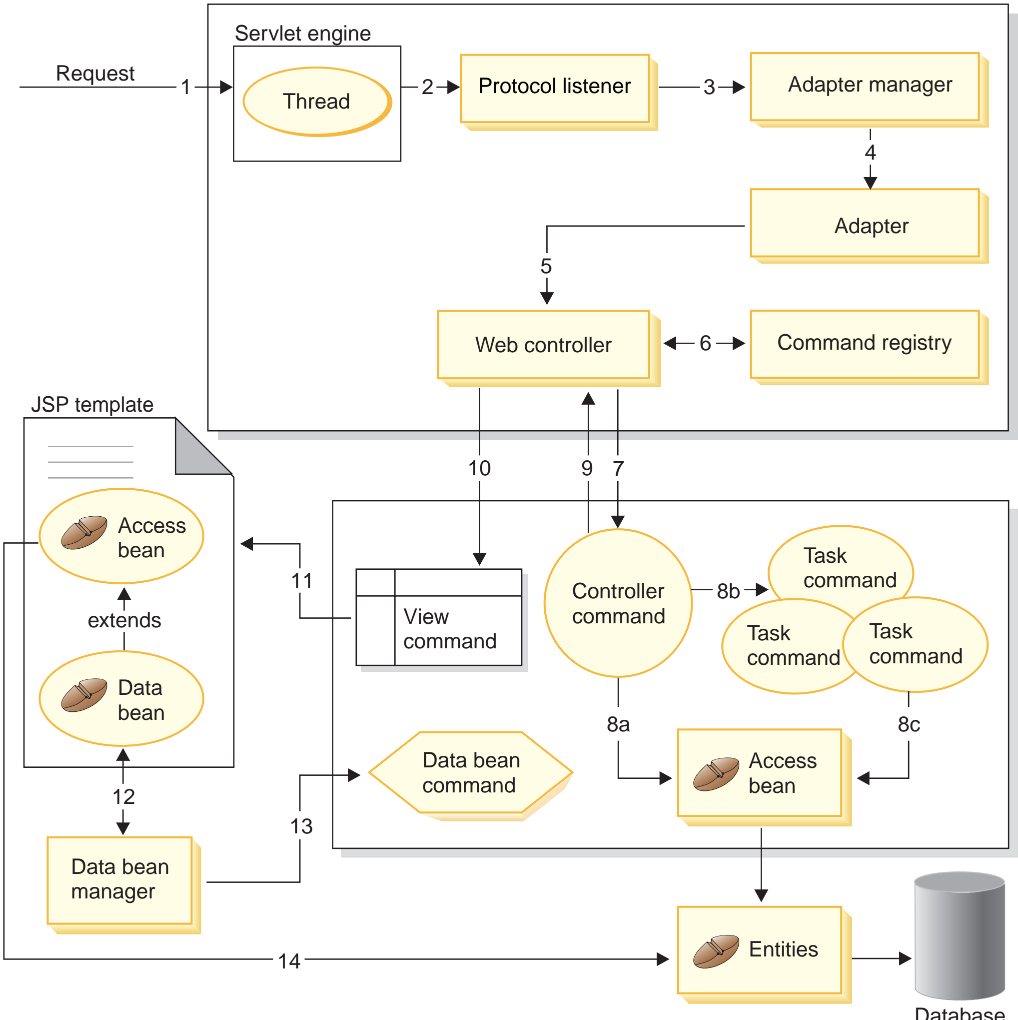


Figure 5.

The following information corresponds to the preceding diagram.

1. The request is directed to the servlet engine by the WebSphere Application Server plug-in.

2. The request is executed in its own thread. The servlet engine dispatches the request to a protocol listener. The protocol listener can be the HTTP request servlet or the MQ Listener.
3. The protocol listener passes the request to the adapter manager.
4. The adapter manger determines which adapter is capable of handling the request and then forwards the request to the appropriate adapter. For example, if the request came from an Internet browser, the adapter manager forwards the request to the HTTP browser adapter.
5. The adapter passes the request to the Web controller.
6. The Web controller determines which command to invoke, by querying the command registry.
7. Assuming that the request requires the use of a controller command, the Web controller invokes the appropriate controller command.
8. Once a controller command begins execution, there are multiple possible paths:
    a. The controller command can access the database using an access bean and its corresponding entity bean.
    b. The controller command can invoke one or more task commands. Then task commands can access the database, using access beans and their corresponding entity beans (shown in 8c in figure 5).
9. Upon completion, the controller command returns a view name to the Web controller.
10. The Web controller looks up the view name in the VIEWREG table. It invokes the view command implementation that is registered for the device type of the requester.
11. The view command forwards the request to a JSP template.
12. Within the JSP template, a data bean is required to retrieve dynamic information from the database. The data bean manager activates the data bean.
13. The data bean manager invokes a data bean command, if required.
14. The access bean from which the data bean is extended accesses the database using its corresponding entity bean.

Part 2. Programming model

© Copyright IBM Corp. 2000, 2003

Chapter 2. Design patterns

A variety of design patterns and mechanisms are used to develop the WebSphere Commerce framework. WebSphere Commerce provides a high-level design pattern to which each WebSphere Commerce application should adhere. The following design patterns are discussed in this chapter:

- The model-view-controller design pattern
- The command design pattern
- The display design pattern

Model-View-Controller design pattern

The model-view-controller (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

The *model* (for example, the data information) contains only the pure application data; it contains no logic describing how to present the data to a user.

The *view* (for example, the presentation information) presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.

Finally, the *controller* (for example, the control information) exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.

Most applications today follow this pattern, many with slight variations. For example, some applications combine the view and the controller into one class because they are already very tightly coupled. All of the variations strongly encourage separation of data and its presentation. This not only makes the structure of an application simpler, it also enables code reuse.

Since there are many publications describing the pattern, as well as numerous samples, this document does not describe the pattern in great detail.

© Copyright IBM Corp. 2000, 2003

The following diagram shows how the MVC design pattern applies to WebSphere Commerce.

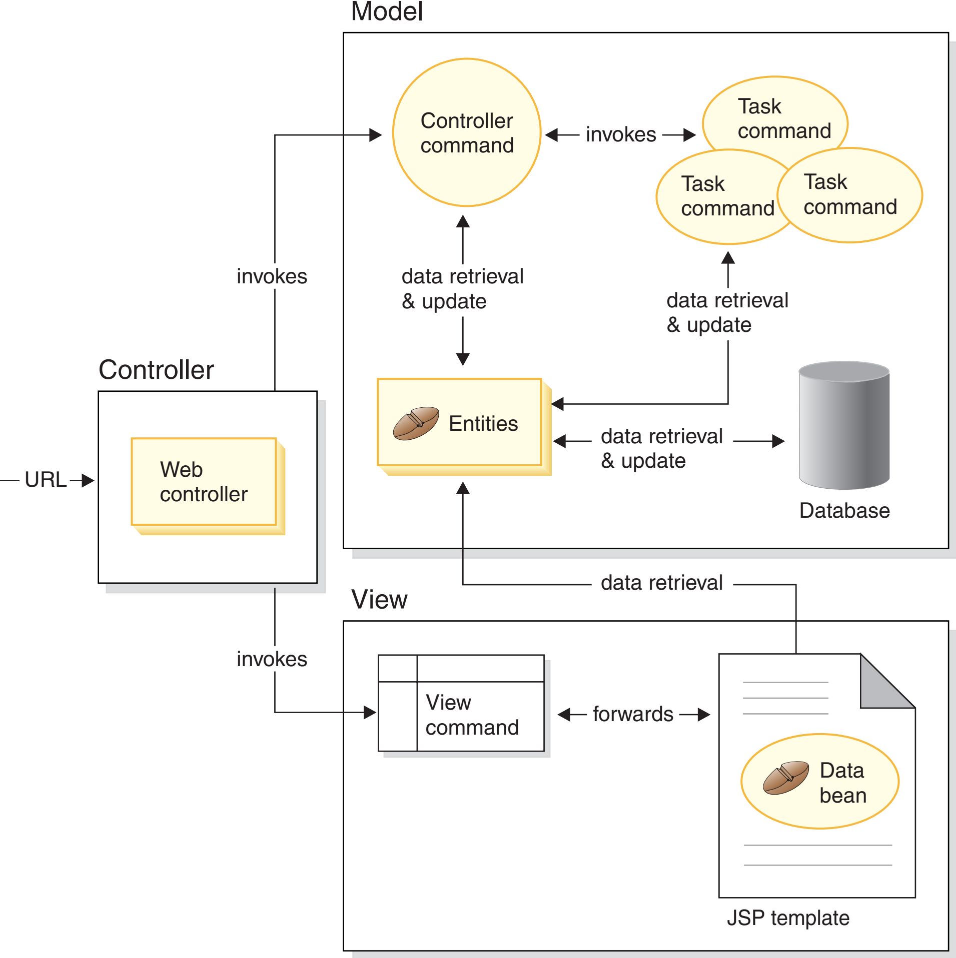


Figure 6.

Command design pattern

The WebSphere Commerce Server accepts requests from browser-based thin-client applications, as well as from other remote applications. For example, a request may come from a remote procurement system, or from another commerce server.

All requests, in their variety of formats, are translated into a common format by the adapters that make up the adapter framework. Once the requests are in this common format, they can be understood by WebSphere Commerce commands.

Commands are beans that perform business logic. They represent procedural logic either in the form of high-level process logic or discrete business logic tasks. A process-based command acts as a controller that spans multiple entities and other commands, while a task command performs a specific task and may only access a single object.

Command framework

Command beans follow a specific design pattern. Every command includes both an interface class (for example, CategoryDisplayCmd) and an implementation class (for example, CategoryDisplayCmdImpl). From a caller's perspective, the invocation logic involves setting input properties, invoking an execute() method, and retrieving output properties.

From the perspective of the command implementer, commands follow the WebSphere command framework, which implements the standard command design pattern allowing a level of indirection between the caller and the implementation. The key mechanisms enabled within this level of indirection include:

1. The ability to invoke an access control policy manager that determines if the user is allowed to invoke the command.
2. The ability to execute a different command implementation for different stores, based upon the store identifier.
3. The ability to execute a different view implementation based upon the device type of the requester.

The following diagram shows a conceptual overview of the interfaces for the four main types of commands:

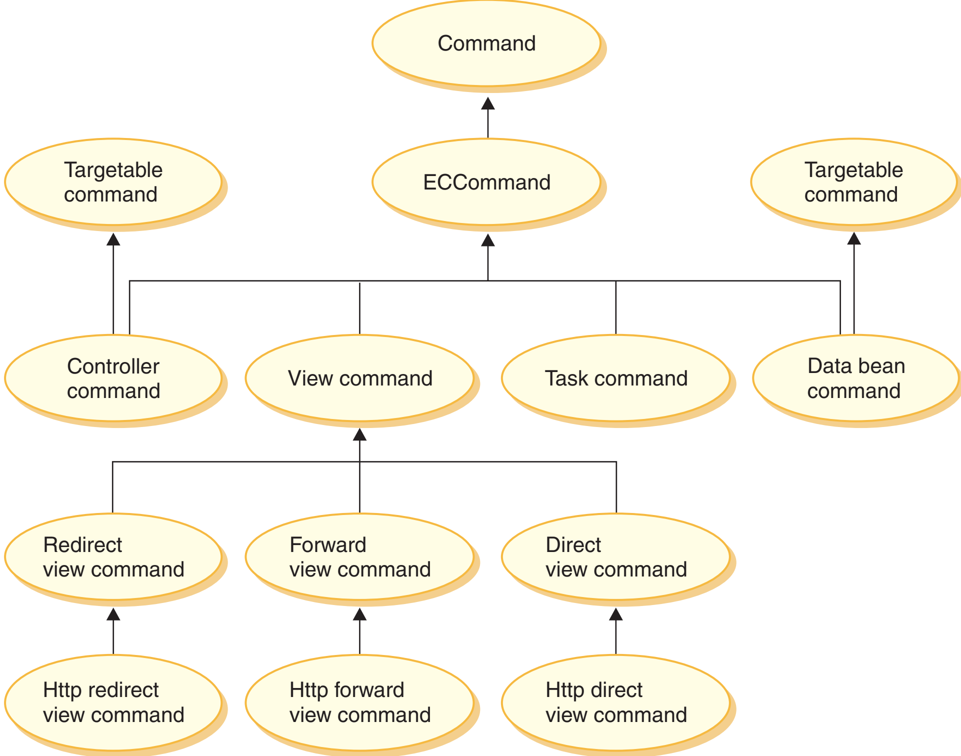


Figure 7.

Controller command

A controller command encapsulates the logic related to a particular business process. Examples of controller commands include the *OrderProcessCmd* command for order processing and the LogonCmd that allows users to log on. In general, a controller command contains the control statements (for example, if, then, else) and invokes task commands to perform individual tasks in the business process. Upon completion, a controller command returns a view name. Based upon the view name, the store identifier, and the device type, the Web controller then determines the appropriate implementation class for the view command and executes the view command.

While a controller command is a targetable command, only the local target is supported.

Task command

A task command implements a specific unit of application logic. In general, a controller command and a set of task commands together implement the application logic for a URL request. A task command is executed in the same container as the controller command.

Data bean command

A data bean command is invoked by a JSP page when a data bean is instantiated. The primary function of a data bean command is to populate the fields of the data bean.

While a data bean command is a targetable command, only the local target is supported.

View command

A view command composes a view as a response to a client request. There are three ways in which a view command can be invoked:

- A controller command specifies a view command name on successful completion of the request.
- A client can request a view directly.
- A controller or task command detects an error and decides that an error task must be executed to process the error and throws an exception with a view command name. When the exception propagates to the Web controller, it executes the view command and returns the response to the client.

There are three types of view commands:

Redirect view command

This view command sends the view using a redirect protocol, such as the URL redirect. A controller command should return a view command of this view type when a redirect protocol is required. When a redirect protocol is used, it changes the URL stacks in the browser. When a reload key is entered, the redirected URL executes instead of the original URL.

Direct view command

This view command sends the response view directly to the client.

Forward view command

This view command forwards the view request to another Web component, such as a JSP template.

Command factory

In order to create new command objects, the caller of the command uses the *command factory*. The command factory is a bean that is used to instantiate commands. It is based on the factory design pattern, which defers instantiation of an object away from the invoking class, to the factory class that understands which implementation class to instantiate.

The factory provides a smart way to instantiate new objects. In this case, the command factory provides a way to determine the correct implementation class when creating a new command object, based upon the individual store. The command interface name and the particular store identifier are passed into the new command object, upon instantiation.

There are two ways for the implementation class of a command to be specified. A default implementation class can be specified directly in the code for the command interface, using the `defaultCommandClassName` variable. For example, the following code exists in the CategoryDisplayCmd interface:

```
String defaultCommandClassName =
     "com.ibm.commerce.catalog.commands.CategoryDisplayCmdImpl"
```

The second way to specify the implementation class is to use the WebSphere Commerce command registry. The command registry should always be used when the implementation class varies from one store to another. More information about the command registry can be found on page 28.

In the case where a default implementation class is specified in the code for the interface and a different implementation class is specified in the command registry, the command registry takes precedence.

The syntax for using the command factory is as follows:

```
cmd = CommandFactory.createCommand(interfaceName, storeId)
```

where *`interfaceName`* is the interface name for the new command bean and *`storeId`* is identifier of the store for which the command should be implemented. Typically, the store ID can be retrieved by using the commandContext.getStoreId() method.

Note: The syntax for using the command factory to create business policy commands is different from the preceding code snippet. For more information about using the command factory to create business policy commands, refer to "Invoking the new business policy" on page 184.

Nested controller commands

You will most often use the command factory to create instances of task commands, however, it can also be used within one controller command to create an instance of another controller command. In other words, it is used when calling one controller command from within another.

The syntax for instantiating task commands and controller commands is the same. That is, you specify the name of the command's interface and the store ID in both scenarios.

If you nest one controller command within another, note the following points:

- Once you have instantiated the nested command, call its setCommandContext method and pass in the current command context. Note that if you are passing a different set of request properties to the nested command and these parameters will affect the command context, you should clone the command context before instantiating the nested command. This will preserve the command context information for the outer command.
- Ideally, call the setRequestProperties method of the nested command and pass a TypedProperties object containing input properties. Otherwise, you can use the individual setter methods defined on the interface of the command to set the required properties.
- After input properties have been set, call the execute method of the nested command.
- Since all controller commands must return a view when processing has completed, the outer command can do nothing with the view returned by the nested command.
- The nested command will be executed within the transaction scope of the outer command.

Consider the following code snippet as an example of a nested controller command. This example shows a method in the outer command and how it can use the command factory to instantiate a second controller command.

```
yourControllerCmd ctrlCmd = null;

public void processAndCallOtherCommand()
   throws ECException
   {
      ctrlCmd = (yourControllerCmd)CommandFactory.createCommand(
         com.yourcompany.commands.yourControllerCmd, this.getStoreId());
      ctrlCmd.setCommandContext(this.getCommandContext());
      ctrlCmd.setRequestProperties(this.getRequestProperties());
      ctrlCmd.execute();
    }
```

As another example, consider the case where the nested command is being executed for a different store than the first. In this case, the command context from the outer command must be preserved so that it does not get overwritten by the inner command.

```
// Make a clone to preserve the command context of the outer commmand
CommandContext cloneCmdCtx = (CommandContext)this.getCommandContext().clone();

//Now pass in a new set of request properties to the cloned command context
cloneCmdCtx.setRequestProperties(reqProp);

yourControllerCmd ctrlCmd = null;

public void processAndCallOtherCommandForOtherStore(int aStoreId)
   throws ECException
```

```
{
   ctrlCmd = (yourControllerCmd)CommandFactory.createCommand(
      com.yourcompany.commands.yourControllerCmd, aStoreId;
   ctrlCmd.setCommandContext(cloneCmdCtx);
   ctrlCmd.setRequestProperties(reqProp);
   ctrlCmd.execute();
 }
```

Command flow

This section provides an overview of the logical flow between commands and the WebSphere Commerce database. The following diagram and descriptions depict this flow.

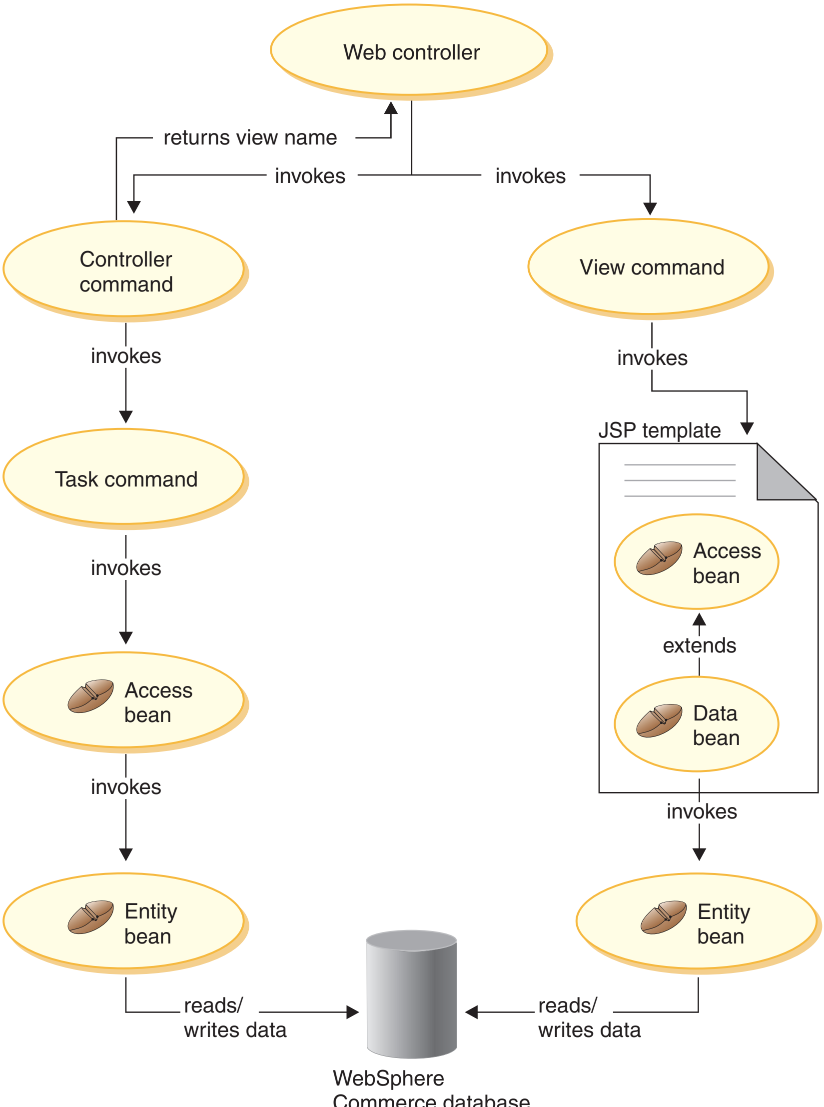


Figure 8.

When the Web controller receives a request, it determines whether the request requires the invocation of a controller command or a view command. In either case, the Web controller also determines the implementation class for the command, and then invokes it.

First examine the left side of the diagram. Since controller commands encapsulate the logic for a business process, they frequently invoke individual task commands to perform specific units of work in the business process.

Access beans are invoked when information in the database must be retrieved or updated. Either a task or controller command can invoke access beans. Requests then flow from access beans to entity beans that can read from, and write to, the WebSphere Commerce database.

Now examine the right side of the diagram. A view command is invoked by the Web controller, either when a controller command has completed processing and it returns the name of a view command to invoke, or when an error occurs and an error view must be displayed.

View commands typically invoke a JSP template to display the response to the client. Within the JSP template, data beans are used to populate dynamic information onto the page. Data beans are activated by the data bean manager. The data bean (which extends from an access bean) invokes its corresponding entity bean. When accessed indirectly from a JSP template, an entity bean typically retrieves information from the database (rather than writing information to the database).

Command registration framework

WebSphere Commerce controller and task commands are registered in the command registry. The following three tables comprise the command registry:

- URLREG
- CMDREG
- VIEWREG

Note: Business This section does not apply to the registration of business policy commands. For information about registering new business policy commands, refer to "Registering the new business policy and business policy command" on page 168.

URLREG table

The URLREG table maps URIs (Universal Resource Indicator) to controller command interfaces. URIs provide a simple and extensible mechanism for resource identification. A URI is a relatively short string of characters used to identify an abstract or physical resource. In WebSphere Commerce, the URI contains only command information. In the following URL, the URI section is shown in bold:

```
http://hostname/webapp/wcs/stores/servlet/StoreCatalogDisplay?
    storeId=store_Id&langId=-1
```

While there is a one-to-one mapping between a URI and an interface name, each store can specify whether HTTPS or AUTHENTICATION is required for the command. For each inbound URL request, the Web controller looks up the

interface name for the controller command and then uses that name to determine the correct implementation class, as registered in the CMDREG table.

The following table describes information contained in the URLREG database table.

Column name	**Description**	**Comments**
URL	URI name	For example `MyNewCommand` or `com.ibm.commerce.catalog.commands.ProductDisplayCmd`
STOREENT_ID	Store entity identifier	This can be set to `0` to use the command for all stores, or to a unique store identifier to indicate that the command is used only for a particular store.
INTERFACENAME	Controller command interface name	For example `com.ibm.commerce.catalog.commands.ProductDisplayCmdImpl`
HTTPS	Secure HTTP required for this URL request	Use `1` when HTTPS is required and `0` when it is not.
DESCRIPTION	Description of URI	For example, `This command is used for testing purposes`.
AUTHENTICATED	User log on is required for this URL request	Use `1` when authentication is required and `0` when it is not.
INTERNAL	Indicates whether or not the command is internal to WebSphere Commerce	Use `1` when the command is internal and `0` when it is external.

When the Web controller receives a URL request, it retrieves the interface name for the requested controller command and uses it to look up the implementation class name from the CMDREG table. It also determines if HTTPS is required for the URL request by checking the HTTPS column in the URLREG table.

Only commands that are invoked by way of URL requests need to be registered in the URLREG table. Therefore, only controller commands must be registered here, not task or view commands.

The following SQL statement creates an entry for MyNewControllerCommand which is used by a particular store (whose store identifier is `5`):

```
insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS,
DESCRIPTION, AUTHENTICATED) values ('MyNewControllerCommand',
5,'com.ibm.commerce.commands.MyNewControllerCommand',0,
'This is a test command.',null)
```

The generic syntax for the insert statement is as follows:

```
insert into table_name (column_name1,column_name2, ... ,column_namen)
values (column1_value,column2_value,...,columnn_value)
```

String values should be enclosed in single quotes.

CMDREG table

CMDREG is the command registration table. This table provides a mechanism for mapping the command interface to its implementation class. Multiple implementations of an interface allow for command customization on a per store basis.

Only controller commands and task commands are registered in the CMDREG table. View commands are registered in the VIEWREG table.

The following describes information contained in the CMDREG database table.

Column name	Description	Comments
STOREENT_ID	Store entity identifier	This can be set to `0` to use the command for all stores, or to a unique store identifier to indicate that the command is used only for a particular store.
INTERFACENAME	Command interface name	This defines the interface; use the same name as you did in the URLREG table.
DESCRIPTION	Description of this command	For example, `This command is used for testing purposes`.
CLASSNAME	Command implementation class name	Typically the interface name with ″Impl″ appended to end.
PROPERTIES	Default name-value pairs set as input properties to the command	Format is same as URL query string. For example ″parm1=val1&parm2=val2″
LASTUPDATE	Last update on this command entry	
TARGET	Command target name. This is where the command is actually executed.	Only local target is supported.

In general, when you create a new controller or task command, you should create corresponding entry in the CMDREG table. For example, the following SQL statement creates an entry for MyNewCommand which is used by a particular store (whose store identifier is 5):

```
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION, CLASSNAME,
PROPERTIES, LASTUPDATE, TARGET) values
(5,'com.mycompany.commands.MyNewCommand', 'This is a test command',
'com.mycompany.commands.MyNewCommandImpl',
'myDefaulParm1=myDefaultVal1', '0000-12-01', 'Local')
```

String values should be enclosed in single quotes.

If the command you are writing always uses the same implementation class, you do not necessarily have to register the command in the CMDREG table. In this case, you can use the defaultCommandClassName attribute in the interface to specify the implementation class. For example, in the code for the interface, you would include the following:

```
String defaultCommandClassName =
     "com.ibm.commerce.command.MyNewCommandImpl"
```

If you specify the implementation class in this manner, you cannot pass default properties to the implementation class and the same implementation class must be used for all stores.

Example of a registered controller command

Consider a scenario in which your site has two stores: StoreA and StoreB. Each store has different security requirements for the MyUrl controller command as well as different implementations of the command. This section shows how the command registry is used to enable this customization.

The following table shows the entries for StoreA and StoreB in the URLREG table:

Column name	Entry for StoreA	Entry for StoreB
URL	MyUrl	MyUrl
STOREENT_ID	11	22
INTERFACENAME	com.ibm.commerce. mycommands.myUrl	com.ibm.commerce. mycommands.myUrl
HTTPS	1	1
DESCRIPTION	Example entry in the URLREG table.	Example entry in the URLREG table.
AUTHENTICATED	1	0
INTERNAL	null	null

Note: The spaces in values for INTERFACENAME are for display purposes only. Each value is actually one continuous string.

Based upon entries in the URLREG table, the Web controller determines that the interface name for the MyURL URI is com.ibm.commerce.mycommands.MyUrl. It also determines that StoreA requires the command to be executed using both HTTPS and authentication, but StoreB requires HTTPS only. The values for HTTPS and authentication are used by the Web controller, not by the interface.

The following diagram shows this flow:

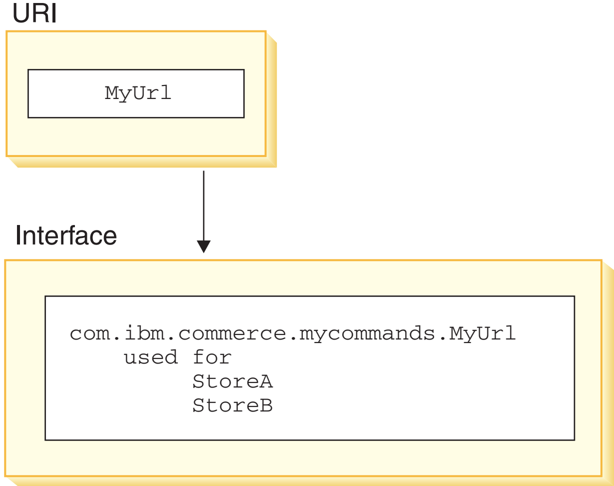


Figure 9.

The following table shows the entries in the CMDREG table. Only columns required for the purpose of this example are displayed:

Column name	Entry for StoreA	Entry for StoreB
STOREENT_ID	11	22
INTERFACENAME	com.ibm.commerce. mycommands.myUrl	com.ibm.commerce. mycommands.myUrl
CLASSNAME	com.ibm.commerce. mycommands. myUrlStoreAImpl	com.ibm.commerce. mycommands.myUrlStoreBImpl

Note: The spaces in values for INTERFACENAME and CLASSNAME are for display purposes only. Each value is actually one continuous string.

Based upon entries in the CMDREG table, the Web controller determines that

for StoreA, the implementation class for the com.ibm.commerce.mycommands.MyUrl interface is com.ibm.commerce.mycommands.MyUrlStoreAImpl. It also determines for StoreB, the implementation class for the same interface is com.ibm.commerce.mycommands.MyUrlStoreBImpl. The following diagram shows this flow:

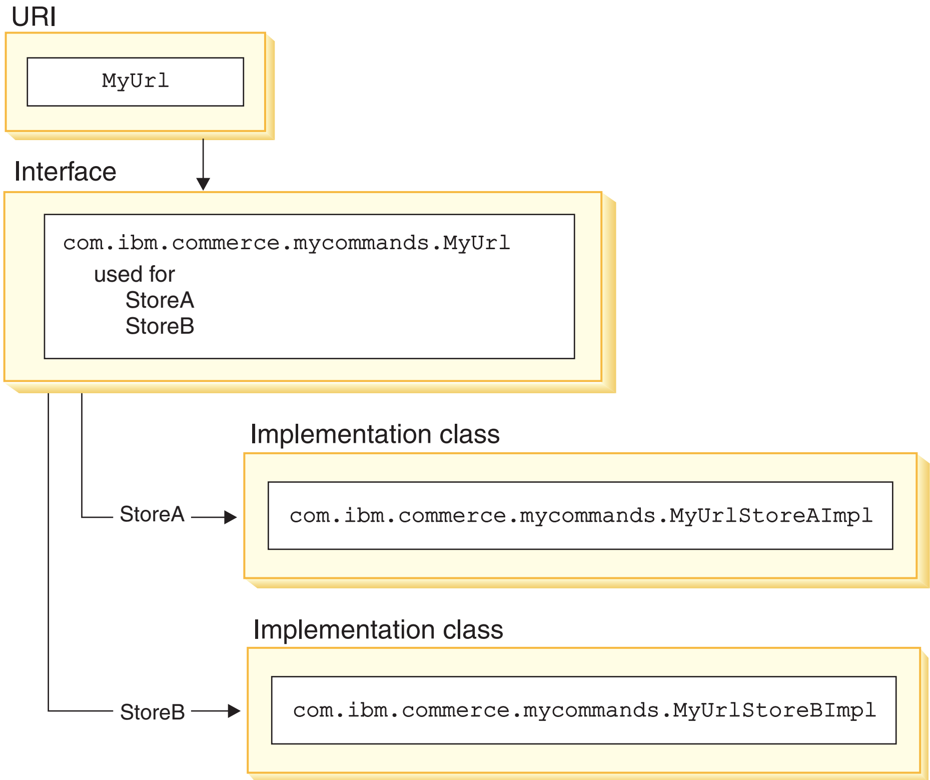


Figure 10.

VIEWREG table

The VIEWREG table allows registration of device-specific and store-specific view implementations. Using this table, multiple implementations of a view can be registered. The command framework is then capable of returning different views to various devices.

When a view name is returned from a controller command or specified in an exception, the Web controller determines the view implementation from the VIEWREG table. Multiple view names can be mapped to the same implementation class.

Column name	Description	Comments
VIEWNAME	View name	For example, `AddressForm`
DEVICEFMT_ID	Device type identifier	Available options include: • BROWSER (default value) • I_MODE • E-mail • MQXML • MQNC
STOREENT_ID	Store entity identifier	This can be set to `0` to use the command for all stores, or to a unique store identifier to indicate that the command is used only for a particular store.
INTERFACENAME	View command interface name	Default options are ForwardView, DirectView and RedirectView.
CLASSNAME	View command implementation class name	Can use the default implementation.
PROPERTIES	Default name-value pairs set as input properties to the command	If the same page is always displayed, set the JSP file name in this property (`docname=`*`jsp_name`*`.jsp`). If the same JSP template is used for all stores, set `storeDir=no` to prevent a store specific directory from being used. If a generic user can invoke the command, set `isGeneric=true`.
DESCRIPTION	Description of this command	
HTTPS	Secure HTTP required for this URL request	Use 1 when HTTPS is required and `0` when it is not.
LASTUPDATE	Last update on this entry	
INTERNAL	Indicates whether or not the command is internal to WebSphere Commerce	Use 1 when the command is internal and `0` when it is external.

When you create a new view, you may need to create a corresponding entry in the VIEWREG table. If one of the following conditions is met, the view must be registered in the VIEWREG table:

- The view is executed under access control

- There are multiple implementations of the view command
- Properties are set in the PROPERTIES column

Registered views can either be accessed through the view registry using the view name, or directly by using the actual display file name. Views that are not registered in the VIEWREG table can only be accessed when a client uses the actual display file name.

Consider the example of a view named *MyView*, with the VIEWREG entry as follows:

Column name	**Entry**
VIEWNAME	`MyView`
DEVICEFMT_ID	`BROWSER`
STOREENT_ID	`0`
INTERFACENAME	`com.ibm.commerce.commands.ForwardViewCommand`
CLASSNAME	`com.ibm.commerce.commands.HTTPForwardViewCommandImp`
PROPERTIES	`docname=MyView.jsp`
DESCRIPTION	`An example for calling a JSP template using either the view name or directly from a URL.`
HTTPS	`0`
LASTUPDATE	`2000–11–30`
INTERNAL	`0`

Since MyView is a registered view, a client can access the view either by using the view name, or by substituting the actual display file name for the view name. Using the view name, a sample URL is:

http://*hostname*.com/webapp/wcs/stores/servlet/MyView

and using the file name, a sample URL is:

http://*hostname*.com/webapp/wcs/stores/servlet/MyView.jsp

If there is a possibility that a client will invoke a registered view directly (using the display file name), you must register the view using the same name for the view as the actual display file name, as shown in this example (MyView and MyView.jsp).

A view that is not registered in the table can only be invoked using the display file name. Therefore, if there is an unregistered view that uses the file MyUnregisteredView.jsp, the URL to access this view is as follows:

```
http://hostname.com/webapp/wcs/stores/servlet/MyUnregisteredView.jsp
```

The following example SQL statement creates an entry for *MyNewView* which is used by one particular store:

```
insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID,
INTERFACENAME, CLASSNAME, PROPERTIES, DESCRIPTION,HTTPS, LASTUPDATE,
INTERNAL) values ('MyNewView', -1, 5,
'com.ibm.commerce.command.ForwardViewCommand',
'com.ibm.commmerce.command.HttpForwardViewCommandImpl',
'docname=MyNewView.jsp', 'A test view.', 0, '0000-12-01', 0)
```

The following table provides another sample VIEWREG table with key information:

VIEW NAME	INTERFACE - NAME	CLASSNAME	PROPERTIES
ProductDisplay View	Forward View Command	HttpForwardView CommandImpl	docname= UserArea/ ServiceSection/ InterestItemList Subsection/ WishListDisplay.jsp
Generic Application Error	Forward View Command	HttpForwardView CommandImpl	docname =Generic Application Error.jsp&storeDir=no
GenericSystem Error	Forward View Command	HttpForwardView CommandImpl	docname = Generic System Error.jsp &storeDir=no
LogonForm	Forward View Command	HttpForwardView CommandImpl	docname = LoginForm.jsp &generic=true &storeDir=no

Note: Any spaces in the values for VIEWNAME, INTERFACENAME, CLASSNAME and PROPERTIES are for display purposes only. Each value is actually one continuous string. The hyphens in the column names are also for display purposes.

The preceding table illustrates the following scenarios:

- The *ProductDisplayView* view name is returned to the Web controller by a controller command (ProductDisplay in this case). The Web controller determines the view command interface and class names using the ProductDisplayView view command name and its device identifier. A view command can have different implementation classes for different stores and device identifiers. The interface name, however, should remain the same, since it defines the view command type.

- If a controller or task command throws an ECApplication exception for a bad user parameter, the following may occur:
  - If there is a view specified within the controller command that should be called in the case of an application exception, the entry for that view is retrieved from the VIEWREG table and processed accordingly.
  - If a view is not specified, the GenericApplicationError command is called and the JSP template registered in the database is displayed. Using the preceding table as an example, this would result in the display of the `GenericApplicationError.jsp` template.
- If a controller or task command throws an ECSystem exception for a system exception, the following may occur:
  - If there is a view specified within the controller command that should be called in the case of a system exception, the entry for that view is retrieved from the VIEWREG table and processed accordingly.
  - If a view is not specified, the GenericSystemError command is called and the JSP template registered in the database is displayed. Using the preceding table as an example, this would result in the display of the `GenericSystemError.jsp` template.
- Browser clients can invoke the logon page by entering the logon URL. Since the storeDir property is set to "no", store-specific information is not included in the path for the JSP template. Hence, the same logon page is displayed for customers at all stores.

Display design pattern

Display pages return a response to a client. Typically, display pages are implemented as JSP templates (the recommended method), however, they can be written directly as servlets.

In order to support multiple device types, a URL access to a view command should use the view name, not the name of the actual JSP file.

The main rationale behind this level of indirection is that the JSP template represents a view. The ability to select the appropriate view (for example, based on locale, device type, or other data in the request context) is highly desirable, especially since a single request often has multiple possible views. Consider the example of two shoppers requesting the home page of a store, one shopper using a typical Web browser and the other using a cellular phone. Clearly, the same home page should not be displayed to each shopper. It is the Web controller's responsibility to accept the request, then based upon information in the command registration framework, determine the view that each shopper receives.

JSP templates and data beans

A data bean is a Java bean that is used within a JSP template to provide dynamic content. A data bean normally provides a simple representation of a WebSphere Commerce entity bean. The data bean encapsulates properties that can be retrieved from or set within the entity bean. As such, the data bean simplifies the task of incorporating dynamic data into JSP templates.

A data bean has a *BeanInfo* class that defines the properties that can be used on the display page. The BeanInfo class also enables the use of data beans in multicultural sites by providing property names in all supported languages of WebSphere Commerce.

A data bean is activated by the following call:

```
com.ibm.commerce.beans.DataBeanManager.activate(data_bean, request)
```

where *data_bean* is the data bean to be activated and *request* is an HTTPServletRequest object.

Store developers should consider properties of the store and globalization issues when developing JSP templates. For more information on globalization, refer to the *WebSphere Commerce Store Development Guide*.

Data beans security consideration

A particular coding practice for the use of data beans minimizes the chance for malicious users to access your database in an unauthorized manner. Insert, select, update and delete parts of SQL statements should be created at development time. Use parameter inserts to gather run-time input information.

An example of using a parameter insert to collect run-time input information follows:

```
select * from Order where owner =?
```

In contrast, you should avoid using input strings as a way to compose the SQL statement. An example of using an input string follows:

```
select * from Order where owner = "input_string"
```

Types of data beans

A data bean is a Java bean that is mainly used to provide dynamic data in JSP templates. There are two types of data beans: smart data beans and command data beans.

A smart data bean uses a *lazy fetch* method to retrieve its own data. This type of data bean can provide better performance in situations where not all data from the access bean is required, since it retrieves data only as required. Smart data beans that require access to the database should extend from the access

bean for the corresponding entity bean and implement the com.ibm.commerce.SmartDataBean interface. For example, the ProductData data bean extends the ProductAccessBean access bean, which corresponds to the Product entity bean.

Some smart data beans do not require database access. For example, the PropertyResource smart data bean retrieves data from a resource bundle, rather than the database. When database access is not required, the smart data bean should extend the SmartDataBeanImpl class.

A command data bean relies on a command to retrieve its data and is a more lightweight data bean. The command retrieves all attributes for the data bean at once, regardless of whether the JSP template requires them. As a result, for JSP templates that use only a selection of attributes from the data bean, a command data bean may be costly in terms of performance time. For JSP templates that require most or all attributes, the command data bean is very convenient.

Command data beans can also extend from their corresponding access beans and implement the com.ibm.commerce.CommandDataBean interface.

Data bean interfaces

Data beans implement one or all of the following Java interfaces:

- com.ibm.commerce.SmartDataBean.
- com.ibm.commerce.CommandDataBean
- com.ibm.commerce.InputDataBean (optional)

Each Java interface describes the source of data from which a data bean is populated. By implementing multiple interfaces, the data bean can access data from a variety of sources. More information about each of the interfaces is provided below.

SmartDataBean interface: A data bean implementing the SmartDataBean interface can retrieve its own data, without an associated data bean command. A smart data bean usually extends from the access bean of a corresponding entity bean. When a smart data bean is activated, the data bean manager invokes the data bean's populate method. Using the populate method, the data bean can retrieve all attributes, except attributes from associated objects. For example, if the data bean extends from an access bean class for an entity bean, the data bean invokes the refreshCopyHelper method. All the attributes from the corresponding entity bean are populated into the smart data bean automatically. However, if the entity bean has associated objects, the attributes from those objects are not retrieved. The main advantages of using smart data beans are:

- Implementation is simple and there is no need to write a data bean command.
- When new fields are added to the entity bean, changes in the data bean are not required. After the entity bean has been modified, the access bean must be regenerated (using the tools in WebSphere Studio Application Developer). As soon as the access bean has been regenerated, all the new attributes are automatically available to the smart data bean.
- Entity beans often contain attributes representing associated objects. For performance reasons, the smart data bean does not automatically retrieve these attributes. Instead, it is preferable to delay retrieval of these attributes until they are required, as shown in the following diagram:

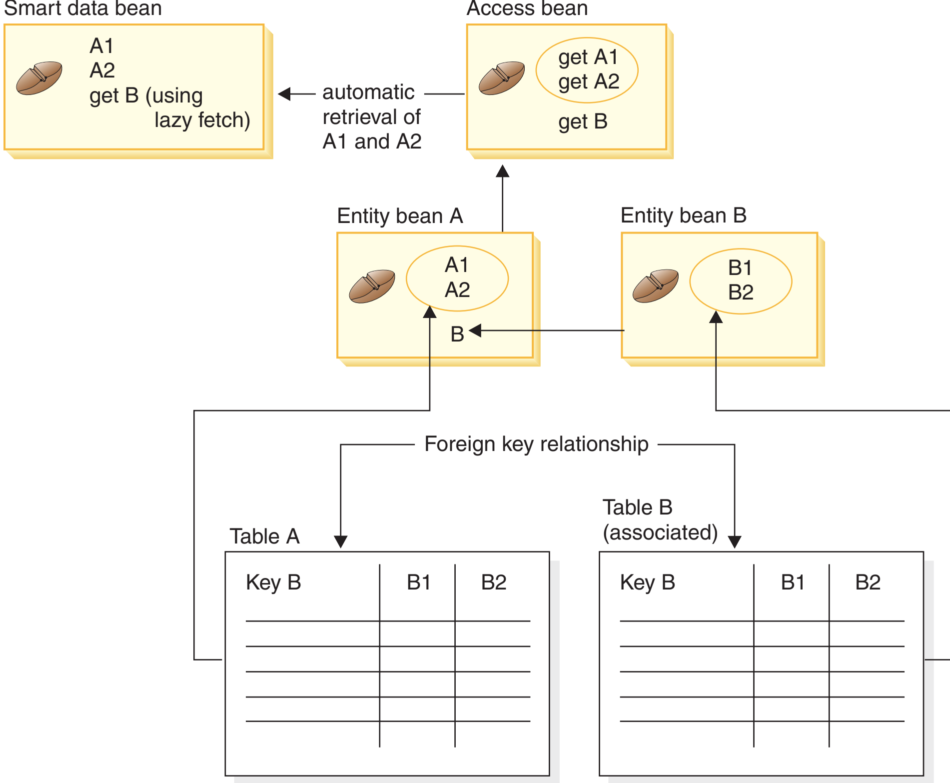


Figure 11.

For more information about implementing a lazy fetch retrieval, refer to "Lazy fetch data retrieval" on page 42.

CommandDataBean interface: A data bean implementing the CommandDataBean interface retrieves data from a data bean command. A data bean of this type is a lightweight object; it relies on a data bean

command to populate its data. The data bean must implement the getCommandInterfaceName() method (as defined by the com.ibm.commerce.CommandDataBean interface) which returns the interface name of the data bean command.

InputDataBean interface: A data bean implementing the InputDataBean interface retrieves data from the URL parameters or attributes set by the view command.

Attributes defined in this interface can be used as primary key fields to fetch additional data. When a JSP template is invoked, the generated JSP servlet code populates all the attributes that match the URL parameters, and then activates the data bean by passing the data bean to the data bean manager. The data bean manager then invokes the data bean's setRequestProperties() method (as defined by the com.ibm.commerce.InputDataBean interface) to pass all the attributes set by the view command. It should be noted that the following code is required in order for the data bean to be activated:

```
com.ibm.commerce.beans.DataBeanManager.activate(data_bean, request)
```

where *data_bean* is the data bean to be activated and *request* is an HTTPServletRequest object.

BeanInfo class

A data bean is not complete without a BeanInfo class that implements the java.lang.Object.BeanInfo interface. The BeanInfo class is used to provide explicit information about the methods and properties of the data bean. It can be used to hide public run-time methods in the data bean implementation class from the Web designer, or to set the appropriate display string for each of the data bean's attributes.

For more information on implementing a BeanInfo class, refer to the JavaBeans specification from Sun Microsystems.

Data bean activation

Data beans can be activated using either the activate or silentActivate methods that are found in the com.ibm.commerce.beans.DataBeanManager class. The activate method is a full activation method in which the activation event is only successful if all attributes are available. If even one attribute is unavailable, an exception is thrown for the whole activation process.

The silentActivate method does not throw exceptions when individual attributes are unavailable.

Invoking controller commands from within a JSP template

Although invoking controller commands from within a JSP template is not consistent with separating logic from display, you may encounter a situation in which this is required. If so, the ControllerCommandInvokerDataBean can be used for this purpose.

Using this data bean, you can specify the interface name of the command to be invoked, or you can directly set the command name to be invoked. You can also set the request properties for the command.

When this data bean is activated by the data bean manager, the controller command is executed and the response properties are available to the JSP template.

If you do not use the setRequestProperties method before activating this data bean, the parameters from the request object will be passed to the bean and, hence, also passed to the controller command. If you do, however, call the setRequestProperties method before activating this data bean, only the specified properties (those passed into the setRequestProperties method) and any default properties specified in the PROPERTIES column of the CMDREG table will be made available to the command.

Once the controller command has executed, you can execute the view.

You should not reuse the same instance of the data bean to invoke other controller commands, as it will contain data and state information from its original usage.

Lazy fetch data retrieval

When a data bean is activated, it can be populated by a data bean command or by the data bean's populate() method. The attributes that are retrieved come from the data bean's corresponding entity bean. An entity bean may also have associated objects, which themselves, have a number of attributes.

If, upon activation, the attributes from all the associated objects were automatically retrieved, a performance problem may be encountered. Performance may degrade as the number of associated objects increase.

Consider a product data bean that contains a large number of cross-sell, up-sell or accessory products (associated objects). It is possible to populate all associated objects as soon as the product data bean is activated. However, populating in this manner may require multiple database queries. If not all attributes are required by the page, multiple database queries may be inefficient.

In general, not all attributes are required for a page, therefore, a better design pattern is to perform a lazy fetch as illustrated below:

```
getCrossSellProducts () {
   if (crossSellDataBeans == null)
     crossSellDataBeans= getCrossSellDataBeans();
   return crossSellDataBean;
    }
```

Setting JSP attributes - overview

The WebSphere Commerce programming model promotes the MVC design pattern. As such, the presentation for the result of a URL request is separated from controller and task commands. These commands are device independent. They implement business logic and produce data to be returned to the client, without having information about the client. Conversely, a view command is device specific.

While the controller and task commands do not directly compose the view, they do pass information to the view. It is important to understand how information is passed to the view. The following diagram demonstrates how properties are passed between the Web controller, command registry, controller command, and view command:

CMREG

INTERFACENAME		PROPERTIES
com.ibm.xxx. NewCommand		parm1=1&parm2=2

CCPd: parm1=1&parm2=2

VIEWREG

INTERFACENAME		PROPERTIES
com.ibm.command.ForwardViewCommand		docname=NewView.jsp

VPd: docName=NewView.jsp

URL: http://*hostname*/webapp/wcs/stores/servlet/NewCommand?storeId=1&...

CCPu: storeID=1&...

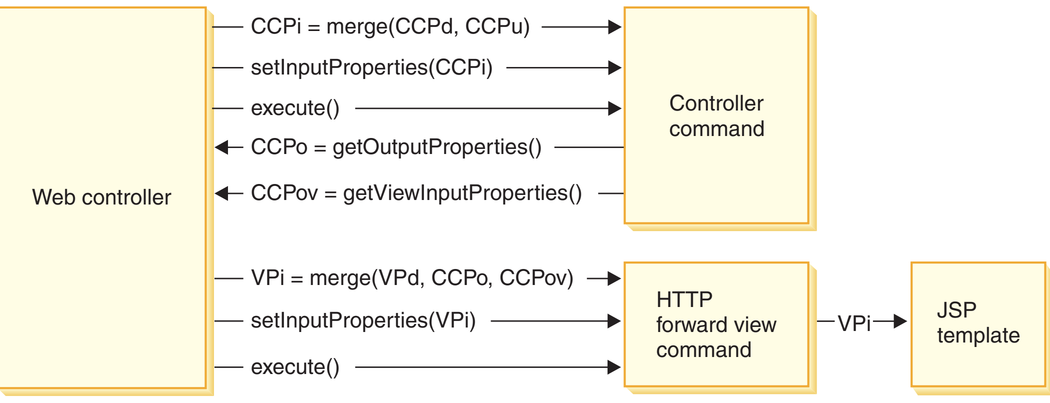


Figure 12.

The preceding diagram shows the following interactions:

- The Web controller merges the input properties from the URL parameters (CCPu) and the entry in the CMDREG table for the controller command (CCPd). This creates CCPi.
- The Web controller passes the merged properties (CCPi) to the controller command and executes the controller command.

- The controller command sets output properties, as CCPo. These are the output properties produced by the command itself. One of the output properties, viewCommandName, is set to the desired view command name. These properties are retrieved by the Web controller using a get method.
- The controller command sets another set of output properties, as CCPov. By default, these are set to the original merged input properties (CCPi). It is possible to customize these properties. For example, it may not be necessary to pass all input parameters to the view command.
- The Web controller merges the three sets of properties, CCPo, CCPov, and VPd (the properties that are registered in the VIEWREG table) into the input properties for the view command (VPi).
- The Web controller sets the merged properties, VPi, and executes the view command.
- The view command sets the attributes to the JSP template from the input properties.

When writing new commands, you do not have explicitly perform the merge of properties. The abstract command classes include a mergeProperties method. For more information about this method, refer to the "Reference" topic in the WebSphere Commerce Production and Development online help.

Required property settings

A controller command must set the following properties for each type of view command. If the properties are not set by the command, they must be defined in the VIEWREG table.

- If using the ForwardView command, set `docname = view_file_name` where *view_file_name* is the name of the display template. For example, `docname=SearchResult.jsp`.
- If using the DirectView command, do one of the following:
  - Set `textDocument = xxx` where *xxx* is a java.io.InputStream object that contains the document in text form
  - Set `rawDocument = yyy` where *yyy* is a java.io.InputStream object that contains the document in binary form

  When using the DirectView command, it is optional to set `contentType = ttt` where *ttt* is the document content type
- If using the RedirectView command, set `url = uuu` where *uuu* is the redirect URL.

Chapter 3. Persistent object model

WebSphere Commerce deals with a large amount of persistent data. There are numerous tables defined in the current database schema. Even with this extensive schema, you may need to extend or customize the database schema for your particular business needs.

WebSphere Commerce uses entity beans that are based on the Enterprise JavaBeans (EJB) Version 1.1 component architecture as the persistent object layer. These entity beans represent WebSphere Commerce data in a manner that models concepts and objects in the commerce domain. This persistence layer provides an extensible framework.

WebSphere Studio Application Developer provides sophisticated EJB tools and a unit test environment that supports development for this framework.

The following sections are within the context of the implementation of the WebSphere Commerce persistent object model implementation, which is at the EJB 1.1 specification.

Implementation of WebSphere Commerce entity beans

WebSphere Commerce entity beans - overview

As mentioned previously, the persistence layer within the WebSphere Commerce architecture is implemented according to the EJB component architecture. The EJB architecture defines two types of enterprise beans: entity beans and session beans. Entity beans are further divided into container-managed persistence (CMP) beans and bean-managed persistence (BMP) beans.

Most of the WebSphere Commerce entity beans are CMP entity beans. A small number of stateless session beans are used to handle intensive database operations, such as performing a sum of all the rows in a particular column. One advantage of using CMP entity beans is that developers can utilize the EJB tools provided in WebSphere Studio Application Developer. These tools allow developers to define Java objects and their database table mappings. The tools automatically generate the required persisters for the entity beans. Persisters are Java objects that persist Java fields to the database and populate Java fields with data from the database.

WebSphere Studio Application Developer provides two extensions to the EJB 1.1 specification: EJB inheritance and association. EJB inheritance allows an

enterprise bean to inherit properties, methods, and method-level control descriptor attributes from another enterprise bean that resides in the same group. An association is a relationship that exists between two CMP entity beans.

Some of the WebSphere Commerce entity beans exploit the EJB inheritance feature. The WebSphere Commerce entity beans do not use the associations feature provided by WebSphere Studio Application Developer. When developing your own entity beans, it is recommended that you do not use WebSphere Studio Application Developer's association feature. This recommendation is made in order to minimize complexity in the object model. Rather than using the association feature provided by WebSphere Studio Application Developer, an object relationship between enterprise beans can be established by adding explicit getter methods in the enterprise beans.

WebSphere Commerce provides two sets of enterprise beans: private and public. Private enterprise beans are used by the WebSphere Commerce run-time environment and tools. You *must not* use or modify these beans.

Public enterprise beans, on the other hand, are used by commerce applications, and can be both used and extended. These public enterprise beans are organized into the following EJB modules:
- Catalog-ProductManagementData
- Enablement-RelationshipManagementData
- Marketing-CampaignsAndScenarioMarketingData
- Marketing-CustomerProfilingAndSegmentationData
- Member-MemberManagementData
- Merchandising-PromotionsAndDiscountsData
- Order-OrderCaptureData
- Order-OrderManagementData
- Trading-AuctionsAndRFQsData

Some of the EJB modules in the preceding list contain session beans. In order to simplify migration in the future, you should not modify a session bean class. If required, you can create a new session bean in the WebSphereCommerceServerExtensionsData EJB module. For more information on creating new session beans, refer to "Writing new session beans" on page 75.

Deployment descriptors for WebSphere Commerce enterprise beans

An EJB deployment descriptor contains deployment settings for enterprise beans. WebSphere Studio Application Developer provides an EJB deployment descriptor editor that can be used to modify this deployment information.

When creating new enterprise beans (entity or session beans), the deployment descriptor information is set within the J2EE Hierarchy view of the J2EE perspective in WebSphere Studio Application Developer. You can view an EJB deployment descriptor for WebSphere Commerce beans by doing the following:

1. Open WebSphere Studio Application Developer and switch to the J2EE perspective.
2. Using the J2EE Hierarchy view, locate the EJB module for which you want to view the deployment descriptor information.
3. Right-click the *EJB_moduleName* EJB module and select **Open with > Deployment Descriptor Editor**.
   The Deployment Descriptor Editor opens.
4. Select the Beans tab and notice the following:
   a. From the list of beans, select a bean. Information for that bean is populated into the other fields.
   b. The bean should be displayed as a Container Managed Entity 1.x bean.
   c. The Reentrant check box should not be selected.
   d. In the WebSphere Bindings section, the default value for the JNDI name is used.
   e. In the WebSphere Extensions section, Enable optimistic locking is not enabled for concurrency control.
   f. Note also that you can view the finders for the bean in the Finders section.
5. Select the Assembly Descriptor tab and notice the following:
   a. In the Method Permissions section, the WCSecurityRole is assigned to all methods in the enterprise bean.
   b. In the Container Transactions section "Required" is specified for all methods in the enterprise bean.
6. Select the Access tab and notice the following:
   a. In the Access Intent for Entities 1.x section, all read-only methods are defined. For example, _copyFromEJB() and handcoded getter methods are assigned to be read-only methods. When you create your own entity beans, ensure that you mark the appropriate methods to be read-only. If read-only methods are not marked in this manner, the EJB container unnecessarily attempts to update the database at the end of a transaction and causes a transaction rollback error in the read-only transaction. This causes performance problems.
   b. The Isolation Level is set as follows:
      - DB2 Repeatable read
      - * DB2 400 Read committed

- Oracle Read committed

Extending the WebSphere Commerce object model

The WebSphere Commerce object model can be extended in the following ways:

- Extend the WebSphere Commerce's public enterprise beans
- Write a new entity bean
- Write a new stateless session bean

Details about how to perform these extensions are contained in the following sections.

Object model extension methodologies

Application requirements may lead to you extend the existing WebSphere Commerce object model. One example of such a requirement is adding additional attributes to your application. This can be accomplished by one of the following ways:

Without modifying an existing WebSphere Commerce public entity bean
: Create a new database table, then create a new entity bean for that table. Add fields and methods to the entity bean to manipulate the new attribute, as required. Generate deployed code and an access bean for the new entity bean. When the application requires the new attribute, it instantiates an access bean object and uses its methods to retrieve, set or manipulate the attribute.

By modifying an existing WebSphere Commerce public entity bean
: Create a new database table, and create a table join between the new table and the existing table that corresponds to the existing enterprise bean that you are modifying. Create new fields in the existing WebSphere Commerce public entity bean and map the fields to their corresponding columns in the new table, using a secondary table map. Add any methods required. Regenerate the deployed code and access bean for the existing entity bean. The new attributes are available when the application instantiates the access bean object.

There are trade-offs between these two approaches. In general, the trade-offs relate to performance and effort for code maintenance.

Extension example: Consider an example in which your application requires you to capture the type of home that a customer has. You create a table called USERRES that contains the customer's ID and type of residence, where residence type (resType) may be a freehold home, a condominium or an apartment. This type of information is demographic information, and as such, is related to the existing Commerce Suite USERDEMO table. In examining the WebSphere Commerce code repository, you find that the Member-

MemberManagementData EJB module contains a "Demographics" enterprise bean. This bean has the getters and setters for demographic information stored in the USERDEMO table.

To perform your customization, there are two options. You can either create a new entity bean that interacts with the USERRES table, or you can add a new field (plus appropriate getter and setter methods) to the Demographics bean.

Using the first approach (creating entirely new code), you create a new Userres entity bean and map its fields to the columns of the USERRES table. When the application requires the customer's residence type, it must instantiate a Userres access bean object and retrieve the data. If the application requires other demographic information at the same time, it must also instantiate a Demographics access bean object and retrieve any other required attributes. Any parts of application logic that attempt to retrieve a complete set of demographic information for a customer must be modified to instantiate the new access bean as well as the original one. The following diagram displays this approach to extending the object model:

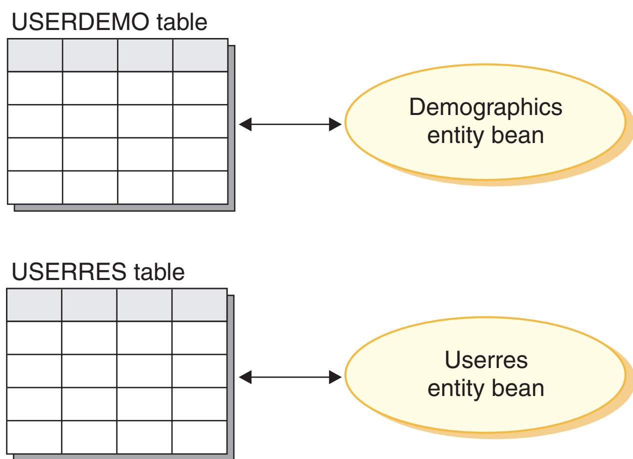


Figure 13.

From a display template perspective, a data bean must be able to access the new attribute, so that the information is available to JSP templates. In order to present a unified view to the Web developer creating the JSP templates, you should create a new data bean that extends the access bean for the original, existing entity bean. The data bean should also use delegation to populate the attributes from the new access bean. The following diagram displays this data bean implementation scenario:

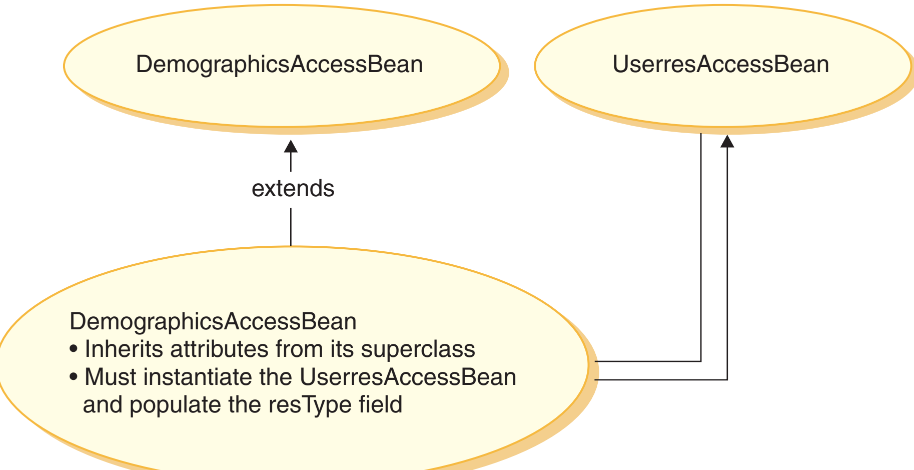


Figure 14.

Using the second approach (modifying existing code), you add a new field to the Demographics entity bean and create a secondary table map between the new field and the appropriate column in the USERRES table. When the application requires the customer's residence type, it instantiates a Demographics access bean object and retrieves the residence type. If the application requires any other demographic information about the customer, it is available in the same call to the bean. The following diagram displays this approach to for enterprise bean modification:

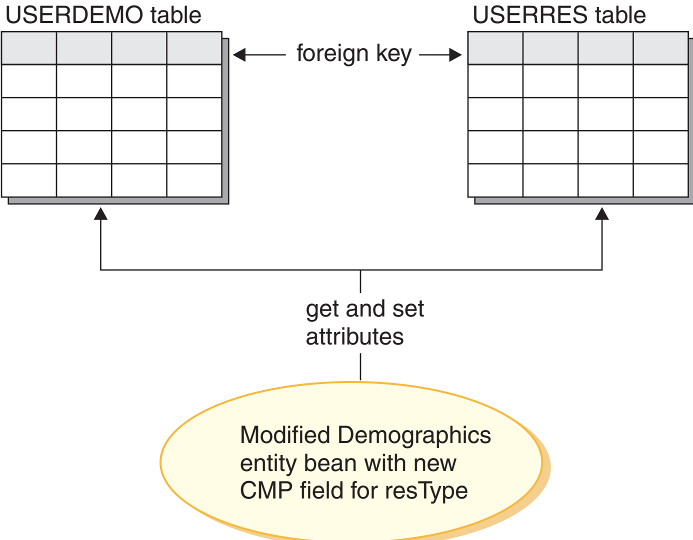


Figure 15.

From a display template perspective, the new attribute (resType) is automatically available in the data bean, as soon as the DemographicsAccessBean is regenerated.

Note that when you are extending the object model, you must *not* add new columns to existing WebSphere Commerce database tables. You must create a new table for the new attribute. If you do attempt to add new columns to existing tables, the new attribute will be lost when you migrate to future releases of WebSphere Commerce.

Performance and code maintenance implications: The second approach has better run-time performance. This is a result of the fact that getting or setting the new attribute only requires the instantiation of a single entity bean and a single fetch is used to retrieve all required attributes.

Due to the fact that the second approach modifies existing WebSphere Commerce code, a migration issue arises when a new version WebSphere Commerce of is released. You must merge your customized code with the new code, but when importing the new WebSphere Commerce workspace, the mapping information between the fields you added to the enterprise bean and the new table is not preserved. As a result, when migrating to a new release of WebSphere Commerce code, the following steps must be performed:

1. Version your customized EJB code.
2. Import the new version of WebSphere Commerce code.

3. Using the tools in WebSphere Studio Application Developer, compare the customized version of code to the new release of WebSphere Commerce code. Merge your customized code back into your workspace.
4. Manually remap any attributes you added to WebSphere Commerce public enterprise beans to the appropriate columns in your database.
5. Regenerate deployed code and access beans for the enterprise beans you modified in step 4.

In order to make this migration simpler, it is important to fully document your object model extensions at development time.

You may select to use a mix of the two approaches when making many extensions to the object model. You can use the first approach for areas of the system that are less susceptible to a degradation in performance and use the second approach were performance is an issue. In this manner, you can minimize effort for future migration, while still maintaining good system performance levels.

Recommended use of session beans

One of the strengths of WebSphere Commerce stems from its ability to take advantage of container-managed persistence (CMP) entity beans. CMP entity beans are distributed, persistent, transactional, server-side Java components that can be generated by the tools provided by WebSphere Studio Application Developer. In many cases, CMP entity beans are an extremely good choice for object persistence and they can be made to work at least as efficiently or even more efficiently than other object-to-relational mapping options. For these reasons, WebSphere Commerce has implemented core commerce objects using CMP entity beans.

There are, however, some situations in which it is recommended to use a session bean JDBC helper. These situations include the following:

- A case where a query returns a large result set. This is referred to as the *large result set* case.
- A case where a query retrieves data from several tables. This is referred to as the *aggregate entity case*.
- A case where an SQL statement performs a database intensive operation. This is referred to as the *arbitrary SQL* case.

More details are provided in the following sections.

Note that if the session bean is being used as a JDBC wrapper to retrieve information from the database, it becomes more difficult to implement resource-level access control. When a session bean is used in this manner, the

developer of the session bean must add the appropriate "where" clauses to the "select" statement in order to prevent unauthorized users from accessing resources.

Large result set case: There are cases where a query returns a large result set and the data retrieved are mainly for read or display purpose. In this case, it is better use a stateless session bean and within that session bean, create a finder method that performs the same functions as a finder method in an entity bean. That is, the finder method in the stateless session bean should do the following:

- Perform an SQL select statement
- For each row that is fetched, instantiate an access bean
- For each column retrieved, set the corresponding attributes in the access bean

When the access bean is returned, the command is unaware of whether the access bean was returned by a finder method in a session bean or from a finder method in an entity bean. As a result, using a finder method in a session bean does not cause any change to the programming model. Only the calling command is aware of whether it is invoking a finder method in a session bean or in an entity bean. It is transparent to all other parts of the programming model.

Aggregate entity case: In this case, one view combines parts of several objects and a single display page is populated with pieces of information that come from several database tables. For example, consider the concept of "My Account". This may consist of information from table of customer information (for example, the customer name, age and customer ID) and information from an address table (for example, an address made up of a street and city).

It is possible to construct a simple SQL statement to retrieve all of the information from the various tables by performing an SQL join. This can be referred to as performing a "deep fetch". The following is an example of an SQL select statement for the "My Account" example, where the CUSTOMER table is T1 and the ADDRESS table is T2:

```
select T1.NAME, T1.AGE, T2.STREET, T2.CITY
  from CUSTOMER T1, ADDRESS T2
  where (T1.ID=? and T1.ID=T2.ID)
```

The tools in WebSphere Studio Application Developer for enterprise beans at the EJB 1.1 specification do not support this notion of a deep fetch. Instead, it does a lazy fetch that results in an SQL select for each associated object. This is not the preferred method for retrieving this type of information.

In order to perform a deep fetch, it is recommended that you use a session bean. In that session bean, create a finder method to retrieve the required information. The finder method should do the following:

- Perform an SQL select statement for the deep fetch
- Instantiate an access bean for each row in the main table as well as for each associated object.
- For each column fetched and for each associated object fetched, set the corresponding attribute in the access bean.

Note that an access bean does not cache a getter method that throws an exception. In this case, you should create a simple wrapper class for the access bean using the following pattern:

```
public class CustomerAccessBeanCopy extends CustomerAccessBean {
   private AddressAccessBean address=null;

   /* The following method overrides the getAddress method in
      the CustomerAccessBean.
   */
   public AddressAccessBean getAddress() {
      if (address == null)
         address = super.getAddress();
      return address;
      }

    /* The following method sets the address to the copy. */

    public void _setAddress(AddressAccessBean aBean) {
       address = aBean;
      }
}
```

Continuing the CUSTOMER and ADDRESS example, the session bean finder method would instantiate a CustomerAccessBean for each row in the CUSTOMER table and an AddressAccessBean for each corresponding row in the ADDRESS table. Then, for each column in the ADDRESS table, it sets the attributes in the AddressAccessBean (street and city). For each column in the ADDRESS table, it sets the attributes in the CustomerAccessBean (name, age and address). This is shown in the following diagram.

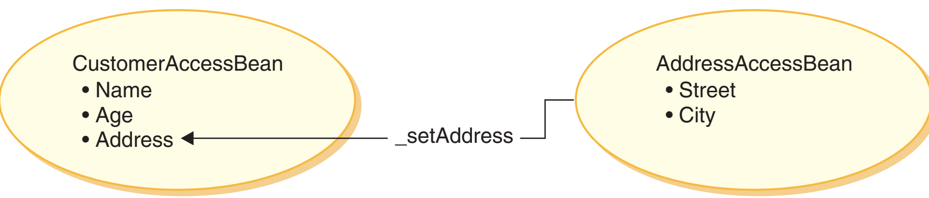


Figure 16.

Arbitrary SQL case: In this case, there is a set of arbitrary SQL statements that perform database intensive operations. For example, the operation to sum all the rows in a table would be considered a database intensive operation. It is possible that not all of the selected rows correspond to an entity bean in the persistent model.

An example that could result in the creation of an arbitrary SQL statement is a when a customer tries to browse through a very large set of data. For example, if the customer wanted to examine all of the fasteners in an online hardware store, or all of the dresses in an online clothing store. This creates a very large result set, but out of this result set, it is most likely that only a few fields from each row are required. That is, the customer may only initially be presented with a summary showing the item name, picture and price.

In this case, create a session bean helper method. This session bean helper method either performs a read or a write operation. When performing a read operation, it returns a read-only value object that is used for display purposes.

With proper data modelling, the number of cases of arbitrary SQL statements can usually be minimized.

Extending public entity beans

This section describes the design pattern of the WebSphere Commerce public entity beans. This design pattern enables you to make extensions such as adding new persistent fields, new business methods, or new finder methods.

The following diagram shows the implementation classes of the Catalog entity bean.

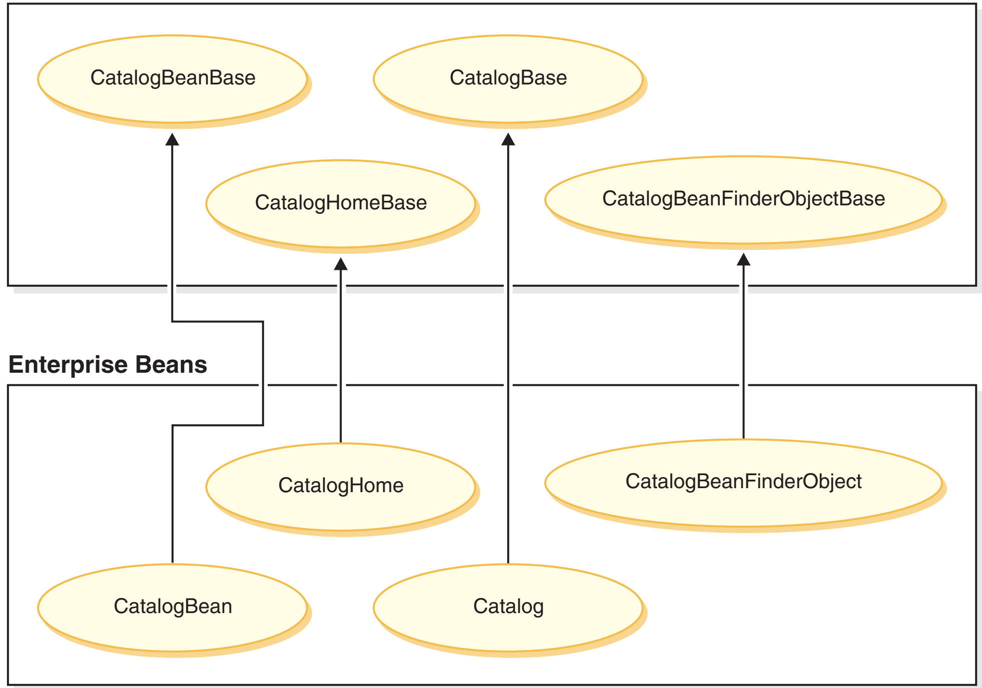


Figure 17.

The preceding diagram also applies to other entity beans because they are structured in a similar fashion and follow the same naming convention. To apply the diagram to another entity bean, substitute the entity bean's name for "Catalog". For example, the InterestItemBean class extends the InterestItemBeanBase class and the InterestItem interface extends the InterestItemBase interface.

The diagram shows that the implementation class or interface for the public enterprise beans has been separated into two parts, using Java inheritance. The superclass or interface contains the WebSphere Commerce implementation code. All of these superclasses and interfaces are defined in separate Java packages from the child classes and interfaces.

The WebSphere Commerce workspace contains binary code for all of these superclasses and interfaces. Modifications can be made to the child classes and interfaces. In general, modifications can be made in the com.ibm.commerce.*xxx*.objects and com.ibm.commerce.*xxx*.objsrc packages (where *xxx* is a component name).

If you add new finder methods to the public enterprise beans, you must follow a particular naming convention for the methods. Name the new methods `findX`*`a_description`* where *`a_description`* is a description of your choice. Some examples of names are `findXByOwnerId` and `findXByOrderStatus`. Using this naming convention avoids the risk of name collision (duplicate names) with WebSphere Commerce finder methods. The deployment descriptor editor is used when adding new finders.

One way to modify an existing WebSphere Commerce public entity bean is to add additional fields. In this case, after adding the new fields, you must examine each finder method in the bean. If the `where` clause portion of the finder methods contain any database aliases (for example, T1. or T2.), the aliases must be removed.

Public entity beans that contain "findForUpdate" types of finders: If a WebSphere Commerce public entity bean contains any "findForUpdate" types of finders, you cannot add new fields to the bean by creating a secondary map to a new table that you have created. This is due to the fact that if you do create a secondary map, the generated SQL statements will be invalid, and the bean will no longer function as desired. If you want to extend the part of the object model represented by such a bean, you must create a new entity bean, and use the original bean and along with the new bean in your customized code.

Creating a new CMP enterprise bean

When you have a new attribute that needs to be added to the WebSphere Commerce object model, you can create a new database table with a column for the required attribute. You must then also include this attribute in an enterprise bean, so that WebSphere Commerce commands can access the information.

One way to integrate the new attribute into the WebSphere Commerce object model is to create a new CMP enterprise bean. In this bean, you create a field that corresponds to the attribute in the new database table.

Due to the fact that the WebSphere Commerce workspace provides a predefined EJB project for your new enterprise beans, you do not need to create an additional EJB project. Your new enterprise beans should be placed into the WebSphereCommerceServerExtensionsData EJB project. Later, when you deploy your customized beans, you create a WebSphereCommerceServerExtensionsData.jar JAR file and replace the existing JAR file in the WebSphere Commerce enterprise application running in WebSphere Application Server. By using this packaging convention, deployment is greatly simplified.

To create a new CMP enterprise bean, you must perform the following steps in WebSphere Studio Application Developer:

1. Create the new CMP enterprise bean, using the Enterprise Bean Creation wizard. For each column in the corresponding database table, add a new CMP field to the bean.
2. Set the transaction isolation level for the new bean.
3. Set the security identity of the new bean.
4. Modify the entity context methods.
5. If required, define new finders using the EJB deployment descriptor editor.
6. Create a new ejbCreate method, if required, and promote the ejbCreate method to the home interface of the enterprise bean. This step is required if the new enterprise bean must create new entries in its corresponding database table.
7. If the bean is protected by the WebSphere Commerce access control system, implement the required access control methods in the bean. Refer to Chapter 4, "Access control," on page 89 for more details about implementing access control in enterprise beans. Optionally, you can implement access control after you have created your access bean.
8. Map the fields in the enterprise bean to the columns in the database table.
9. Generate the corresponding access bean for the enterprise bean.
10. Generate the deployed code for the enterprise bean.
11. Test the bean using the Universal Test Client in WebSphere Studio Application Developer.

More detail on each of these steps is contained in the following sections. When reading through these sections, suppose that you have a new table called XUSERRES that specifies some information about the type of residence a user has. This table contains three columns: a USERID column, a HOME column that specifies the type of home and a ROOMS column that specifies the number of bedrooms in the residence.

Creating the new CMP enterprise bean: To create your new CMP enterprise bean, you can use the Enterprise Bean Creation wizard, as follows:

1. Within the J2EE Hierarchy view, expand **EJB Modules**.
2. Right-click the **WebSphereCommerceServerExtensionsData** module and select **New > Enterprise Bean**.
   The Enterprise Bean Creation wizard opens.
3. From the **EJB Project** drop-down list, select **WebSphereCommerceServerExtensionsData** and click **Next**.
4. In the Create an Enterprise Bean window, do the following:
   a. Select **Entity bean with container-managed persistence (CMP) fields**

b. In the **Bean name** field, enter an appropriate name for your bean. Typically, the bean name matches the name of the corresponding database table. For example, you would name the bean XUserRes to correspond to the XUSERRES table.
c. In the **Source folder** field, leave the default value that is specified (ejbModule).
d. In the **Default package** field, enter `com.mycompany.mycomponent.objects`.
e. Click **Next**.

5. In the Enterprise Bean Details window, do the following:
   a. Click **Add** to add a new CMP attributes for columns in your database table.
   The Create CMP Attribute window opens. In this window, do the following:
      1) In the **Name** field, enter an appropriate name for the new CMP field. Note that if you want to use the Match by name function later when mapping this field to its corresponding column in the database table, name your field exactly (case insensitive) to the name of the column.
      2) In the **Type** field, enter the appropriate data type for the field. Note that you should use the wrapper classes for primitive data types (for example, use the *java.lang.Long* data type, not the *long* data type
      3) If the field is the primary key, select the **Key Field** check box and click **Apply**.
      4) If the field is not the primary key, select the **Access with getter and setter methods** check box.
      5) If the field is not the primary key, clear the **Promote getter and setter methods to remote interface** check box. The **Make getter read-only** check box will be made unavailable and then click **Apply**.
      6) Click **Add** again and repeat steps to add new fields for each column in the database table that requires a CMP field.
      7) Click **Close** to close this window.
   b. Clear the **Use the single key attribute type for the key class** check box, then click **Next**.
6. In the EJB Java Class Details window, do the following:
   a. To select the bean's superclass, click **Browse**.
   The Type Selection window opens.
   b. In the **Select a class using: (any)** field, enter `ECEntityBean` and click **OK**. This selects the com.ibm.commerce.base.objects.ECEntityBean as the superclass.

c. If the new enterprise bean is to be protected under the WebSphere Commerce access control framework, specify the interfaces that the remote interface should extend by clicking **Add**. The Type Selection window opens.
d. In the **Select a class using: (any)** field, enter `Protectable` and click **OK**. This selects the com.ibm.commerce.security.Protectable. This interface is required in order to protect the new resource under access control.
e. Click **Finish**.

In the bean class, WebSphere Studio Application Developer creates the private field called `EntityContext`. WebSphere Commerce provides its own entity context field in the ECEntityBean and your new entity bean should use that field, rather than the generated field. As such, you should remove the generated `EntityContext` from your new entity bean. This is described in a subsequent section.

When you specify `com.ibm.commerce.base.objects.ECEntityBean` as the superclass your bean inherits certain functions. The following example of code demonstrates these functions:

```
public class myEJB extends com.ibm.commerce.base.objects.ECEntityBean {
   public void ejbLoad() {
      super.ejbLoad();--the super method will add EJB trace
       --your logic --
   }
   public void ejbStore() {
      super.ejbStore();--the super method will add EJB trace
      --your logic --
   }
}
```

Setting the transaction isolation level: You must set the transaction isolation level for the bean to the correct value for your development database type. To set the transaction isolation level, do the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**.
2. Double-click the **WebSphereCommerceServerExtensionsData** project to open it with the Deployment Descriptor Editor.
3. Click the **Access** tab.
4. Click **Add** next to the Isolation Level text box.
   The Add Isolation Level window opens.
5. DB2 Select **Repeatable Read**, then click **Next**.
   * DB2 400 Select **Read Committed**, then click **Next**.
   Oracle Select **Read Committed**, then click **Next**.
6. From the **Beans found** list, select the *yourNewBean* bean, then click **Next**.

7. From the **Methods found** list, select *yourNewBean* to select all of its methods, and click **Finish**.
8. Save your work (Ctrl+S), and keep the editor open.

Setting the security identity of the bean: Next set the security identity of the bean, by doing the following:
1. In the Deployment Descriptor editor, ensure that you have the Access tab selected.
2. Click **Add** next to the Security Identity text box.
   The Add Security Identity window opens.
3. Select **Use identity of EJB server**, then click **Next**.
4. From the **Beans found** list, select the *yourNewBean* bean, then click **Next**.
5. From the **Methods found** list, select *yourNewBean* to select all of its methods, and click **Finish**.
6. Save your work (Ctrl+S). Keep the editor open.

Setting the security role of the bean: Next, set the security role for the methods in the bean, by doing the following:
1. In the Deployment Descriptor editor, select the Assembly Descriptor tab.
2. In the Method permissions section, click **Add**.
3. Select **WCSecurityRole** as the security role and click **Next**.
4. From the list of beans found, select *yourNewBean* and click **Next**.
5. In the Method elements page, click **Apply to All**, then click **Finish**.
6. Save your work (Ctrl+S) and then close the Deployment Descriptor editor.

Deleting the entity context fields and methods: The next step is to remove some of the fields and methods related to entity context that WebSphere Studio Application Developer generates. The reason that these fields need to be deleted is that the ECEntityBean base class provides its own implementation of these methods. To delete the generated entity context fields and methods, do the following:
1. In the J2EE Hierarchy view, expand the **WebSphereCommerceServerExtensionsData** project.
2. Expand **Entity Beans**, the *yourNewBean* bean, and then double-click the *yourNewBean***Bean** class.
3. In the Outline view, do the following:

   Note: Express WebSphere Studio Application Developer 5.1 prompts you to delete getEntityContext() and setEntityContext(EntityContext), so you do not have to delete them manually as mentioned below. Ensure that you select you delete these.

   a. Right-click the **myEntityCtx** field and select **Delete**.

b. Right-click the **getEntityContext()** method and select **Delete**.
c. Right-click the **setEntityContext(EntityContext)** method and select **Delete**.
d. Right-click the **unsetEntityContext()** method and select **Delete**.

4. Save your work (Ctrl+S).

Adding new finders:

Introduction to working with finders: The use of finder helper interfaces has been deprecated. For any new development work, it is required that you use the EJB deployment descriptor editor rather than the finder helper interface to define your queries and method declarations.

For detailed information about the use of the EJB query language for creating finders, advantages of this approach over other approaches to finders and more details about related tools, refer to the WebSphere Studio Application Developer online help.

Adding a new finder to your new bean: If you need to add a new finder to your enterprise bean, do the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**.
2. Double-click the **WebSphereCommerceServerExtensionsData** project to open the EJB Deployment Descriptor Editor.
3. Click the **Beans** tab.
4. In the Beans pane, select the *yourNewBean* bean, then in the pane on the right, scroll down and expand **WebSphere Extensions**.
5. Click **Add** next to the **Finders** text box.
   The Add Finder Descriptor window opens.
6. Select **New**, then in the **Name** field, enter `findByX`*`yourArg`* (where *yourArg* is the name of the argument by which you are searching). Use the "findXBy" naming convention for your field name to ensure that your field names are always unique from WebSphere Commerce field names.
7. Click **Add** next to the **Parameters** text box, then do the following
   a. In the **Name** field, enter *`yourArg`*.
   b. In the **Type** field enter the appropriate data type.
   c. Click **OK**.
8. In the **Return Type** field, enter one of the following and click **Next**:
   - If the FinderHelper method uses the primary key to query the database and the method should return a unique record, specify the EJB object as the return type. For example, enter `UserRes`.
   - If the FinderHelper method returns a result set instead of a unique record, specify the return type as `java.util.Enumeration`.

9. From the **Finder type** drop-down list, select **WhereClauseFinderDescriptor**.
10. In the **Finder statement** field, an appropriate finder. For example, enter `T1.MEMBERID = ?`, then click **Finish**.
11. Save your work, then close the EJB Deployment Descriptor editor.

For security reasons, when creating FinderHelper methods for a new entity bean, you should use parameter inserts as shown in the preceding steps. The reason for this recommendation is that it protects the query from being altered by users. An alternative approach would be to use a construct similar to the following:

```
T1.MEMBERID = "input_string ";
```

where *input_string* is a string value passed in from a URL. This is not desirable, since a malicious user could enter a value such as "'123' OR 1=1" which changes the SQL statement. If a user can change the SQL statement, they may be able to make unauthorized access to data. Therefore, the recommended approach is to use parameter inserts.

If you cannot use a parameter insert and therefore, have to use an input string to compose the SQL statement, you must enforce parameter checking on the input string to ensure that the input parameter is not a malicious attempt to access data.

Creating a new ejbCreate method: When the enterprise bean is created, the ejbCreate method is automatically generated. This method is then promoted to the remote interface, so that it is available in the access bean. The default ejbCreate method only contains parameters that are either the primary key, or part of the primary key. This means, only those values get instantiated upon instantiation.

If your enterprise bean contains fields that are not part of the primary key and are non-nullable fields, you must create a new ejbCreate method in which you specifically instantiate those fields. By doing so, each time a new record is created, all non-nullable fields will be populated with the appropriate data.

To create a new ejbCreate method, do the following:

1. In the J2EE Hierarchy view, double-click the ***yourNewBean*Bean** class to open it and view its source code.
2. You must modify the source code so that each non-nullable CMP field is included as an input parameter to the method, and so each CMP field is instantiated with the appropriate value. In the UserRes example where the UserId is the primary key, the source code initially appears as:

```
public void ejbCreate(int argUserId)
   throws javax.ejb.CreateException {
   _initLinks();
   userId = argUserId;
}
```

But, you may want to ensure that both the number of rooms and type of home are initialized. In this case, you would change the code to the following:

```
public void ejbCreate(int argUserId, String argHome, byte Rooms)
   throws javax.ejb.CreateException {
   _initLinks();
   // All CMP fields should be initialized here
   userId = argUserId;
   home = argHome;
   rooms = argRooms;
}
```

Note: If you want to use a system generated primary key, refer to "Primary keys" on page 80 for details.

3. You must add the new ejbCreate method to the home interface. This makes the method available in the generated access bean. To add the method to the home interface, do the following:
   a. Right-click the **ejbCreate(***yourParameters***)** method in the Outline view and select **Enterprise Bean > Promote to Home Interface**.

Creating a new ejbPostCreate method: Next, you must create a new ejbPostCreate method that has the same input parameters as the new ejbCreate method. To create this new method, do the following:

1. Double-click the ***yourNewBean*****Bean** class to open it and view its source code.
2. Create a new ejbPostCreate method with the same input parameters that were used in the new ejbCreate method. To continue the user residence example, you would the following code into the class:

   ```
   public void ejbPostCreate(int argUserId,
      String argHome, byte Rooms)
      {
      }
   ```

   Save the code changes.

Adding access control methods to the bean: If your new bean is to be protected under access control, you must add the getOwner method. Another method that is optional for access control purposes is the fulfills method. For details about required and optional methods, refer to "Implementing access control in enterprise beans" on page 106. To add access control methods to the new bean, do the following:

1. In the J2EE Hierarchy view, double-click the ***yourNewBean*****Bean** class to open it and view its source code.
2. Into the source code, add a getOwner method that includes logic to return the owner of this resource. For example, to return the owner of the UserRes bean, you would return the member ID of the user:
```
public java.lang.Long getOwner()
   throws java.lang.Exception {
      return getMemberId();
   }
```
3. For the purpose of this example, you would add a fulfills method that specifies what relationship the user must satisfy before they are allowed to act upon this resource. In this case, you would specify that only the creator of this UserRes object is allowed to take action. In other words, each user is only allowed to take action upon their own UserRes object. This relationship requirement is shown in the following code snip:
```
public boolean fulfills(Long member, String relationship)
   throws java.lang.Exception {
      if (relationship.equalsIgnoreCase("creator"))
      {
         return member.equals(getMemberId());
      }
      return false;
   }
```
4. Save your work.

Mapping the database table to the new enterprise bean: Once you have created the new enterprise bean, you must create a mapping between the CMP fields in the bean and the columns in the database table. When both the enterprise bean and its corresponding database table exist, a "Meet-in-the-middle" type of mapping is used. WebSphere Studio Application Developer provides tools to simplify this task.

To create the mapping, do the following:
1. In the J2EE Hierarchy view, right-click **WebSphereCommerceServerExtensionsData** and select **Generate > EJB to RDB Mapping**.
   The EJB to RDB Mapping Window opens.
2. Select **Meet In The Middle** and click **Next**.
3. In the Database Connection window, do the following:
   a. In the **Connection name** field, enter `WebSphereCommerceServerExtensionsData`
   b. In the **Database field**, enter *developmentDB*
   c. In the **User ID** field, enter *dbuser*
   d. In the **Password** field, enter *dbpassword*

e. From the database drop-down list, select the database vendor type for your development database.
   - DB2 DB2 Universal Database 8.1
   - Oracle Oracle 9i
f. Oracle In the **Host** field, enter the fully-qualified host name of your database server. For example, enter `dbserver.yourcompany.com`
g. Oracle In the Class Location field, enter the location of the classes12.zip file. For example, enter `D:\oracle\ora92\jdbc\lib\classes12.zip`
h. Click **Next**. Once the connection is established, the list of tables in the database is displayed. You can also view the Connection Document later by looking at the Database Servers view in the Data perspective.

4. Select the ***yourNewTable*** table and click **Next**.
5. Select **Match By Name and Type** and then click **Finish**. The Mapping Editor now opens.
6. Oracle If any columns have a data type of "NUMBER", you must modify the data type. Right-click the ***yourNewTable*** table and select **Open Table Editor**. In the table editor, do the following:
   a. Select the **Column** tab.
   b. Select the column whose data type requires changing and change the column type from NUMBER to a more specific type. For example, change it to INTEGER.
   c. Save your changes.
7. In the Enterprise Beans pane, expand the ***yourNewBean***bean. In the Tables pane, expand the ***yourNewTable*** table.
8. Map the fields in the *yourNewBean* bean to the columns in the *yourNewTable* table, by doing the following:
   a. Right-click the ***yourNewBean*** bean and select **Match By Name**.
9. Save the changes made to the Map.mapxmi file and then close the file.
10. Oracle You must edit the table definition using a text editor, as follows:
    a. Open the *yourNewBean*.xmi file with a text editor.
    b. Replace all occurrences of `SQLNumeric6` to `SQLNumeric3`.
    c. Save your changes and then close the file.

Modifying the schema name: The next step is to modify the schema name so that your bean will be portable to other databases. The special value to allow a bean to be portable in this manner is NULLID. To modify the schema name, do the following:

1. In the J2EE Perspective, switch to the J2EE Hierarchy view.

2. Expand **Databases**, then **expand WebSphereCommerceServerExtensionsData**.
3. Right-click on the schema node (for example, db2user) and select **Rename**.
4. Set the value to `NULLID`.

Creating the access bean: An access bean acts as a wrapper for the enterprise bean that simplifies how other components interact with the enterprise bean. You must create an access bean for your new enterprise bean. The tools in WebSphere Studio Application Developer are used to generate this access bean, based upon the entity that you have already created (in particular, only methods that have been promoted to the remote interface will be used by the access bean).

To create the access bean, do the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**, then right-click **WebSphereCommerceServerExtensionsData** and select **New > Access Bean**.
   The Add an Access Bean window opens.
2. Select **Copy Helper** and click **Next**.
3. Select the ***yourNewBean***bean and click **Next**.
4. From the Constructor method drop-down list, select **findByPrimaryKey(***yourPackageName***.** *yourNewBean***Key)** as the constructor method.
5. Select all attributes in the Attribute Helpers section.
6. Click **Finish**.
7. Save your work.

Generating deployed code: The code generation utility analyzes the beans to ensure that Sun Microsystems' EJB specifications are met and it ensures that rules specific to the EJB server are followed. In addition, for each selected enterprise bean, the code-generation tool generates the home and EJBObject (remote) implementations and implementation classes for the home and remote interfaces, as well as the JDBC persister and finder classes for CMP beans. It also generates the Java ORB, stubs, and tie classes required for RMI access over IIOP, as well as stubs for the home and remote interfaces

To generate the deployed code, do the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**, then right-click **WebSphereCommerceServerExtensionsData** and select **Generate > Deploy and RMIC Code**.
   The Deploy and RMIC Code window opens.
2. Select the ***yourNewBean*** bean and click **Finish**.

You can view the newly generated code by switching to the Business Professional J2EE Navigator view Express Project Navigator view. You will find the following:

Table 1.

Type of code	Class name
Container implementation generated code	EJSCMP*yourNewBean*HomeBean.java
	EJSRemoteCMP*yourNewBean*.java
	EJSRemoteCMP*yourNewBean*Home.java
	EJSFinder*yourNewBean*Bean.java
JDBC access code	EJSJDBCPersisterCMP*yourNewBean*Bean.java
RMI tie and stub code	_EJSRemoteCMP*yourNewBean*_Tie.java
	_*yourNewBean*_Stub.java
	_EJSRemoteCMP*yourNewBean*Home_Tie.java
	_*yourNewBean*Home_Stub.java

Using the test client to test the enterprise bean: WebSphere Studio Application Developer provides a test client that can be used to test enterprise beans. To use the test client to test your new bean, do the following:

1. Switch to the Servers perspective.
2. Double-click the **WebSphereCommerceServer** server and click the **Configuration** tab.
3. Select **Enable universal test client**. Save the changes.
4. Right-click the **WebSphereCommerceServer** server and select **Start**.
5. Right-click the **WebSphereCommerceServerExtensionsData** and select **Run on server**.
   The Web browser opens with the universal test client. The starting place for testing the enterprise bean is the JNDI Explorer where you can find the names of the beans that run in the EJB container on this server.
6. Right-click **JNDI Explorer**
7. Next you must navigate to the *yourNewBean*Home interface, by expanding the hierarchy as follows: **cell > nodes > localhost > servers > server1 > ejb >** *yourPackageStructure*.
8. Click the ***yourNewBean*Home** interface. In the References pane, select **EJB References >** ***yourNewBean*** **>** ***yourNewBean*Home**. Click the create method.
9. In the fields in the right pane that correspond to the required input parameters for the create method, enter appropriate values.
10. Click **Invoke** and the result is shown in the bottom pane.

11. Click **Work with Object** to add the remote interface to the Reference pane and you will see the values you entered under Object Reference. A new record has been created in your new table.
12. Test other methods, as appropriate.
13. Close the test client and stop the server.

Coding Practices: The following enterprise bean coding practices should be observed:

- Do not use either the BLOB or CLOB data type.
- Enterprise bean code should not reference anything outside of the enterprise bean modules. For example, you should not reference commands or data beans in the enterprise bean code
- The previous sections describe how to include access control in a new bean when initially creating the bean. It can also be added after you have created your bean by adding the com.ibm.commerce.security.Protectable interface. If required, also add the com.ibm.commerce.security.Groupable interface to the enterprise bean's remote interface. Certain methods must be implemented in the bean as well. After adding these interfaces and adding required methods, regenerate the bean's deployed code and access bean. For more information, refer to "Implementing access control in enterprise beans" on page 106.

Creating a simple data bean

A data bean is a bean that is used in JSP templates to retrieve information from the enterprise bean. A simple data bean extends its corresponding access bean and implements the SmartDataBean interface. Most code for the data bean is automatically generated by WebSphere Studio Application Developer.

New data beans are stored in the WebSphereCommerceServerExtensionsLogic project.

To create a simple data bean, you must perform the following steps:

1. Create a package to store the data bean code.
2. Create a data bean that extends the corresponding access bean and implements the appropriate data bean interface.
3. Create the set methods for the data bean.
4. Create the get methods for the data bean.

Each step is described in more detail in subsequent sections.

Creating the package for data bean code: Creating package creates a place in which your data bean code can be stored.

To create a new package, do the following:

1. Open WebSphere Studio Application Developer and switch to the Java perspective.
2. Right-click the **WebSphereCommerceServerExtensionsLogic** project select **New > Package**. The New Java Package wizard opens.
3. The value for **Source Folder** is prepopulated with `WebSphereCommerceServerExtensionsLogic/src`. Keep this value.
4. In the **Name** field, enter an appropriate name for your new package. For example, enter `com.mycompany.mydatabeans`.
5. Click **Finish**.

Creating a data bean: A data bean is a Java bean that is used within a JSP template to provide dynamic content to the page. It normally provides a simple representation (indirectly) of an entity bean by extending an access bean. The data bean encapsulates properties that can be retrieved from or set within the entity bean.

To create a data bean, do the following:

1. Right-click the package into which you will store the data bean and select **New > Class**.
   The New Java Class wizard opens.
2. The project and package name fields are already populated.
3. In the **Name** field, enter a name for your new data bean. For example, to create a data bean that extends the UserResAccessBean, enter `UserResDataBean`.
4. From the **Modifiers** list, select **public**.
5. To specify the superclass, click **Browse**, then in the pattern field, enter the name of the corresponding access bean. For example, enter `UserResAccessBean` and click **OK**.
6. To specify the interfaces that the data bean should implement, click **Add**. In the Interface window, do the following:
   a. In the **Pattern** field, enter `com.ibm.commerce.beans.SmartDataBean` then click **Add**.
   b. In the **Pattern** field, enter `com.ibm.commerce.beans.InputDataBean` then click **Add**.
   c. Click **OK**.
7. Click **Finish**.

Adding required fields to the data bean: This section describes how to modify the required fields in your new data bean. The two required fields are for the following types of information:

- command context
- request properties

To modify the iCommandContext field, do the following:

1. Double-click the new data bean (for example, UserResDataBean) to view its source code.
2. Locate the getCommandContext method. It initially appears as follows:

   ```
   public CommandContext getCommandContext() {
     return null;
    }
   ```

   Modify the source code so that it appears as follows:

   ```
   private CommandContext iCommandContext = null;

   public com.ibm.commerce.command.CommandContext getCommandContext ()
   {
     return iCommandContext;
   }
   ```
3. Locate the setCommandContext method. It initially appears as follows:

   ```
   public void setCommandContext(CommandContext arg0) {
    }
   ```

   Modify the source code so that it appears as follows:

   ```
   public void setCommandContext(com.ibm.commerce.command.CommandContext
      aCommandContext)
   {
     iCommandContext = aCommandContext;
   }
   ```
4. Save your work.

To modify the iRequestProperties field, do the following:

1. Double-click the new data bean (for example, UserResDataBean) to view its source code.
2. Locate the getRequestProperties method. It initially appears as follows:

   ```
   public TypedProperty getRequestProperties() {
     return null;
    }
   ```

   Modify the source code so that it appears as follows:

   ```
   private com.ibm.commerce.datatype.TypedProperty
      requestProperties;

   public TypedProperty getRequestProperties()
   {
     return requestProperties;
   }
   ```
3. Locate the setRequestProperties method. It initially appears as follows:

   ```
   public void setRequestProperties(TypedProperty arg0) throws Exception {
    }
   ```

Modify the source code so that it appears as follows:

```
public void setRequestProperties(com.ibm.commerce.datatype.TypedProperty
   aParam)
throws Exception
{
  // copy input TypedProperties to local
  requestProperties = aParam;
}
```

4. Save your work.

Populating the primary key of the corresponding access bean: Note that you may want to modify the source code to populate the primary key of the corresponding access bean. The recommended way to do this is to use the data bean manager to indirectly set this value. This indirect method is designed to ensure that a primary key value taken from the URL properties will not override the primary key, if it has previously been set. To have your setRequestProperties method follow this model, code it in a fashion that is similar to the following code snippet. Note that in the following example, the primary key is the user ID. This may be different, depending upon the situation (as such, the following code may not immediately compile in your application).

```
public void setRequestProperties(
   com.ibm.commerce.datatype.TypedProperty arg1)
   throws Exception
   {
      iRequestProperties = arg1;
      try {
          if (// check for nulls
              getDataBeanKeyUserId() == null)
             {
                super.setInitKey_UserId(aUserId);
             }
      } catch (com.ibm.commerce.exception.ParameterNotFoundException e)
           {
           }
   }
```

There are two other ways in which the primary key for the access bean can be set. It can be done externally from the data bean, for example in the JSP template. In this case, before activating the data bean in the JSP template, explicitly call the data bean's set method for the primary key. For example, the JSP could include code similar to the following (where `db` is the data bean object):

```
db.setInitKey_UserId(/*input parameter*/)
db.activate();
```

Alternatively, the primary key can be set in a direct way. That is, the JSP template only contains the db.activate method and then the data bean

manager explicitly sets the primary key in the access bean. For example, the code for the setRequestProperties method of the data bean would appear similar to the following:

```
public void setRequestProperties(
   com.ibm.commerce.datatype.TypedProperty arg1)
   throws Exception
      {
         iRequestProperties = arg1;
         try
             {
                super.setInitKey_UserId(aUserId);
             }
       } catch (com.ibm.commerce.exception.ParameterNotFoundException e)
           {
           }
      }
```

Note that the recommended procedure for setting the primary key is the indirect method.

Modify the populate() method: You must modify the populate method, by doing the following:

1. Expand your new data bean to view its fields and methods.
2. In the Outline view, select the **populate()** method to view its source code. It initially appears as follows:

   ```
   public void populate () throws Exception {}
   ```
3. Modify the source code so the method appears as follows:

   ```
   try {
      super.refreshCopyHelper();
     } catch (javax.ejb.FinderException e) {
         throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
        "UserResDataBean", "populate");
   ```

   Save your work (Ctrl+S).

Note that if the new data bean is extending an access bean that requires additional input parameters upon instantiation, you must also set those values in the populate method of the data bean.

Writing new session beans

When creating new session beans, create them in the WebSphereCommerceServerExtensionsData project.

Your new session bean should extend the com.ibm.commerce.base.helpers.BaseJDBCHelper class. The superclass provides methods that allow you to obtain a JDBC connection object from the data source object used by the WebSphere Commerce Server, so that the

session bean participates in the same transaction as the other entity beans. The following is an example of code to demonstrate the functions provided by the superclass:

```
public class mySessionBean extends com.ibm.commerce.base.helpers.BaseJDBCHelper
   implements SessionBean {

   public Object myMethod () throws javax.naming.NamingException,
      SQLException {

       ////////////////////////////////////////////////
      //  -- your logic, such as initialization --    //
      ////////////////////////////////////////////////

         try {
             // get a connection from the WebSphere Commerce data source
             makeConnection();
             PreparedStatement stmt = getPreparedStatement(
                "your sql string");
             ////////////////////////////////////////////////////////////
             // -- your logic such as set parameter into the prepared  //
             // statement --                                           //
             ////////////////////////////////////////////////////////////
             ResultSet rs = executeQuery(stmt, false);

             ////////////////////////////////////////////////
             // -- your logic to process the result set -- //
            ////////////////////////////////////////////////

                }
          finally {
             // return the connection to the WebSphere Commerce data source
            closeConnection();
          }

        ////////////////////////////////////////////////////
       // -- your logic to return the result ---          //
       ////////////////////////////////////////////////////

   }

}
```

In the preceding code example, the `executeQuery` method takes two input parameters. The first is a prepared statement and the second is a boolean flag related to a cache flush operation. Set this flag to `true` if you need the container to flush all entity objects for the current transaction from the cache before executing the query. This would be required if you have performed updates on some entity objects and you need the query to search through these updated objects. If the flag were set to `false` those entity object updates would not be written to the database until the end of the transaction.

You should limit the use of this flush operation and generally set the flag to `false`, except in those cases where it is really required. The flush operation is a resource intensive operation.

Object life cycles

The enterprise beans in the object model include both *independent* and *dependent* objects. An independent object has its own life cycle, controlled directly by the create or remove requests of the business logic invoking the object. A dependent object has a life cycle that is attached to another object, known as the *owner object* (which may also in turn be a dependent object, but further up the association hierarchy, an independent object exists). When the owner object is deleted, all dependent objects are also deleted. The actual deletes are controlled by cascading delete specifications within the database.

For example, given a user object that returns an address book object and a list of order objects, if the user object is deleted, its address book object is also deleted (since the book is owned by the user), and so are all the address objects within the book (since the addresses are owned by the book). However, the order objects are not deleted because the owner of orders is a store object, not the user object.

A specific design pattern is used for the creation of dependent objects. The create method of a dependent object must supply a reference to its owner object; therefore, the owner object must exist before the dependent object can be created.

Transactions

The Enterprise JavaBeans Version 1.1 architecture specifies three alternative commit-time options with respect to the instance state. They are described as options A, B, and C in the specification document. For complete details on these options, refer to Sun Microystem's Enterprise JavaBeans version 1.1 specification document.

Although the WebSphere Application Server implements options A and C, option A assumes that the database is not shared.

In option C, the enterprise bean container does not cache a "ready" instance between transactions. As soon as a transaction has completed, the instance is returned to the pool of available instances. WebSphere Commerce uses option C because the database is shared across multiple WebSphere Commerce applications. In this implementation, the container loads persistent data for entity beans at the start of each transaction and the entity beans are only cached for the duration of the transaction. The container activates multiple instances of an entity bean, one for each transaction in which the entity is being accessed. Transaction synchronization is performed by the database.

The transaction attribute of each enterprise bean is set to `TX_REQUIRED`. Since the Web controller starts a transaction before executing a command that accesses an enterprise bean (through its corresponding access bean), the business methods of the enterprise bean are invoked within the context of this transaction.

Other considerations for entity beans

Find for update

A situation in which multiple applications can access the same row in a database for the purpose of updating the row, is known as a *concurrent update*. There are situations in which concurrent updates may be allowed, and other situations where they are definitely not desired.

If the database update is an overwrite, where the new value has no relation to the current value in the database, concurrent updates may be allowed. If concurrent update is allowed and multiple applications attempt to update the same row in a database, the last attempt is the one that gets updated in the database.

If the database update depends upon the current value in the database, a concurrent update is not desired. For example, if an application is updating product inventory, only one application should be allowed to update the inventory at a time.

Factors that affect whether or not concurrent update is permitted include database locks and enterprise bean isolation levels.

In order to prevent a second application from concurrently updating a row, the first application accessing the row must fetch the row using the "find for update" option. When the "for update" option is used, a *write lock* (also known as an exclusive lock) is applied to the row. With this write lock applied to the row, any application that attempts to access the row using the "find for update" is blocked.

If your application permits concurrent updates, it can just fetch the data, without locking the row.

Consider the OrderProcess scenario in which UpdateInventory needs to find all the products included in an order and update the inventory accordingly. Since the same products may be included in many other orders, *find for update* should be used, and it should be used as early as possible within a transaction scope to reduce the possibility of deadlocks. Therefore, the UpdateInventory algorithm may be represented by the following pseudo code:

```
UpdateInventory
 find all the order items in the order
 for each order item
  fetch its inventory using "find for update"
  ...
```

In the long-running Business-to-Business scenario, where an order may have many items, find for update should be used as early as possible. The logic may become the following:

```
find for update the inventory of all the products in an order
for each product
 if (total quantity ordered for that product < inventory)
        deduct quantity from inventory
 else
        error
```

Flush remote method

Since WebSphere Application Server does not write changes made on the entity beans to the database until the transaction commit time, the database may get temporarily out of synchronization with the data cached in the entity bean's container.

A flush remote method is provided (in the `com.ibm.commerce.base.helpers.BaseJDBCHelper` class) that writes all the committed changes made in all transactions (that is, it takes information from the enterprise bean cache) and updates the database. This remote method can be called by a command. Use this method, only when absolutely required, since it is expensive in terms of overhead resources, and therefore, has a negative impact on performance.

Consider a logon command that has the following piece of code:

```
UserAccessBean uab = ...;
 uab.setRegisteredTimestamp(currentTimestamp);
 uab.commitCopyHelper();
```

Before the transaction has been committed, the REGISTRATIONUPDATE in the USERS table will not have been updated with the current time stamp. The update only occurs at transaction commit time. The flush method has to be used so that any direct JDBC query (in the same transaction), for example, *select from user where registeredstamp ...* returns the user with the specified registration time stamp.

Securing enterprise beans

If you are using the WebSphere Application Server for securing enterprise beans, you must assign the `WCSecurityRole` role to the methods of any new enterprise beans. The WebSphere Application Server Application Assembly Tool is used to perform this task. Perform this step when you are deploying the new enterprise beans. Additionally, if you modify existing WebSphere

Commerce entity beans, you must assign the `WCSecurityRole` role to each method in each of the entity beans in the modified EJB project.

For a description of the deployment process for customized code, refer to Chapter 9, “Deployment details,” on page 201.

Primary keys

A primary key is a unique key that is part of the definition of a table. It can be used to distinguish one record from others. All records must have a primary key. When you create a new record in a table, you may need to generate a unique primary key for the record.

In the WebSphere Commerce programming model, the persistence layer includes entity beans that interact with the database. As such, database records may be created when an entity bean is instantiated. Therefore, the ejbCreate method for the instantiation of an entity bean may need to include logic to generate a primary key for new records.

When an application requires information from the database, it indirectly uses the entity beans by instantiating the bean’s corresponding access bean and then getting or setting various fields. An access bean is instantiated for a particular record in a database (for example, for a particular user profile) and it uses the primary key to select the correct information from the database.

The following sections describe how to create a unique primary key and how to select by primary key.

Creating primary keys: The `ejbCreate` method is used to instantiate new instances of an entity bean. This method is automatically generated but the generated method only includes logic to initialize primary keys to a static value.

You may need to ensure that the primary key is a new, unique value. In this case, you may have an ejbCreate method similar to the following code snippet:

```
public void ejbCreate(int argMyOtherValue)
   throws javax.ejb.CreateException {
      //Initialize CMP fields
      MyKeyValue = com.ibm.commerce.key.ECKeyManager.
         singleton().getNextKey("table_name");
      MyOtherValue = argMyOtherValue;
}
```

In the preceding code snippet, the getNextKey method generates a unique integer for the primary key. The *`table_name`* input parameter for the method must be an exact match to the TABLENAME value that is defined in the KEYS table. Be certain to match the characters and case exactly.

In addition to including the preceding code in your ejbCreate method, you must also create an entry in the KEYS table. The following is an example SQL statement to make the entry in the KEYS table:

```
insert into KEYS (TABLENAME, COUNTER, KEYS_ID)
   values ("table_name", 0, 1)
```

Note that with the preceding SQL statement default values for the other columns in the KEYS table are accepted. The value for COUNTER indicates the value at which the count should start. The value for KEYS_ID should be any positive value.

If your primary key is defined as a long data type (BIGINT for DB2 or NUMBER(38, 0) for Oracle), use the `getNextKeyAsLong` method.

Selecting by primary key: Within an access bean, you must select the appropriate database record by using the primary key. The following code snip demonstrates how to perform this select. It also includes additional logic, that is explained later.

```
UserProfileAccessBean abUserProfile = new UserProfileAccessBean();
abUserProfile.setInitKey_UserId(getUserId().toString());
abUserProfile.refreshCopyHelper();
```

The first line in the preceding code snippet instantiates a new UserProfileAccessBean that is called "abUserProfile". The second line sets the primary key in the access bean. The `setInitKey_`*`xxx`* (where *`xxx`* is the primary key field name) naming convention is used by WebSphere Studio Application Developer to name the set methods for primary keys. When instantiating an access bean, you should ensure that all fields set by a setInitKey_xxx method are initialized before using the refreshCopyHelper method. The order in which the setInitKey_xxx methods are called is not important.

After all setInitKey_xxx methods have been called, you have initialized all required fields and can use the refreshCopyHelper method to retrieve information from the database.

If you update values in the local cache of the access bean, you must also include a commitCopyHelper call to update the database with the updated information. For example, if after retrieving data using the refreshCopyHelper method you update a customer's name (by setting the name value) you must then call `abUserProfile.commitCopyHelper()` to update the database with the new information.

Using entity beans

A program that uses enterprise beans must deal with the Java Naming and Directory Interface (JNDI) as well as the home and remote interfaces of enterprise beans. To simplify the programming model, an access bean for each enterprise bean is generated. When creating your own enterprise beans, use the tooling in WebSphere Studio Application Developer to generate this access bean.

WebSphere Commerce commands interact with access beans rather than directly with the entity beans. As the diagram illustrates, using the access bean provides the following advantages:

- A simpler programming interface. The access bean behaves like a Java bean and hides all the enterprise bean specific programming interfaces, like JNDI, home and remote interfaces from the clients.
- At run time the access bean caches the enterprise bean home object because look ups to the home object are expensive, in terms of time and resource usage.
- The access bean implements a copyHelper object that reduces the number of calls to the enterprise bean when commands get and set enterprise bean attributes. Therefore, only a single call to the enterprise bean is required, when reading or writing multiple enterprise bean attributes.

The following diagram displays the interaction between commands, access beans, entity beans and the database.

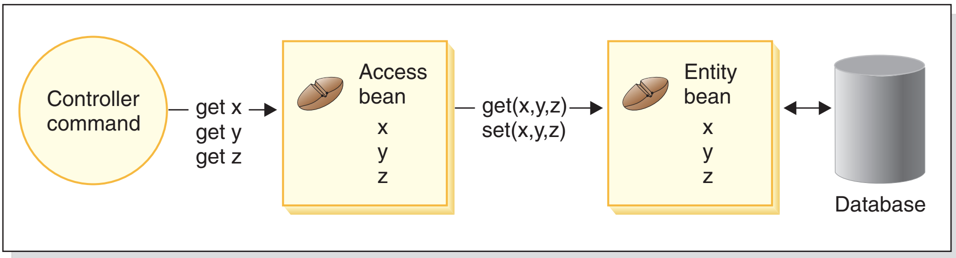


Figure 18.

Database considerations

As you customize your e-commerce application, you may create new database tables. When creating these tables, it is recommended that you follow a set of conventions, so that your tables are created in a manner consistent with the WebSphere Commerce tables.

Database schema object naming considerations

Subsequent sections provide guidance for the naming of database schema objects.

Naming conventions for tables and views

The following list provides guidance for the naming of new tables and views:

- In order to avoid name collision (duplicate names) with WebSphere Commerce tables and views in future releases, the first character in the table or view name should be `X`. For example, `XMYTABLE`.
- The table or view name should be no more than 10 characters in length. If the desired name exceeds this limit, shorten the length by removing vowels from the end of the name, until only 10 characters remain.
- The table or view name should not contain any special characters, such as "_", "+", "$", "%", or blank spaces.
- Do not use database reserved words as a table or view name.
- View names should end with `VW`.
- The table and view names should be singular nouns.

Naming conventions for columns

In general, when you create new tables that follow the preceding conventions for table names, you can implement your own naming convention for columns within those tables. This assumes that you always use fully-qualified column names in SQL statements. If, however, you wish to perform joins with existing WebSphere Commerce tables and you do not want to use fully-qualified column names, then you must follow the column naming conventions described in this section.

The following list provides guidance for the naming of columns in new tables:

- In order to avoid name collision (duplicate names) with columns in WebSphere Commerce tables in future releases, the first character in the column name should be `X`. For example, `XMYCOLUMN`.
- The column name should be no more than 18 characters in length. If the desired name exceeds this limit, shorten the length by removing vowels from the end of the name, until there are only 18 characters.
- Columns names (other than foreign keys) should not contain any special characters, such as "_", "+", "$", "%", or blank spaces.
- Do not use database reserved words as a column name.
- Combined words may be used as column names using the active voice combination. For example, `COMBINERESULT`.
- The generated primary key columns should be named as *`table`*`_id`. For example, the primary key for the USERS table is `USERS_ID`.
- The generated foreign key column names should not be changed.

- If reserving any columns for future customization, they should be named `field`*x* where *x* is a numeric digit starting from 1.

Naming conventions for indexes

The following list provides guidance for the naming of indexes in new tables:

- The index name should be no more than 18 characters in length.
- The index name should not contain blank spaces.
- The index name should not contain any database reserved words.
- A non-unique index should be named as `I_`*tablex* where *table* is the name of the table and *x* is a number, beginning at 1. For example, a non-unique index for the USERS table is `I_USERS1`.
- A unique index should be named as `UI_`*tablex* where *table* is the name of the table and *x* is a number, beginning at 1. For example, a unique index for the USERS table is `UI_USERS1`.
- The total size of the index should be no larger than 254 bytes.
- The index name must be unique across the whole database schema.

Naming conventions for primary keys

The following list provides guidance for the naming of primary keys for new tables:

- The primary key name should be no more than 18 characters in length.
- The primary key name should not contain blank spaces.
- The primary key name should not contain any database reserved words.
- The primary key should be named as `P_`*table* where *table* is the name of the table. For example, the primary key for the USERS table is `P_USERS`.
- The primary key name must be unique across the whole database schema.

Naming conventions for foreign keys

The following list provides guidance for the naming of foreign keys for new tables:

- The foreign key name should be no more than 18 characters in length.
- The foreign key name should not contain blank spaces.
- The foreign key name should not contain any database reserved words.
- The foreign key should be named as `F_`*table* where *table* is the name of the table. For example, the foreign key for the USERS table is `F_USERS1`.
- The foreign key name must be unique across the whole database schema.

Naming conventions for database triggers

The following list provides guidance for the naming of database triggers:

- The database trigger name should be no more than 18 characters in length.
- The database trigger name should not contain blank spaces.
- The database trigger name should not contain any database reserved words.

- The database trigger should be named as `T_table` where *table* is the name of the table. For example, the database trigger name for the USERS table is `T_USERS1`.
- The database trigger name must be unique across the whole database schema.

Database column data type considerations

This section introduces column data types that can be used when creating new tables. The descriptions of the various data types use DB2 terminology. "Data type differences among databases" on page 86 describes the differences if you are using a different database.

BIGINT

This is a 64-bit signed integer that has a range from -9223372036854775807 to 9223372036854775807. Contrast this to INTEGER, which is only half the size of BIGINT.

INTEGER

This is a 32-bit signed integer that has a range from -2147483647 to 2147483647. In general, INTEGER should be the default finite numeric data type, instead of BIGINT. Unless there is a strong business reason for using BIGINT, for performance reasons it is better to use INTEGER as the numeric data type. A common user of the BIGINT data type is a system generated key.

The use of either SMALLINT or SHORT data types is strongly discouraged because these data types are mapped to a non-object Java data type and these non-object data types will cause problems in some enterprise bean object instantiations.

TIMESTAMP

This is a is a seven-part value (year, month, day, hour, minute, second, and microsecond) that designates a date and time, except that the time includes a fractional specification of microseconds. The internal representation of a timestamp is a string of 10 bytes, each of which consists of 2 packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the last 3 bytes the microseconds.

CHAR

This is a fixed-length character string of length INTEGER, which may range from 1 to 254 characters. If the length specification is omitted, a length of 1 character is assumed. Since CHAR is a fixed length database column, any unused trailing character spaces are changed into white spaces. Unless for performance reasons, it is not recommended to use CHAR data type because CHAR is not flexible and length cannot be changed at a later time. As a rule of thumb, if your string column is less then 64 characters in length and is regularly retrieved or updated, use CHAR instead for better performance.

VARCHAR
: This is a variable-length character string of maximum length integer, which may range from 1 to 32672. However, unlike CHAR where the column data is stored along with the table, VARCHAR is internally represented as a reference pointer inside a database page. Therefore, length of a VARCHAR column can be changed at any time after creation.

LONG VARCHAR
: This is the variable-length character string that can be used if VARCHAR cannot be created within the same database page. LONG VARCHAR is very similar to VARCHAR except that it can span multiple database pages. Restrict the use of the LONG VARCHAR data type to only those cases when it is absolutely required because LONG VARCHAR objects are typically expensive in terms of performance.

CLOB This another variable-length character string that can be used if the length of the column needs to exceed the 32KB limit of LONG VARCHAR. The length of a CLOB object can reach 1 GB without modifying the database configuration. Text data that is stored as CLOB is converted appropriately when moving among different systems.

BLOB This is a variable-length binary character string that stores unstructured data in the database. BLOB objects can store up to 4 GB of binary data. In general, you should avoid using BLOB as a column data type, unless it is absolutely necessary. In terms of performance, a BLOB object is considered to be one of the most expensive objects in any database.

DECIMAL(20,5)
: This data type is specially defined to be used for most fixed decimal point numbers, such as currency units. For other floating point decimal numbers, FLOAT can be used instead.

* Data type differences among databases

* The following table shows the correspondences among the WebSphere
* Commerce database schema data types for the different database
* implementations supported.

DB2 AIX Linux Solaris Windows	DB2 400	DB2 390 z/OS	Oracle AIX Solaris Windows
BLOB()	BLOB()	BLOB()	BLOB
TIMESTAMP	TIMESTAMP	TIMESTAMP	DATE
INTEGER	INTEGER	INTEGER	INTEGER
DECIMAL(,)	DECIMAL(,)	DECIMAL	DECIMAL(,)
BIGINT	BIGINT	DECIMAL	NUMBER(38,0)
FLOAT	FLOAT	DOUBLE	NUMBER(38,0)
CHAR()	GRAPHIC() CCSID 13488	CHAR()	VARCHAR2()
CHAR() for bit data	CHAR() for bit data	BLOB()	RAW()
VARCHAR()	VARGRAPHIC() CCSID 13488	VARCHAR()	VARCHAR2()
LONG VARCHAR	VARGRAPHIC(4000) ALLOCATE() CCSID 13488	VARCHAR(4000)	VARCHAR2() (See notes following table for details.)
LONG VARCHAR for bit data	VARCHAR(8000) ALLOCATE() for bit data	BLOB()	LONG RAW
CLOB()	DBCLOB() CCSID 13488	CLOB()	CLOB()

Notes:

1. As a result of inconsistent rates of success for when the Oracle JDBC driver handles information that is of the LONG data type, it is recommended that you avoid using the LONG data type whenever possible. The most commonly reported error in this situation is the "Stream has already been closed" error.
2. If you must use this data type, you can only have one column per database table that uses the LONG type. In addition, when constructing a select statement, do not put the LONG column as either the first or last element in the select. Another workaround for operations under a heavy load is to avoid mapping this particular column to a CMP field in an entity bean. Instead, use a session bean to perform retrieves and updates on this column.

Chapter 4. Access control

Understanding access control

The access control model of a WebSphere Commerce application has three primary concepts: users, actions and resources. Users are the people that use the system. Resources are the entities that are maintained in or by the application. For example, resources may be products, documents, or orders. User profiles that represent people are also resources. Actions are the activities that users can perform on the resources. Access control is the component of the e-commerce application that determines whether a given user can perform a given action on a given resource.

In a WebSphere Commerce application, there are two main levels of access control. The first level of access control is performed by the WebSphere Application Server. In this respect, WebSphere Commerce uses WebSphere Application Server to protect enterprise beans and servlets. The second level of access control is the fine-grained access control system of WebSphere Commerce.

The WebSphere Commerce access control framework uses access control policies to determine if a given user is permitted to perform a given action on a given resource. This access control framework provides fine-grained access control. It works in conjunction with, but does not replace the access control provided by the WebSphere Application Server.

Overview of resource protection in WebSphere Application Server

The following WebSphere Commerce resources are protected under access control by WebSphere Application Server:

- Entity beans
  These beans model objects in an e-commerce application. They are distributed objects that can be accessed by remote clients.
- JSP templates
  WebSphere Commerce uses JSP templates for display pages. Each JSP template can contain one or more data beans that retrieve data from entity beans. Clients can request JSP pages by composing a URL request.
- Controller and view commands
  Clients can request controller and view commands by composing URL requests. In addition, one display page may contain a link to another by using the JSP file name or the view name, as registered in the VIEWREG table.

© Copyright IBM Corp. 2000, 2003

The WebSphere Commerce Server is typically configured to use the following Web paths:

- /webapp/wcs/stores/servlet/*
  This is used for requests to the request servlet.
- /webapp/wcs/stores/*.jsp
  This is used for requests to the JSP servlet.

The following diagram shows the route that requests could potentially follow to access WebSphere Commerce resources, for the preceding Web path configuration.

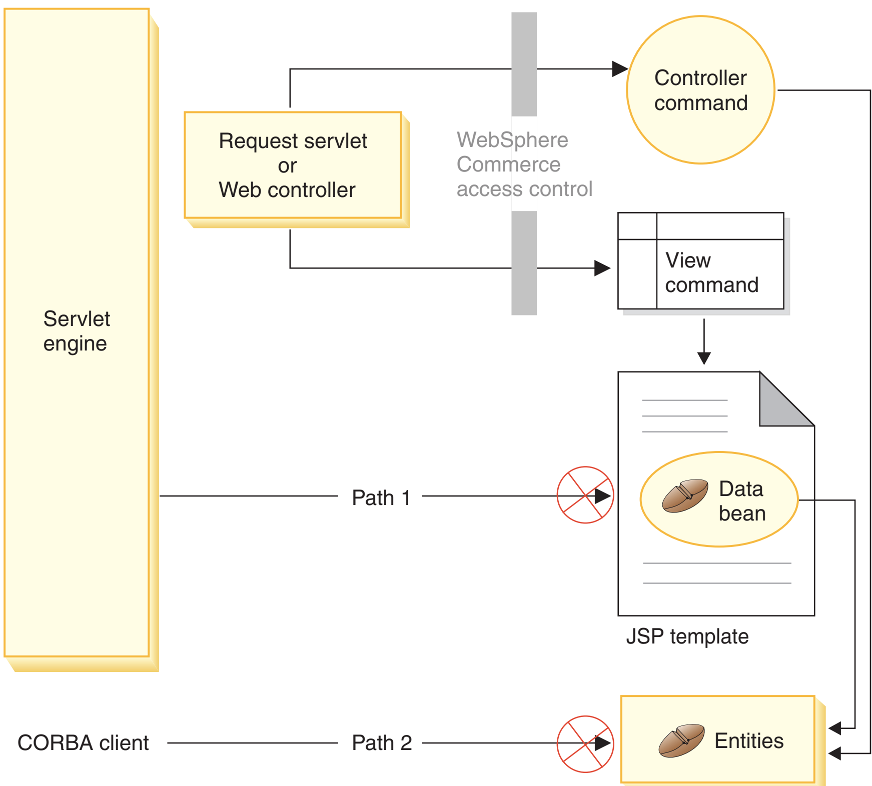


Figure 19.

All the legitimate requests should be directed to the request servlet, which then directs them to the Web controller. The Web controller implements access control for controller commands and views. The Web paths shown above do, however, make it possible for malicious users to directly access JSP templates (path 1) and entity beans (path 2). In order to prevent these malicious attacks from being successful, they must be rejected at run time.

Direct access to the JSP templates and entity beans can be prevented by using one of the following approaches:

WebSphere Application Server security

WebSphere Application Server provides a number of security features. WebSphere Commerce uses one of these features to ensure that all enterprise bean methods and JSP templates are configured to be invoked only by a chosen identity. To access these WebSphere Commerce resources, a URL request must be routed to the request servlet. The request servlet sets the chosen identity on the current thread before passing it to the Web controller. The Web controller then ensures that the caller has the required authorization before passing the request to the corresponding controller command or view. Any attempts to directly access JSP templates and entity beans (that is, without using the Web controller) are rejected by the WebSphere Application Server security component.

For information about configuring WebSphere Application Server to secure WebSphere Commerce resources, refer to the *WebSphere Commerce Security Guide*. For information about security within WebSphere Application Server, refer to the System Administration topic in the WebSphere Application Server documentation.

Firewall protection

When a WebSphere Commerce Server runs behind the firewall, Internet clients are not able to directly access the entity beans. Using this approach, protection for JSP templates is provided by the data bean that is included in the page. The data bean is activated by the data bean manager. The data bean manager detects if the JSP template was forwarded by a view command. If it was not forwarded by a view command an exception is thrown and the request for the JSP template is rejected.

Security consideration for URL parameters

URL parameters provide a useful way of passing request specific information. In order to minimize the chance of a malicious user gaining access to your database in an unauthorized manner, a particular coding practice should be followed. Insert, select, update, and delete parts of SQL statements should be created at development time. Parameter inserts should be used to gather run-time input information.

An example of using a parameter insert to collect run-time input information follows:

```
select * from Order where owner =?
```

In contrast, you should avoid using input strings as a way to compose the SQL statement. An example of using an input string follows:

```
select * from Order where owner = "input_string"
```

The reason for avoiding the use of the input string to compose the select is that the input string is easy to manipulate. A malicious user can set such an input string to an unexpected value and potentially gain unauthorized access, or perform undesired operations against the database.

Introduction to WebSphere Commerce access control policies

This section provides a very brief introduction to the main components of the WebSphere Commerce access control framework. It is provided so that some of the access control related programming tasks are viewed in the correct context. If you need information about setting up access control on a production system, or more details on the access control framework, refer to the *WebSphere Commerce Security Guide*.

The WebSphere Commerce access control model is based upon the enforcement of access control policies. Access control policies allow access control rules to be externalized from business logic code, thereby removing the need to hard code access control statements into code. For example, you do not need to include code similar to the following:

```
if (user.isAdministrator())
   then {}
```

Access control policies are enforced by the access control policy manager. In general, when a user attempts to access a protected resource, the access control policy manager first determines what access control policies are applicable for that protected resource, and then, based upon the applicable access control policies, it determines if the user is allowed to access the requested resources.

An access control policy is a 4-tuple policy that is stored in the ACPOLICY table. Each access control policy takes the following form:

```
AccessControlPolicy [UserGroup, ActionGroup, ResourceGroup, Relationship]
```

The elements in the 4-tuple access control policy specify that a user belonging to a specific user group is permitted to perform actions in the specified action group on resources belonging to the specified resource group, as long as the user satisfies the conditions specified in the relationship or relationship group, with respect to the resource in question. For example, `[AllUsers, UpdateDoc, doc, creator]` specifies that all users can update a document, if they are the creator of the document.

The user group is a specific type of member group that is defined in the MBRGRP database table. A user group must be associated with member group type of `-2`. The value of `-2` represents an access group and is defined in

the MBRGRPTYPE table. The association between the user group and member group type is stored in the MBRGRPUSG table.

The membership of a user into a particular user group may be stated explicitly or implicitly. An explicit specification occurs if the MBRGRPMBR table states that the user belongs to a particular member group. An implicit specification occurs if the user satisfies a condition (for example, all users that fulfill the role of Product Manager) that is stated in the MBRGRPCOND table. There may also be combined conditions (for example, all users that fulfill the role of Product Manager and that have been in the role for at least 6 months) or explicit exclusions.

Most conditions to include a user in a user group are based upon the user fulfilling a particular role. For example, there could be an access control policy that allows all users that fulfill the Product Manager role to perform catalog management operations. In this case, any user that has been assigned the Product Manager role in the MBRROLE table is then implicitly included in the user group.

For more details about the member group subsystem, refer to the WebSphere Commerce Production and Development online help.

The ActionGroup element comes from the ACACTGRP table. An action group refers to an explicitly specified group of actions. The listing of actions is stored in the ACACTION table and the relationship of each action to its action group (or groups) is stored in the ACACTACTGP table. An example of an action group is the ″OrderWriteCommands″ action group. This action group includes the following actions that are used to update orders:

- com.ibm.commerce.order.commands.OrderDeleteCmd
- com.ibm.commerce.order.commands.OrderCancelCmd
- com.ibm.commerce.order.commands.OrderProfileUpateCmd
- com.ibm.commerce.order.commands.OrderUnlockCmd
- com.ibm.commerce.order.commands.OrderScheduleCmd
- com.ibm.commerce.order.commands.ScheduledOrderCancelCmd
- com.ibm.commerce.order.commands.ScheduledOrderProcessCmd
- com.ibm.commerce.order.commands.OrderItemAddCmd
- com.ibm.commerce.order.commands.OrderItemDeleteCmd
- com.ibm.commerce.order.commands.OrderItemUpdateCmd
- com.ibm.commerce.order.commands.PayResetPMCmd

A resource group is a mechanism to group together particular types of resources. Membership of a resource in a resource group can be specified in one of two ways:

- Using the conditions column in the ACRESGRP table
- Using the ACRESGPRES table

In most cases, it is sufficient to use the ACRESGPRES table for associating resources to resource groups. Using this method, resources are defined in the ACRESCGRY table using their Java class name. Then, these resources are associated with the appropriate resource groups (ACRESGRP table) using the ACRESGPRES association table. In cases where the Java class name alone is not sufficient to define the members of a resource group (for example, if you need to further restrict the objects of this class based on an attribute of the resource), the resource group can be defined entirely using the conditions column of the ACRESGRP table. Note that in order to perform this grouping of resources based on an attribute, the resource must also implement the Groupable interface.

The following diagram shows an example resource grouping specification. In this example resource group `10023` includes all the resources that are associated with it in the ACRESGPRES table. Resource group `10070` is defined using the conditions field column in the ACRESGRP table. This resource group includes instances of the Order remote interface, that also have status = `"Z"` (specifying a shared requisition list).

Note: Details about the XML information for the Conditions column of the ACRESGRP table are found in the *WebSphere Commerce Security Guide*.

ACRESGRP

AcResGrp_Id	GrpName	Conditions
10023	AccountRepresentatives CmdResourceGroup	null
10070	SharedRequisitionList ResourceGroup	<profile> <andListCondition> <simpleCondition> <variable name="Status"/> <operator name="="/> <value data="Z"/> </simpleCondition> <simpleCondition> <variable name="classname"/> <operator name="="/> <value data="com.ibm.commerce.order. objects.Order"/> </simpleCondition> </andListCondition> </profile>

ACRESGRPES

AcResGrp_Id	AcResCgry_Id
10023	10246
10023	10247
10023	10248
10023	10249
10023	10250

ACRESCGRY

AcResCgry_Id	ResClassname
10246	com.ibm.commerce.contract. commands.ContractCreateCmd
10247	com.ibm.commerce.contract. commands.ContractUpdateCmd
10248	com.ibm.commerce.contract. commands.ContractDeleteCmd
10249	com.ibm.commerce.contract. commands.ContractCancelCmd
10250	com.ibm.commerce.contract. commands.ContractCloseCmd

Figure 20.

The MEMBER_ID column of the ACACTGRP, ACRESGRP, and ACRELGRP tables should have a value of `-2001` (Root Organization).

The access control policy can optionally include either a Relationship or RelationshipGroup element as its fourth element.

If your access control policy uses a Relationship element, this comes from the ACRELATION table. If, on the other hand, it includes a RelationshipGroup element, that comes from the ACRELGRP table. Note that neither need be

included, but if you include one, you cannot include the other. A RelationshipGroup specification from the ACRELGRP table takes precedence over the Relationship information from the ACRELATION table.

The ACRELATION table specifies the types of relationships that exist between users and resources. Some examples of types of relationships include creator, submitter, and owner. An example of the use of the relationship element is to use it to ensure that the creator of an order can always update the order.

The ACRELGRP table specifies the types of relationship groups that can be associated with particular resources. A relationship group is a grouping of one or more relationship chains. A relationship chain is a series of one more relationships. An example of a relationship group is to specify that a user must be the creator of the resource and also belong to the buying organizational entity that is referenced in the resource.

The relationship group (or relationship) specification is an optional part of the access control policy. It is commonly used if you have created your own commands and these commands are not restricted to certain roles. In these cases, you might want to enforce a relationship between the user and the resource. Typically, if commands are to be restricted to certain roles, it is accomplished through the UserGroup element of the access control policy rather than by using the Relationship element.

Another important concept related to access control policies is the concept of access control policy groups. WebSphere Commerce Version 5.5 supports various business models, and each business model has its own set of access control policies. In order to group the sets of policies within the models, policy groups are used. Policies are explicitly assigned to appropriate policy groups and then organizations can subscribe to one or more of these policy groups. In previous versions of WebSphere Commerce, a policy applied to all resources owned by the descendants of that policy's owner organization.

In WebSphere Commerce Version 5.5, if an organization subscribes to one or more policy groups, only the policies in those policy groups will apply to the resources of that organization. If a resource is owned by an organization that does not subscribe to any policy groups, the access control policy manager will search up the organization hierarchy until it encounters the closest ancestor organization that subscribes to at least one policy group; once found, it will apply the policies belonging to those policy groups.

Consider the following diagram:

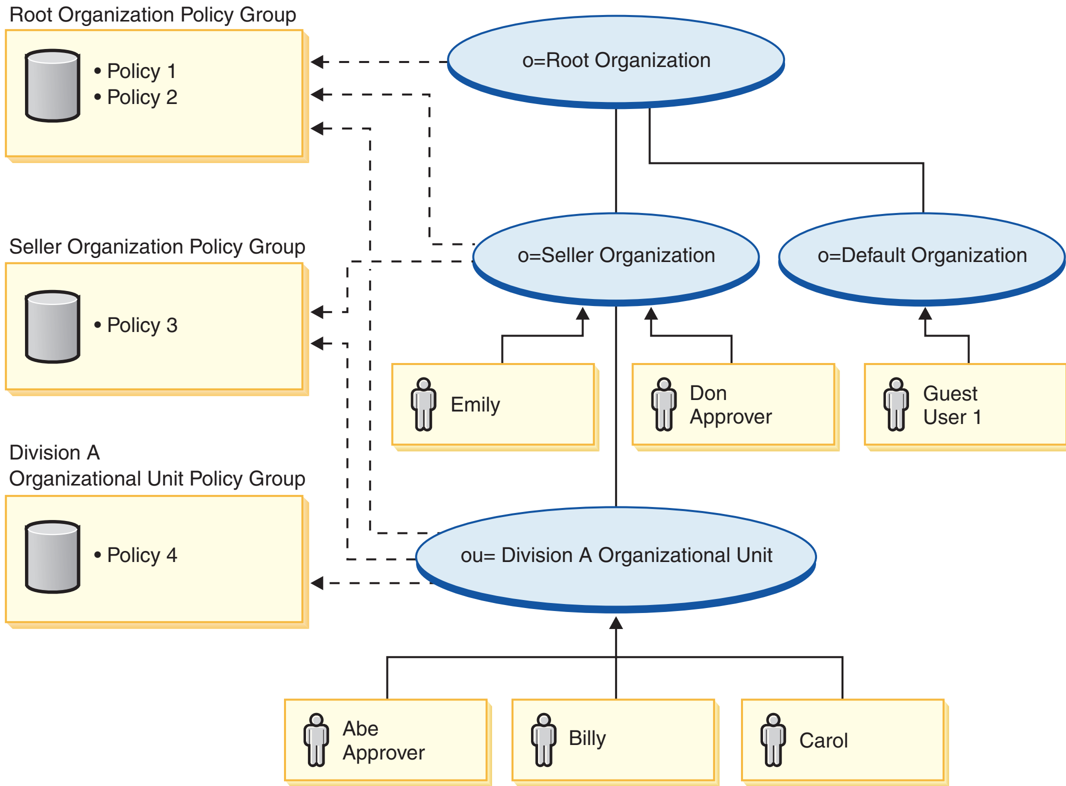


Figure 21.

For any resources owned by the Seller Organization, the policies in the Seller Organization policy group and those in the Root Organization policy group apply. These policies are applicable because the Seller Organization explicitly subscribes to those two policy groups (the dotted arrows show the subscription). In total, three policies apply for this organization. This example is one where the organization that owned the resource explicitly subscribes to policy groups. The diagram also shows an example where the organization does not explicitly subscribe to any policy groups, so the access control framework must look farther up in the hierarchy. For resources that are owned by Default Organization, one must go up the hierarchy to the Root Organization. The Root Organization only subscribes to one policy group, so those are the only policies (policies 1 and 2) that apply to the resources of the Default Organization.

Relationship groups

A relationship group allows you to specify multiple relationships. A relationship can be directly between a user and the resource in question, or it can be a chain of relationships that indirectly relate the user to the resource.

Note: Express Professional For the following sections related to relationship groups, it is important to recognize that the only organizations available in WebSphere Commerce Professional Edition and WebSphere Commerce - Express are the RootOrganization, the DefaultOrganization, and the SellerOrganization. Business Examples that refer to other organizations only apply to WebSphere Commerce Business Edition.

Comparing relationships to relationship groups: Access control policies can specify that a user must fulfill a particular relationship with respect to the resource being accessed, or they can specify that a user must fulfill the conditions specified in a relationship group.

In most cases, specifying a relationship should satisfy the access control requirements for your application. If, however, the policy is such that you must specify a relationship that is not directly between the user and the resource, but that is actually a series of relationships between the user and the resource, you must then use a relationship group.

For example, if you must specify an association between a user and a buying organization where the relationship requires that the user is playing a particular role for that organization or that the user is a member of the buying organization, then you must use a relationship group and a chain of relationships.

If you merely need to enforce an association that is directly between the user and the resource in question, you can use a simple relationship. For example, this would be the case if you need to enforce that the user must be the creator of the resource.

If you combine multiple simple relationships, for example, the user must be the creator *or* the submitter, then this becomes a chain of relationships and you must use a relationship group. This combination of simple relationships may occur when using either WebSphere Commerce Professional Edition or WebSphere Commerce Business Edition.

General information about relationship groups: A relationship chain is a series of one more relationships. The length of a relationship chain is determined by the number of relationships that it contains. This can be determined by examining the number of `<parameter name="aName" value="aValue" />` elements in the XML representation of the relationship chain.

Only the last `<parameter name="Relationship" value="aValue" />` element must be handled by the fulfills() method of the resource. The rest are handled internally by the access control policy manager.

When a relationship chain has a length of 2, the first `<parameter name="aName" value="aValue" />` element is between a user and an organizational entity. The last `<parameter name="aName" value="aValue" />` element is between an organizational entity and the resource.

If you need to define relationship groups, you must do so by defining the relationship group information in an XML file. You can create your own XML file based on the defaultAccessControlPolicies.xml file. For more information about creating this XML-based information, refer to the *WebSphere Commerce Security Guide*.

Types of access control

There are two types of access control, both of which are policy-based: command-level access control and resource-level access control.

Command-level (also known as “role-based”) access control uses a broad type of policy. You can specify that all users of a particular role can execute certain types of commands. For example, you can specify that users with the Account Representative role can execute any command in the AccountRepresentativesCmdResourceGroup resource group. Or, as depicted in the following diagram, another example policy is to specify that all store administrators can perform any action specified in the ExecuteCommandAction Group on any resource that is specified by the StoreAdminCmdResourceGrp.

Note: The XML information for the Conditions column of the MBRGRPCOND table is generated when you use the Administration Console to set up your access groups. For information about using the Administration Console to set up access groups, refer to the WebSphere Commerce Production online help.

ACPOLICY

PolicyName	Member_Id	MbrGrp_Id	AcActGrp_id	AcResGrp_Id	AcRelGrp_Id
StoreAdministrators ExecuteStoreAdministrators CmdResourceGroup	-2001	-8	10052	10018	null

MBRGRP

MbrGrp_Id	MbrGrpName
-8	StoreAdministrators

MBRGRPCOND

MbrGrp_Id	Conditions
-8	<profile> <simpleCondition> <variable name="role"/> <operator name="="/> <value data="Store Administrator"/> </simpleCondition> </profile>

ACACTGRP

AcActGrp_Id	GroupName
10052	ExecuteCommandActionGroup

ACRESGRP

AcResGrp_Id	GrpName
10018	StoreAdministratorsCmdResourceGroup

Figure 22.

A command-level access control policy always has the `ExecuteCommandActionGroup` as the action group for controller commands. For views, the resource group is always `ViewCommandResourceGroup`.

All controller commands must be protected by command-level access control. In addition, any view that can be called directly, or that can be launched by a redirect from another command (in contrast to being launched by forwarding to the view) must be protected by command-level access control.

Command-level access control does not consider the resource that the command would act upon. It merely determines if the user is allowed to execute the particular command. If the user is allowed to execute the

command, a subsequent resource-level access control policy could be applied to determine if the user can access the resource in question.

Consider when a store administrator attempts to perform an administrative task. The first level of access control checking would be to determine if this user is allowed to execute the particular store administration command. Once it has been determined that the user is in fact permitted to do this (because store administrators are allowed to execute commands in the storeAdminCmds group), a resource-level access control policy may be invoked. This policy may state that store administrators are only permitted to perform administrative tasks for stores that are owned by the organization for which the user is a store administrator.

To summarize, in command-level access control the “resource” is the command itself and the “action” is merely to execute the command (in other words, to instantiate the command object). The access control check determines if the user is permitted to execute the command. By contrast, in resource-level access control the “resource” is any protectable resource that the command or bean accesses and the “action” is the command itself.

Access control interactions

This section presents an interaction diagram showing how access control works in the WebSphere Commerce access control policy framework.

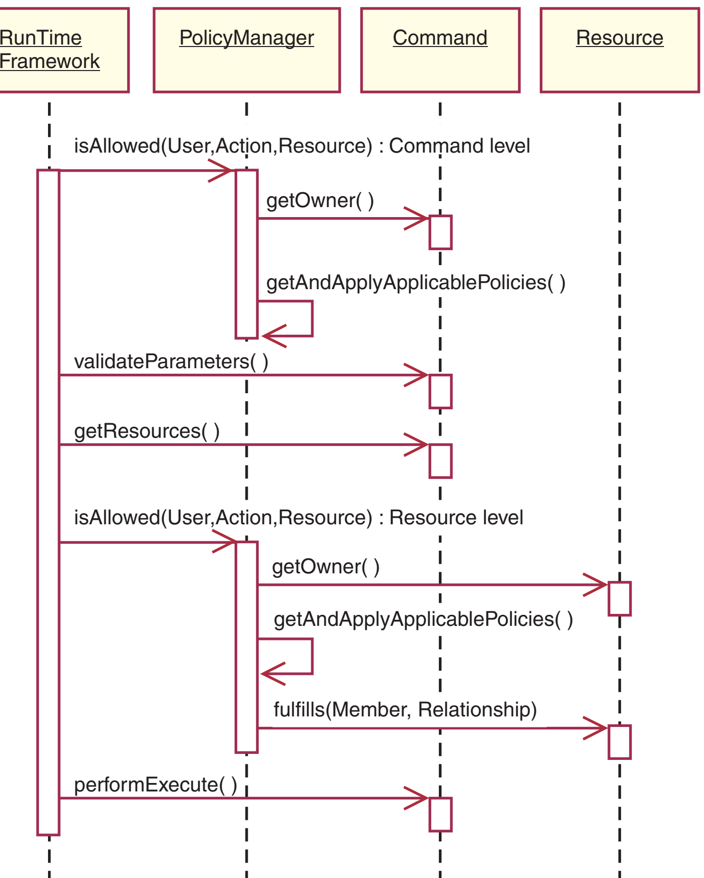


Figure 23.

The preceding diagram shows actions that are performed by the access control *policy manager*. The access control policy manager is the access control component that determines whether or not the current user is allowed to execute the specified action on the specified resource. It determines this by searching through the policies in groups to which the resource owner subscribes. If the resource owner does not subscribe to any policy groups, it searches through the policies in groups to which the resource owner's closest ancestor subscribes. If at least one policy grants access, then permission is granted.

The following list describes the actions from the preceding interaction diagram. They are ordered from the top of the diagram to the bottom.

1. `isAllowed()`
   The run-time components determine if the user has command-level access for either the controller command or view.

2. `getOwner()`
   The access control policy manager determines the owner of the command-level resource. The default implementation returns the member identifier (`memberId`) of the owner of the store (`storeId`) that is in the command context. If there is no store identifier in the command context, then the root organization (`-2001`) is returned.
3. `getAndApplyApplicablePolicies()`
   The access control policy manager finds and processes the applicable policies, based on the specified user, action and resource. If at least one applicable policy grants access, the command-level access check passes, and the policy manager will continue to the next step to begin checking for resource-level authorization. Conversely, if none of the applicable policies grant command-level access, the policy manager returns at this point and denies access.
4. `validateParameters()`
   Initial parameter checking and resolving.
5. `getResources()`
   Returns an access vector that is a vector of resource-action pairs.

   If nothing is returned, resource-level access control checking is not performed. If there are resources that should be protected an access vector (consisting of resource-action pairs) should be returned.

   Each *resource* is an instance of a protectable object (an object that implements the `com.ibm.commerce.security.Protectable` interface). In many cases, the resource is an access bean.

   An access bean may not implement the `com.ibm.commerce.security.Protectable` interface, however, the access control check can still occur as long as the corresponding enterprise bean is protected, according to the information included in “Implementing access control in enterprise beans” on page 106.

   The *action* is a string representing the operation to be performed on the resource. In most cases, the action is the interface name of the command.
6. `isAllowed()`
   The run-time components determine if the user has resource level access to all of the resource-action pairs specified by `getResources()`.
7. `getOwner()`
   The resource returns the `memberId` of its owner. This determines which policies apply.
8. `getAndApplyApplicablePolicies()`
   The access control policy manager searches for applicable policies and then applies them. If at least one policy per resource-action pair that grants the user permission to access the resource is found, then access is granted, otherwise access it is denied.

9. `fulfills()`
   If an applicable policy has a relationship or relationship group specified, a check is done on the resource to see if the member satisfies the specified relationship, with respect to the resource.
10. `performExecute()`
    The business logic of the command.

Protectable interface

A key factor for having a resource protected by the WebSphere Commerce access control policies, is that the resource must implement the `com.ibm.commerce.security.Protectable` interface. This interface is most commonly used with enterprise beans and data beans, but only those particular beans that require protection need to implement the interface.

With the Protectable interface, a resource must provide two key methods: `getOwner()`, and `fulfills(Long member, String relationship)`.

Access control policies are owned by organizations or organizational entities. The `getOwner` method returns the memberId of the owner of the protectable resource. After the access control policy manager determines the owner of the resource, it also gets the memberId of each of the ancestors for the owner in the member hierarchy. All access control policies that belong to the owner from the original getOwner request as well as all access control policies that belong to any of the owner's ancestors are then applied.

Access control policies that apply to the specified owner, as well as access control policies that apply to any of the owner's higher level ancestors in the membership hierarchy, are applied.

The `fulfills` method only returns true if the given member satisfies the required relationship with respect to the resource. Typically the member is a single user, however it can also be an organization. It would be an organization if you are using a relationship group in the access control policy.

Groupable interface

The application of an access control policy is specific to a group of resources. Resource groupings can be made based upon attributes such as the class name, the state of an order or the storeId value.

If a resource is going to be grouped by an attribute other than its class name for the purpose of applying access control policies, it must implement the `com.ibm.commerce.grouping.Groupable` interface.

The following code snippet represents the Groupable interface:

```
Groupable interface {
 Object getGroupingAttributeValue (String attributeName, GroupContext context)
}
```

For example, to implement a policy that only applies to orders that are in the pending state (`status = P (pending)`), the remote interface of the Order entity bean implements the Groupable interface and the value for attributeName is set to `"status"`.

Usage of the Groupable interface is rare.

Finding more information about access control

For more information about the WebSphere Commerce access control model, refer to the *WebSphere Commerce Security Guide*. This guide provides a detailed overview of access control and describes how to use the Administration Console to create or modify policies, action groups, and resource groups.

Implementing access control

This section describes how to implement access control in customized code.

Identifying protectable resources

In general, enterprise beans and data beans are resources that you may want to protect. However, not all enterprise beans and data beans should be protected. Within the existing WebSphere Commerce application, resources that require protection already implement the protectable interface. Typically, the question of what to protect comes into play when you create new enterprise beans and data beans. Deciding which resources to protect depends upon your application.

If a command returns an enterprise bean in the `getResources` method, then the enterprise bean must be protected because the access control policy manager will call the getOwner method on the enterprise bean. The `fulfills` method will also be called if a relationship is specified in the corresponding resource-level access control policy.

If you were to implement the protectable interface (and therefore, put the resource under protection) for all of your own enterprise beans and data beans, your application could require many policies. As the number of policies increases, performance may degrade and policy management becomes more challenging.

A theoretical distinction is made between primary resources and dependent resource. A *primary resource* can exist upon its own. A *dependent resource* exists only when its related primary resource exists. For example, in the out-of-the-box WebSphere Commerce application code, the Order entity bean is a protectable resource, but the OrderItem entity bean is not. The reason for

this is that the existence of an OrderItem depends upon an Order -- the Order is the primary resource and the OrderItem is a dependent resource. If a user should have access to an Order, it should also have access to the items in the order.

Similarly, the User entity bean is a protectable resource, but the Address entity bean is not. In this case, the existence of the address depends on the user, so anything that has access to the user, should also have access to the address.

Primary resources should be protected, but dependent resources often do not require protection. If a user is allowed to access a primary resource, it makes sense that, by default, the user should also be allowed to access its dependent resources.

Implementing access control in enterprise beans

If you create new enterprise beans that require protection by access control policies, you must do the following:

1. Create a new enterprise bean, ensuring that it extends from `com.ibm.commerce.base.objects.ECEntityBean`.
2. Ensure that the remote interface of the bean extends the `com.ibm.commerce.security.Protectable` interface.
3. If a resource is going to be grouped by an attribute other than its Java class name for the purpose of applying access control policies, the remote interface of the bean must also extend the `com.ibm.commerce.grouping.Groupable` interface.
4. The enterprise bean class inherits inherits default implementations for the following methods from com.ibm.commerce.base.objects.ECEntityBean:
   - `getOwner`
   - `fulfills`
   - `getGroupingAttributeValue`

   Override any methods that you need. At a minimum, you must override the `getOwner` method.

   The fulfills method must be implemented if there is an access control policy that includes this resource in its resource group, and also specifies a relationship or relationship group. The getGroupingAttributeValue method must be implemented if there is an access control policy with an implicit resource group that includes certain instances of this resource, based on specific attribute values (for example, if there were an access control policy that pertains only pertains to Orders with status = ’P’ (pending)).

Note that if the only relationship needed is "owner", then you do not need to override the fulfills method. In this case, the policy manager will make use of the result of the getOwner() method.

The default implementations of these methods are shown in the following code snippets. These implementations come from the ECEntityBean class.

```
***********************************************************************
public Long getOwner() throws Exception
{
   return null;
}
***********************************************************************
***********************************************************************
public boolean fulfills(Long member, String relationship)
   throws Exception
{
      return false;
}
***********************************************************************
***********************************************************************
public Object getGroupingAttributeValue(String attributeName,
   GroupingContext context) throws Exception
{
      return null;
}
***********************************************************************
```

The following are sample implementations of these methods based on the implementations used in the OrderBean bean:

- For the getOwner method, the logic of the provided method is:

```
*********************************************************************
   com.ibm.commerce.common.objects.StoreEntityAccessBean storeEntAB =
      new com.ibm.commerce.common.objects.StoreEntityAccessBean();
   storeEntAB.setInitKey_storeEntityId(getStoreEntityId().toString());
   return storeEntAB.getMemberIdInEJBType();
*********************************************************************
```

- For the fulfills method, the logic of the provided method is:

```
**********************************************************************
if ("creator".equalsIgnoreCase(relationship))
{
   return member.equals(bean.getMemberId());
}
else if ("BuyingOrganizationalEntity".equalsIgnoreCase(relationship))
{
   return (member.equals(bean.getOrganizationId()));
}
else if ("sameOrganizationalEntityAsCreator".
   equalsIgnoreCase(relationship))
{
   com.ibm.commerce.user.objects.UserAccessBean creator =
      new com.ibm.commerce.user.objects.UserAccessBean();
```

```
      creator.setInitKey_MemberId(bean.getMemberId().toString());
      com.ibm.commerce.user.objects.UserAccessBean ab =
         new com.ibm.commerce.user.objects.UserAccessBean();
      ab.setInitKey_MemberId(member.toString());
      if (ab.getParentMemberId().equals(creator.getParentMemberId()))
         return true;
   }
   return false;
   ************************************************************************
```

- For the getGroupingAttributeValue method, the logic of the provided method is:

```
   ************************************************************************
      if (attributeName.equalsIgnoreCase("Status"))
          return getStatus();
      return null;
    ************************************************************************
```

5. Create (or recreate) the enterprise bean's access bean and generated code.

Note that if you examine other WebSphere Commerce public entity beans to understand how the getOwner, fulfills and getGroupingAttributeValue methods are implemented, you will notice that these methods are implemented in the access helper class for the beans. As a result of the fact that the methods are implemented in the access helper classes instead of directly in the bean class, the method signatures are slightly different. In particular, for the methods take an extra input parameter for the object itself to be passed into the access helper.

You *must* ensure that when you create new beans, you implement these methods directly in the bean class. Additionally, you must not modify any of those methods in the access helper classes of the WebSphere Commerce public entity beans.

Implementing access control in data beans

If a data bean is to be protected, it can either be directly, or indirectly protected by access control policies. If a data bean is directly protected, then there exists an access control policy that applies to that particular data bean. If a data bean is indirectly protected, it delegates protection to another data bean, for which an access control policy exists.

To determine if a data bean should be protected, consider the following:

1. Is the information in the data bean information that requires protection? For example, is it something of a private nature? If not, direct protection by access control is not required. If yes, continue.
2. Data beans are instantiated by views. Will the view instantiate the data bean by using information from the command context (or some other

predetermined information)? If yes, and access control has already determined that access is allowed, directly protecting this data bean would not be required. If no, continue.

3. Will the view instantiate the data bean by using information from some input parameter? In this case, due to the fact that you are not certain that access control has already determined whether or not the user is allowed access to this information, you should protect the new data bean.

If you create a new data bean that is to be directly protected by an access control policy, the data bean must do the following:

1. Implement the `com.ibm.commerce.security.Protectable` interface. As such, the bean must provide an implementation of the `getOwner()` and `fulfills(Long member, String relationship)` methods.

   When a data bean implements the Protectable interface, the data bean manager calls the `isAllowed` method to determine if the user has the appropriate access control privileges, based upon the existing access control policies. The `isAllowed` method is described by the following code snippet:

   ```
   isAllowed(Context, "Display", protectable_databean);
   ```

   where *protectable_databean* is the data bean to be protected.
2. If resources that the bean interacts with are grouped by an attribute other than the resource's Java class name, the bean must implement the `com.ibm.commerce.grouping.Groupable` interface.
3. Implement the `com.ibm.commerce.security.Delegator` interface. This interface is described by the following code snippet:

   ```
   Interface Delegator {
    Protectable getDelegate();
   }
   ```

   Note: In order to be directly protected, the `getDelegate` method should return the data bean itself (that is, the data bean delegates to itself for the purpose of access control).

The distinction between which data beans should be protected directly versus which should be protected indirectly is similar to the distinction between primary and dependent resources. If the data bean object can exist on its own, it should be directly protected. If the existence of data bean depends upon the existence of another data bean, then it should delegate to the other data bean for protection.

An example of a data bean that would be directly protected is the Order data bean. An example of a data bean that would be indirectly protected is the OrderItem data bean.

If you create a new data bean that is to be indirectly protected by an access control policy, the data bean must do the following:

1. Implement the `com.ibm.commerce.security.Delegator` interface. This interface is described by the following code snippet:

   ```
   Interface Delegator {
    Protectable getDelegate();
   }
   ```

   Note: The data bean returned by `getDelegate` must implement the Protectable interface.

If a data bean does not implement the Delegator interface, it is populated without the protection of access control policies.

Implementing access control in controller commands

When creating a new controller command, the implementation class for the new command should extend the `com.ibm.commerce.commands.ControllerCommandImpl` class and its interface should extend the `com.ibm.commerce.command.ControllerCommand` interface.

For command-level accss control policies for controller commands, the interface name of the command is specified as a resource. In order for a resource to be protected, it must implement the Protectable interface. According to the WebSphere Commerce programming model, this is accomplished by having the command's interface extend from `com.ibm.commerce.command.ControllerCommand` interface, and the command's implementation extend from `com.ibm.commerce.command.ControllerCommandImpl`. The `ControllerCommand` interface extends `com.ibm.commerce.command.AccCommand` interface, which in turn extends `Protectable`. The `AccCommand` interface is the minimum interface that a command should implement in order to be protected by command level access control.

If the command accesses resources that should be protected, create a private instance variable of type `AccessVector` to hold the resources. Then override the `getResources` method since the default implementation of this method returns a null value and therefore, no resource checking occurs.

In the new `getResources` method, you should return an array of resources or of resource-action pairs upon which the command can act. When an action is not explicitly specified, the action defaults to the interface name of the command being executed.

The action only needs to be specified when a command is doing both a read and write operation on different instances of the same resource class. For example, in the OrderCopy command, it can read from a source order and

write to a destination order. In this case, a differentiation must be made between the two actions. This is accomplished by specifying the "-Read" action for the source order, and the "-Write" action for the destination order. When the access control framework detects these actions, it automatically prepends them with the interface name of the command before searching for applicable policies. In this case, the actions that will ultimately be used in the policies are the "com.ibm.commerce.order.commands.OrderCopyCmd-Read" and "com.ibm.commerce.order.commands.OrderCopyCmd-Write" actions.

Additionally, it is recommended that the method determines if it must instantiate the resource or if it can use the existing instance variable holding the reference to the resource. Checking to see if the resource object already exists can help to improve system performance. You can then use the same instance variable, if required, in the `performExecute` method of the new controller command.

To determine if you must override the getResources method, consider the following:

- If you derive the resource based upon a predefined source of information, such as the command context, you would not need to override the getResources method. For example, the WebSphere Commerce UserRegistrationUpdate command derives the user's ID from the command context. In this case, the user is already authorized to act upon their own registration information, so the getResources method does not need to be overridden.
- If your command is arbitrarily specifying a new resource (and this resource is of a private nature), you must override the getResources method. As an example, the WebSphere Commerce OrderItemUpdate command takes an order ID as an input parameter. In this case, when the order resource is instantiated, you do not know if the user has authority to take action upon that particular resource. In this case, the getResources method is overridden.

The following is an example of the `getResources` method:

```
private AccessVector resources = null;

public AccessVector getResources() throws ECException {

   if (resources == null) {
      OrderAccessBean orderAB = new OrderAccessBean();
      orderAB.setInitKey_orderId(getOrderId().toString());
      resouces = new AccessVector(orderAB);
      }
   return resources;
}
```

As an example, consider the OrderItemUpdate command. The `getResources` method of this command returns one or more Order objects (which are protectable), when it is updating existing orders. Since the action is not specified, the action defaults to the interface for the OrderItemUpdate command.

Multiple resources may be returned by the `getResources` method. When this occurs, a policy that gives the user access to all of the specified resources must be found if the action is to be carried out. If a user had access to two out of three resources, the action may not proceed (three out of three are required).

If you need to perform additional parameter checking or resolving of parameters in the controller command, you can use the `validateParameters()` method. Use of this method is optional.

Additional resource level checking

It is not always possible to determine all of the resources that need to be protected, at the time the `getResources` method of the controller command is called.

If necessary, a task command can also implement a `getResources` method to return a list of resources, upon which the command can act.

Another way to invoke resource level checking is to make direct calls to the access control policy manager, using the `checkIsAllowed(Object resource, String action)` method. This method is available to any class that extends from the `com.ibm.commerce.command.AbstractECTargetableCommand` class. For example, the following classes extend from the `AbstractECTargetableCommand` class:

- `com.ibm.commerce.command.ControllerCommandImpl`
- `com.ibm.commerce.command.DataBeanCommandImpl`

The `checkIsAllowed` method is also available to classes that extend the `com.ibm.commerce.command.AbstractECCommand` class. For example, the following class extends from the `AbstractECCommand` class:

- `com.ibm.commerce.command.TaskCommandImpl`

The following shows the signature of the `checkIsAllowed` method:

```
void checkIsAllowed(Object resource, String action)
   throws ECException
```

This method throws an ECApplicationException if the current user is not allowed to perform the specified action on the specified resource. If access is granted, then the method simply returns.

Access control for “create” commands

Since the `getResources` method is called before the `performExecute` method in a command, a different approach must be taken for access control for resources that are not yet created. For example, if you have a `WidgetAddCmd`, the `getResources` method cannot return the resource that is about to be created. In this case, the getResources method should return the container of the new resource. For example, if an order is being created, this is done within the store resource; and a new user is created within an organization resource.

Default implementations for command-level access control

For command-level access control, the default implementation of the getOwner() method returns the memberId of the store owner, if the storeId is specified. If the storeId is not specified, the memberId of the root organization is returned (`memberId = -2001`).

The default implementation of the `getResources()` method returns `null`.

The default implementation of the `validateParameters()` does nothing.

Implementing access control policies in views

Resource-level access control for views is performed by the data bean manager. The data bean manager is invoked in the following cases:

1. When the JSP template includes the `<useBean>` tag and the data bean is not in the attribute list.
2. When the JSP template includes the following activate method:
   ```
   DataBeanManager.activate(xyzDatabean, request);
   ```

Note: Any data bean that is to be protected (either directly or indirectly) must implement the Delegator interface. Any data bean that is to be directly protected will delegate to itself, and thus must also implement the Protectable interface. Data beans that are indirectly protected should delegate to a data bean that implements the Protectable interface.

While it is not recommended, a bypass of the access control checks occurs in the following cases:

1. If the JSP template makes direct calls to access beans, rather than using data beans.
2. If the JSP template invokes the data bean’s populate() method directly.

If the results of a controller command are to be forwarded to a view (using the ForwardViewCommand), then command-level access control is not performed on the views. Furthermore, if the controller command puts the populated data beans (that are used in the view) on the attribute list of the response property and then forwards to a view, the JSP template can access

the data without going through the data bean manager. This does require that the <useBean> tags are used in the JSP template. This can be a way to make a JSP template more efficient, since it can bypass any redundant resource-level access control checks on resources (data beans) to which the user has already been granted access via the controller command.

Modifying access control on existing WebSphere Commerce resources

This section provides information to guide you through modifying access control on existing WebSphere Commerce resources. In particular, the following scenarios are reviewed:

- Adding a new relationship to an existing WebSphere Commerce entity bean that is already protected under access control.
- Adding access control protection to an existing WebSphere Commerce entity bean that is *not* already protected under access control.
- Understanding the implications to access control when extending an existing controller command.

Adding a new relationship to an existing WebSphere Commerce entity bean

WebSphere Commerce entity beans that implement the Protectable interface are already protected under access control. The terms of the access control requirements are determined by the way that the bean is used in the out-of-the-box features and functions of WebSphere Commerce. You may encounter situations in which you need to add additional relationships to access control for such beans. For example, if you use an existing bean in some of your customized code, or if you modify an existing WebSphere Commerce public entity bean, you may need to add additional relationships to the bean.

The following list provides the high-level steps to add new relationships to an existing WebSphere Commerce entity bean that is already protected by access control:

1. Examine the existing fulfills method for the entity bean. This is located in the bean's access helper class. Do not modify this class, use it only to determine if you need to add one or more new relationships to this logic, or if you need to override this method. For example, the following fulfills method appears in the com.ibm.commerce.fulfillment.objsrc.FulfillmentCenterBeanAccessHelper class:

```
public boolean fulfills(Object obj, Long member, String relationship)
   throws Exception {

 FulfillmentCenterBean bean = (FulfillmentCenterBean) obj;

 if ("ShippingArrangementOrganizationalEntity".
```

```
    equalsIgnoreCase(relationship))
  {
      FulfillmentJDBCHelperAccessBean ffmJDBCAB =
         new FulfillmentJDBCHelperAccessBean();
      int count = ffmJDBCAB.
         checkFulfillmentCenterByMemberIdAndFulfillmentCenterId(
         member,bean.getFulfillmentCenterId());
      if(count>0)
         return true;
  }
  return false;
 }
```

2. The next step is to create a new fulfills method in the bean class. For example, you could create a new fulfills method in the com.ibm.commerce.fulfillment.objects.FulfillmentBean.java class. The declaration for the method should appear as:

```
public boolean fulfills(Long member, String relationship)
  throws Exception {
      // Place holder for relationship information
  }
```

3. If you are adding an additional relationship to the existing relationships, the first line in your method should be to call the fulfills method from the superclass. Then, if that method returns false, then check for your new relationships, as follows:

```
public boolean fulfills(Long member, String relationship)
  throws Exception {
      if (super.fulfills().equals(false))
      {
         // Check if new relationship is met
         return true;
      }

      return false;
  }
```

4. If you are replacing the relationships from the original implementation entirely, you should not call the super.fulfills method, as follows:

```
public boolean fulfills(Long member, String relationship)
  throws Exception {
      // Check if new relationship is met
      // If it is, then return true;

     // If the relationship is not met, return false;
  }
```

5. Save your changes. Regenerate the deployed and RMIC code for the bean, as well as the corresponding access bean.

Adding access control to an existing WebSphere Commerce entity bean that is not already protected

If you are using an existing WebSphere Commerce entity bean and your application requires that the bean be protected by access control, you can add this protection.

The following list provides the high-level steps protect an existing WebSphere Commerce entity bean under the WebSphere Commerce access control system:

1. Open the *BeanName*.java class. This is the remote interface. Modify this so that it extends the `com.ibm.commerce.security.Protectable` interface.
2. If a resource is going to be grouped by an attribute other than its Java class name for the purpose of applying access control policies, the remote interface of the bean must also extend the `com.ibm.commerce.grouping.Groupable` interface.
3. Save your changes to the remote interface.
4. Open the *BeanName*Bean.java class, where *BeanName* is the name of the entity bean to which you are adding access control protection.
5. The enterprise bean class inherits default implementations for the following methods from com.ibm.commerce.base.objects.ECEntityBean:
   - `getOwner`
   - `fulfills`
   - `getGroupingAttributeValue`

   Override any methods that you need. At a minimum, you must override the `getOwner` method. Refer to "Implementing access control in enterprise beans" on page 106 for more information about these methods.
6. Save your changes. Regenerate the deployed and RMIC code for the bean, as well as the corresponding access bean.

Understanding the access control implications when a controller command is extended

According to the WebSphere Commerce programming model, you can create your own implementations of existing controller commands. In this case, you create a new implementation class and then associate the new implementation class to the existing interface by updating the command registry.

There are three potential access control-related implications of performing such an extension:

1. Effect on the getResources method.
2. Effect on the command-level access control policies
3. Effect on the resource-level access control policies

Each of these points is described in more detail in subsequent sections.

Effect on the getResources method

If you are extending an existing controller command, meaning that the existing logic of the command will be performed as well as your new customized logic, your new command will subclass from the existing command. The performExecute method of the new implementation calls the performExecute method of the superclass. If your new command does not access any new resources that require protection, you do not have to override the getResources method. However, if new protectable resources are accessed, the new command should implement its own getResources method and in this method call the getResources method from the superclass, before implementing your own getResources logic.

The results of the getResources method from the superclass should be stored in a private instance variable of type AccessVector. The results of the local getResources method should then be appended to the end of this vector. For example, the new implementation class would contain code similar to the following pseudocode:

```
private AccessVector resources = null;

public AccessVector getResources() throws ECException {
   // First, get the resources from the original implementation
   resources = super.getResources();

   // Now, append the new resources

   ///////////////////////////////////////
   //  Logic for getting new resources   //
   //  and appending to the vector.      //
  ////////////////////////////////////////

   return resources;
}
```

If you are not extending the logic of an existing controller command, but instead you are replacing the logic completely, you would not call the super.performExecute method in your new implementation. You would then not need to call the super.getResources method from within your own implementation. Instead, just implement your own getResources method, as appropriate.

Effect on the command-level access control policies

Command-level policies for controller commands consist of the “Execute” action operating on the command resource. The command resource is specified by its interface name. When you extend an existing command, the command still implements the original interface, and as such, the existing command-level policy is sufficient to maintain the previous command-level access control. Therefore, no changes are required.

Effect on the resource-level access control policies

If you have created a new implementation of an existing command and associated this implementation to the interface of that existing command, changes are not required in the resource-level access control policies.

If instead you create a new implementation class that extends an existing implementation and in this class you call the super.performExecute method and you also implement a new interface, then changes are required in resource-level access control policies.

In this latter case, if the new command implements the getResources method or inherits a non-trivial implementation of the getResources method from the base command, resource-level policy changes are required. Typically, resource-level policies consist of the command action operating on the business object resource. Since the action is simply a string representation of the command's interface, the new command generally needs to be added to the action groups of the base command. However, if the new command overrides the base implementation of getResources method by simply returning null, then no access control check will be done for this new command. Be aware that if this is not done carefully, this could potentially result in opening up the new command to malicious users.

When modifying the resource-level access control policies, the following are the high-level steps:

1. Locate the defaultAccessControlPolicies.xml file, which is found in the following directory:
   - Developer *WCDE_installdir*\Commerce\xml\policies\xml
2. Make a copy of this file. As an example, name the file *myDefaultAccessControlPolicies.xml*.
3. In this new file, you must define the new interface as a new action. For example, you would add the following:
   ```
   <Action Name="yourNewInterface"
      CommandName="yourNewInterface">
   </Action>
   ```
   where *yourNewInterface* is the name of the new interface.
4. Next, you must search through the file to locate all of the action groups to which the original action belonged and then add in your new action.
5. If the new command is operating on resources that were not previously specified by the base command, then the resource group of the corresponding access control policies must also be changed to accomodate the new resources.

6. Once you have completed your changes to the XML file, you can remove sections that you have not modified. Ensure that you keep the text before the `<Policies>` tag.
7. Load the new policy information into the database, according to the instructions contained in the *WebSphere Commerce Security Guide*.

Sample access control policies for development purposes

This section provides some very simple access control policies that can be used in the development environment, so that you can quickly test new resources. They are not designed to be used on any WebSphere Commerce production environment, as they do not provide adequate resource protection.

For information about how to load these policies, refer to the *WebSphere Commerce Security Guide*.

Sample access control policy for new views

If you create a new view, you can use the following access control policy so that you will be able to test the new view in your development environment (modify the policy for your environment and load it using the acpload command):

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE Policies SYSTEM "../dtd/accesscontrolpolicies.dtd">

<Policies>
   <Action Name="YourNewView"
        CommandName="YourNewView">
   </Action>
   <ActionGroup Name="AllSiteUsersViews"
             OwnerID="RootOrganization">
      <ActionGroupAction Name="YourNewView"/>
   </ActionGroup>
</Policies>
```

where *YourNewView* is the name of the newly created view. The preceding access control policy adds the new view to the existing AllSiteUsersViews action group. This policy allows any user to access the new view.

Sample command-level access control policy for new controller commands

Controller commands require access control policies in order to meet the requirements of the access control framework. If you create a new controller command, the name of the command’s interface is specified as a resource. The following XML snip can be modified for your new command and loaded using the acpload command:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE Policies SYSTEM "../dtd/accesscontrolpolicies.dtd">
```

```
<Policies>

   <Action Name="ExecuteCommand"
         CommandName="Execute">
   </Action>

   <ResourceCategory Name="com.yourcompany.yourpackage.commands.
         YourControllerCmdResourceCategory"
         ResourceBeanClass="com.yourcompany.yourpackage.commands.
         YourControllerCmd">
        <ResourceAction Name="ExecuteCommand"/>
   </ResourceCategory>

   <ResourceGroup Name="AllSiteUserCmdResourceGroup"
       OwnerID="RootOrganization">
      <ResourceGroupResource Name="com.yourcompany.yourpackage.commands.
         YourControllerCmdResourceCategory" />
   </ResourceGroup>

</Policies>
```

where:

- *com.yourcompany.yourpackage.commands* represents your packaging structure
- *YourControllerCmd* represents the name of your new controller command

As an example, the following XML file is used to load the access control policy for a new controller command that is created in a tutorial contained in this book.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE Policies SYSTEM "../dtd/accesscontrolpolicies.dtd">

<Policies>
   <Action Name="ExecuteCommand"
   CommandName="Execute">
   </Action>

   <ResourceCategory Name="com.ibm.commerce.sample.commands.
          MyNewControllerCmdResourceCategory"
          ResourceBeanClass="com.ibm.commerce.sample.commands.
          MyNewControllerCmd">
         <ResourceAction Name="ExecuteCommand"/>
   </ResourceCategory>

   <ResourceGroup Name="AllSiteUserCmdResourceGroup"
       OwnerID="RootOrganization">
      <ResourceGroupResource Name="com.ibm.commerce.sample.commands.
         MyNewControllerCmdResourceCategory" />
   </ResourceGroup>

</Policies>
```

Sample resource-level access control policy for a new command and enterprise bean

The following XML file is taken from a tutorial contained in this guide. It can act as a template for access control requirements when you create new entity beans. In the case of the following file, the new entity bean is called the Bonus bean, it corresponds to the XBONUS database table, and it gets used by the MyNewControllerCmd controller command. In this access control policy, only the creator of a bonus bean object can perform the MyNewControllerCmd action upon that object.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE Policies SYSTEM "../dtd/accesscontrolpolicies.dtd">

<Policies>
   <Action Name="MyNewControllerCmd"
      CommandName="com.ibm.commerce.sample.commands.MyNewControllerCmd">
   </Action>

   <ResourceCategory Name="com.ibm.commerce.extension.objects.
       BonusResourceCategory"
       ResourceBeanClass="com.ibm.commerce.extension.objects.Bonus" >

         <ResourceAction Name="MyNewControllerCmd" />
         </ResourceCategory>

   <ActionGroup Name="MyNewControllerCmdActionGroup"
      OwnerID="RootOrganization">
   <ActionGroupAction Name="MyNewControllerCmd"/>
   </ActionGroup>

   <ResourceGroup Name="BonusResourceGroup"  OwnerID="RootOrganization" >
      <ResourceGroupResource Name="com.ibm.commerce.extension.objects.
            BonusResourceCategory" />
   </ResourceGroup>
   <Policy Name="AllUsersUpdateBonusResourceGroup"
      OwnerID="FashionFlowMemberId"
      UserGroup="AllUsers"
      UserGroupOwner="RootOrganization"
      ActionGroupName="MyNewControllerCmdActionGroup"
      ResourceGroupName="BonusResourceGroup"
      RelationName="creator"
      PolicyType="groupableStandard">
   </Policy>

   <PolicyGroup Name="ManagementAndAdministrationPolicyGroup"
         OwnerID="RootOrganization">
   <!-- Define policies in this policy group -->
   <PolicyGroupPolicy  Name="AllUsersUpdateBonusResourceGroup"
            PolicyOwnerID="FashionFlowMemberId" />

   </PolicyGroup>

</Policies>
```

where *FashionFlowMemberId* is the member ID of the store in which the new resource is being used.

In the preceding access control policy, the interface name of the controller command is specified as the action, without fully-qualifying it with its package name. If your application has multiple interfaces with the same name, you must fully-qualify them with their package names when specifying them as actions in access control policies. As an example, if there was ambiguity with the interface names, the preceding access control policy would require changes, as follows (note, only changed lines are displayed and the modifications are shown in bold):

```
<Action Name="com.ibm.commerce.sample.commands.MyNewControllerCmd"
CommandName="com.ibm.commerce.sample.commands.MyNewControllerCmd">
.
.
.
<ResourceAction Name="com.ibm.commerce.sample.commands.MyNewControllerCmd" />
.
.
.
<ActionGroupAction Name="com.ibm.commerce.sample.commands.MyNewControllerCmd"/>
```

Chapter 5. Error handling and messages

Command error handling

WebSphere Commerce uses a well-defined command error handling framework that is simple to use in customized code. By design, the framework handles errors in a manner that supports multicultural stores. The following sections describe the types of exceptions that a command can throw, how the exceptions are handled, how message text is stored and used, how the exceptions are logged, and how to use the provided framework in your own commands.

Types of exceptions

A command can throw one of the following exceptions:

ECApplicationException
: This exception is thrown if the error is related to the user. For example, when a user enters an invalid parameter, an ECApplicationException is thrown. When this exception is thrown, the Web controller does not retry the command, even if it is specified as a retriable command.

ECSystemException
: This exception is thrown if a run-time exception or a WebSphere Commerce configuration error is detected. Examples of this type of exception include null-pointer exceptions and transaction rollback exceptions. When this type of exception is thrown, the Web controller retries the command if the command is retriable and the exception was caused by either a database deadlock or database rollback.

Both of the above listed exceptions are classes that extend from the ECException class, which is found in the com.ibm.commerce.exception package.

In order to throw one of these exceptions, the following information must be specified:

- Error view name
  The Web controller looks up this name in the VIEWREG table.
- ECMessage object
  This value corresponds to the message text contained within a properties file.
- Error parameters
  These name-value pairs are used to substitute information into the error message. For example, a message may contain a parameter to hold the

name of the method which threw the exception. This parameter is set when the exception is thrown, then when the error message is logged, the log file contains the actual method name.

- Error data
  These are optional attributes that can be made available to the JSP template through the error data bean.

Exception handling is tightly integrated with the logging system. When a system exception is thrown, it is automatically logged.

Error message properties files

In order to simplify the maintenance of error messages and to support multilingual stores, the text for error messages is stored in properties files. WebSphere Commerce message text is stored in the `ecServerMessages_XX_XX.properties` file, where `_XX_XX` is the locale indicator (for example, `_en_US`).

The command context returns an identifier to indicate the language used by the client. When a message is required, the Web controller determines which properties file to use based upon the language identifier.

There are two types of messages defined in the `ecServerMessagesXX_XX.properties` file: user messages and system messages. User messages are displayed to customers in their browsers. Both system and user messages are captured automatically in the message log.

When an error is thrown, one of the required parameters is a message object. For ECSystemExceptions, the message object must contain two keys, one for the system message and one for the user message. For ECApplicationExceptions, the message object contains the key for the user message (system messages are not used).

All system messages are predefined. You cannot create your own system messages. Therefore, when customized code throws an ECSystemException, it must specify a message key for one of the predefined system messages. Customized user messages can be created. New user messages must be stored in a separate properties file.

Exception handling flow

The following diagram shows the flow of information when an exception is caught. A description of each step follows.

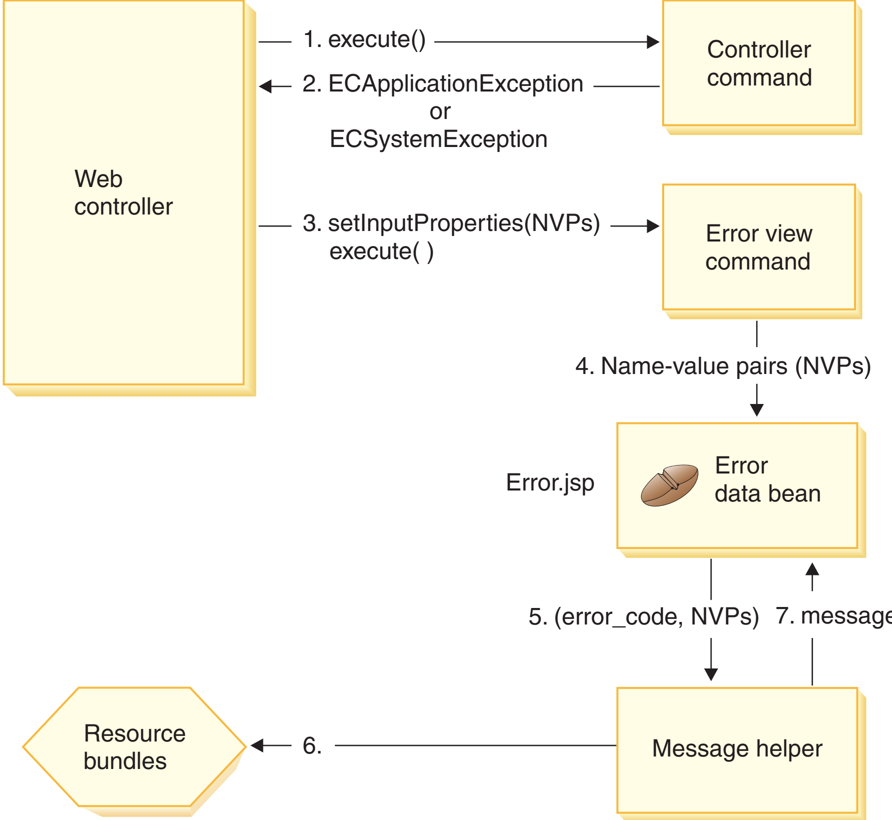


Figure 24.

1. The Web controller invokes a controller command.
2. The command throws an exception that is caught by the Web controller. This can be either an ECApplicationException, or an ECSystemException. The exception object contains the following information:
   - Error view name
   - ECMessage object
   - Error parameters
   - (optional) Error data
3. The Web controller determines the error view name from the VIEWREG table and invokes the specified error view command. When invoking the command, the Web controller composes a set of properties from the ECException object and sets it to the view command using the view command's setInputProperties method.
4. The view command invokes an error JSP template (Error.jsp in this case) and the name-value pairs are passed to the JSP template.

5. The ErrorDataBean passes the error parameters to the message helper object.
6. The message helper object gets the required message (using the message object and the error parameters) from the appropriate properties file.
7. The error data bean returns the message to the JSP template.

Exception handling in customized code

When creating new commands, it is important to include appropriate exception handling. You can take advantage of the error handling and messaging framework provided in WebSphere Commerce, by specifying the required information when catching an exception.

Writing your own exception handling logic, involves the following steps:

1. Catching the exceptions in your command that require special processing.
2. Constructing either an ECApplicationException or ECSystemException, based upon the type of exception caught.
3. If the ECApplicationException uses a new message, defining the message in a new properties file.

Catching and constructing exceptions

To illustrate the first two steps, the following code snippet shows an example of catching a system exception within a command:

```
try {
// your business logic
}
catch(FinderException e) {
     throw new ECSystemException (ECMessage._ERR_FINDER_EXCEPTION,
          className, methodName, new Object [] {e.toString()}, e);
}
```

The preceding _ERR_FINDER_EXCEPTION ECMessage object is defined as follows:

```
public static final ECMessage _ERR_FINDER_EXCEPTION =
new ECMessage (ECMessageSeverity.ERROR, ECMessageType.SYSTEM,
     ECMessageKey._ERR_FINDER_EXCEPTION);
```

The _ERR_FINDER_EXCEPTION message text is defined within the `ecServerMessages_xx_XX.properties` file (where `_xx_XX` is a locale indicator such as *_en_US*), as follows:

```
_ERR_FINDER_EXCEPTION  =
     The following Finder Exception occurred during processing: "{0}".
```

When catching a system exception, there is a predefined set of messages that can be used. These are described in the following table:

Message Object	Description
_ERR_FINDER_EXCEPTION	Thrown when an error is returned from an EJB finder method call.
_ERR_REMOTE_EXCEPTION	Thrown when an error is returned from an EJB remote method call.
_ERR_CREATE_EXCEPTION	Thrown when an error occurs creating an EJB instance.
_ERR_NAMING_EXCEPTION	Thrown when an error is returned from the name server.
_ERR_GENERIC	Thrown when an unexpected system error occurs. For example, a null pointer exception.

When catching an application exception, you can either use an existing message that is specified in the appropriate `ecServerMessages_xx_xx.properties` file, or create a new message that is stored in a new properties file. As specified previously, you *must not* modify any of the `ecServerMessages_xx_XX.properties` files.

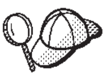
The following code snippet shows an example of catching an application exception within a command:

```
try {
// your business logic

}
// catch some new type of application exception
catch(//your new exception)
 {
     throw new ECApplicationException (MyMessages._ERR_CUSTOMER_INVALID,
          className, methodName, errorTaskName, someNVPs);
}
```

The preceding _ERR_CUSTOMER_INVALID ECMessage object is defined as follows:

```
public static final ECMessage _ERR_CUSTOMER_INVALID =
     new ECMessage (ECMessageSeverity.ERROR, ECMessageType.USER,
     MyMessagesKey._ERR_CUSTOMER_INVALID, "ecCustomerMessages");
```



When constructing new user messages, you should assign them with a type of USER, as follows:

```
ECMessageType.USER
```

The text for the _ERR_CUSTOMER_INVALID message is contained in the `ecCustomerMessages.properties` file. This file must reside in a directory that is in the class path. The text is defined as follows:

```
_ERR_CUSTOMER_INVALID = Invalid ID "{0}"
```

Creating messages

If your command throws an ECApplicationException that uses a new message, you must create this new message. Creating a new message involves the following steps:

1. Creating a new class that contains the message keys.
2. Creating a new class that contains the ECMessage objects.
3. Creating a resource bundle.

Details about each step are found in the following sections.

Creating a class for message keys

The first step in creating new user messages is to create a class that contains the new message keys. A message key is a unique indicator that is used by the logging service to locate the corresponding message text in a resource bundle. This new class should be created within your own package and stored in the WebSphereCommerceServerExtensionsLogic project.

Consider an example, called `MyNewMessages`, in which you create a new class, called `MyMessageKeys` that contains the `_ERR_CUSTOMER` and `_ERR_CUSTOMER_INVALID_ID` message keys and you put this class in the `com.mycompany.messages` package. In this case, the class definition appears as follows:

```
public class MyMessageKeys
{
    public static String _ERR_CUSTOMER="_ERR_CUSTOMER";
    public static String _ERR_CUSTOMER_INVALID_ID="_ERR_CUSTOMER_INVALID_ID";
}
```

Providing String wrappers for message keys allows the compiler to check their validity.

Creating a class for ECMessage objects

Within the same package that you created the class for your message keys, create another class that contains the ECMessage objects. The ECMessage class defines the structure of a message object. It is used to retrieve and persist locale-sensitive text messages.

The message object has the following attributes: severity, type, key, resource bundle and associated resource bundle. There are several constructor methods for this class. Refer to the “Reference” section of the WebSphere Commerce online help for complete details.

Following the MyNewMessages example, create a new class called `MyMessages` within the `com.mycompany.messages` package, as follows:

```
import com.ibm.commerce.ras.*;

public class MyMessages
    {
        static String myResourceBundle = "ecCustomerMessages";

        public static ECMessage _ERR_CUSTOMER = new ECMessage
             (ECMessageSeverity.ERROR,ECMessageType.USER,
                   MyMessageKeys._ERR_CUSTOMER, myResourceBundle);
        public static ECMessage _ERR_CUSTOMER_INVALID_ID = new ECMessage
             (ECMessageSeverity.ERROR, ECMessageType.USER,
                  MyMessageKeys._ERR_CUSTOMER_INVALID_ID,
                  myResourceBundle);

    }
```

In the preceding code snippet, the import statement is required for the creation of the ECMessage object. The object MyMessage._ERR_CUSTOMER is a user message of severity `ERROR`. The MyMessageKeys._ERR_CUSTOMER is used by the WebSphere Commerce logging service to find the message text contained in the *ecCustomerMessages* properties file.

Creating a user message resource bundle

You must create a new resource bundle, in which the message keys with corresponding message text are stored. This resource bundle can be implemented either as a Java object, or as a properties file. It is recommended that you use properties files, since they are easier to translate and maintain. Properties files are used for WebSphere Commerce messages.

To continue the MyNewMessages example, create a text file by the name of `ecCustomerMessages.properties`. If the messages are to be used by a single store servlet, place this file in the following directory:

Developer *workspace_dir*\Stores\Web Content\WEB-INF\classes*storeDir* where *storeDir* is the name of your store.

If the messages are to be used by the WebSphere Commerce Accelerator, place this file in the following directory:

Developer *workspace_dir*\CommerceAccelerator\Web Content\WEB-INF\classes

If the messages are to be used by the Administration Console, place this file in the following directory:

Developer *workspace_dir*\SiteAdministration\Web Content\WEB-INF\classes

Business If the messages are to be used by the Organization Administration Console, place this file in the following directory:

Developer *workspace_dir*\OrganizationAdministration\Web Content\WEB-INF\classes

If the messages are to be used globally by any servlet in the enterprise application, place this file in the following directory:

Developer *workspace_dir*\WebSphereCommerceServer\properties

The preceding directories are specified within the context of the development environment. Once you have completed testing in that environment, refer to Chapter 9, "Deployment details," on page 201 for information about deploying to the target WebSphere Commerce Server.

Since the properties file contains pairs of message keys and the corresponding message text, the `ecCustomerMessages.properties` file contains the following lines:

```
_ERR_CUSTOMER_MESSAGE = The customer message "{0}".
_ERR_CUSTOMER_INVALID_ID = Invalid ID "{0}".
```

Execution flow tracing

When you need to trace the flow of execution through your commerce application, you should use the WebSphere Application Server JRas facility. This is a message logging and diagnostic trace API that can be used by applications.

For information about how to use this facility in your customized code, refer to the WebSphere Application Server InfoCenter.

For information about configuring component tracing in the development environment, refer to Appendix A, "Configuring WebSphere Commerce component tracing in the WebSphere Commerce development environment," on page 379.

JSP template error handling

Error handling for JSP templates can be performed in various ways:

- Error handling from within the page
  For JSP files that require more intricate error handling and recovery, the file can be written to directly handle errors from the data bean. The JSP file can either catch exceptions thrown by the data bean or it can check for error codes set within each data bean, depending on how the data bean was activated. The JSP file can then take an appropriate recovery action based on the error received. Note that a JSP file can use any combination of the following error handling scopes.
- Error JSP at the page level
  A JSP file can also specify its own default error JSP template from an exception occurring within itself through the JSP error tag. This enables a JSP program to specify its own handling of an error. A JSP file which does not specify a JSP error tag will have an error fall through to the application

level JSP error template. In the page level error JSP, it must call the JSP helper class (com.ibm.server.JSPHelper) to rollback the current transaction.

- Error JSP at the application level
  An application under WebSphere can specify a default error JSP template when an exception from within any of its servlets or JSP files occur. The application level error JSP template can be used as a mall level or store level (for a single store model) error handler. In the application level error JSP template, a call must be made to the servlet helper class to roll back the current transaction. This is because the Web controller will not be in the execution path to roll back the transaction. Whenever possible, you should rely on the preceding two types of JSP error handling. Use the application level error handling strategy only when required.

Chapter 6. Command implementation

This section provides information about how to write new controller, task, and data bean commands. It also describes how to extend existing controller, task, and data bean commands.

Note: Business This chapter does not describe business policy commands. For information about business policy commands, refer to Chapter 7, "Trading agreements and business policies (Business Edition)," on page 157.

New commands - introduction

The WebSphere Commerce programming model defines four types of commands: controller, task, view and data bean commands. When creating new business logic for your e-commerce application, it is expected that you may need to create new controller, task and data bean commands. You should not need to create new view commands. More information on view commands is found later in this section.

New commands must implement their corresponding interface (which in turn should extend from an existing interface). To simplify command writing, WebSphere Commerce includes an abstract implementation class for each type of command. New commands should extend from these classes.

As an overview, the following table provides information about which implementation class a new command should extend from, and which interface it should implement:

Command type	Example command name	Extends from	Implements example interface
Controller command	MyControllerCmdImpl	com.ibm.commerce. command. ControllerCommandImpl	MyControllerCmd
Task command	MyTaskCmdImpl	com.ibm.commerce. command. TaskCommandImpl	MyTaskCmd
Data bean command	MyDataBeanCmdImpl	com.ibm.commerce. command. DataBeanCommandImpl	MyDataBean

© Copyright IBM Corp. 2000, 2003

Note: Any spaces in names of implementation classes are for presentation purposes only.

The following diagram illustrates the relationship between the interface and implementation class of a new controller command with the existing abstract implementation class and interface. The abstract class and interface are both found in the com.ibm.commerce.command package.

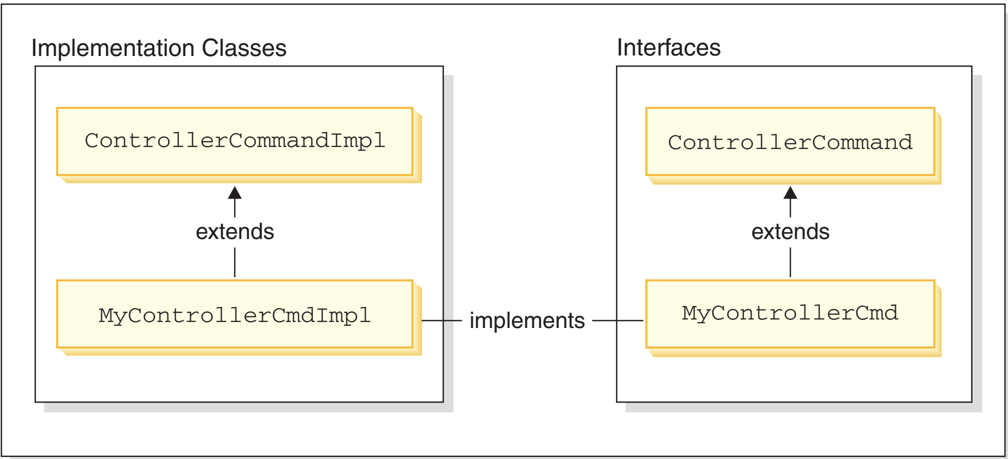


Figure 25.

The following diagram illustrates the relationship between the interface and implementation class of a new task command with the existing abstract implementation class and interface. The abstract class and interface are both found in the com.ibm.commerce.command package.

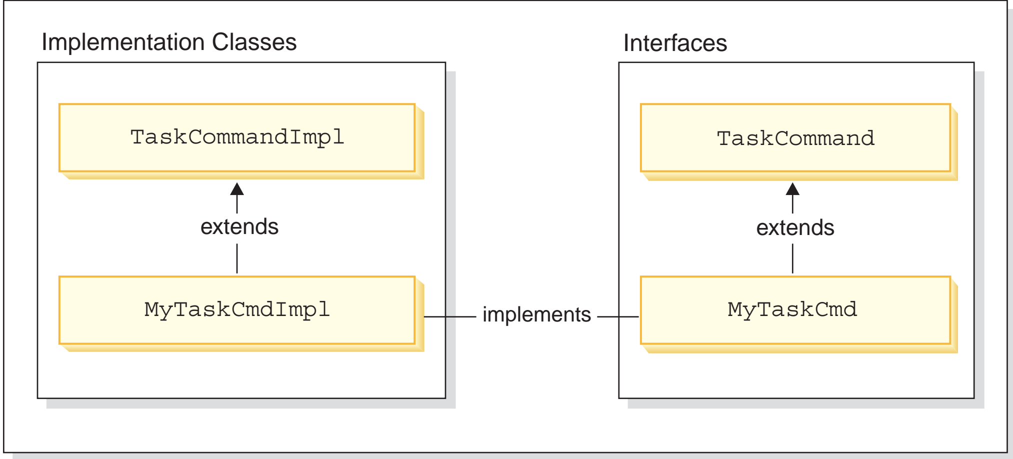


Figure 26.

The following diagram illustrates the relationship between the interface and implementation class of a new data bean command with the existing abstract implementation class and interface. The abstract class and interface are both found in the com.ibm.commerce.command package.

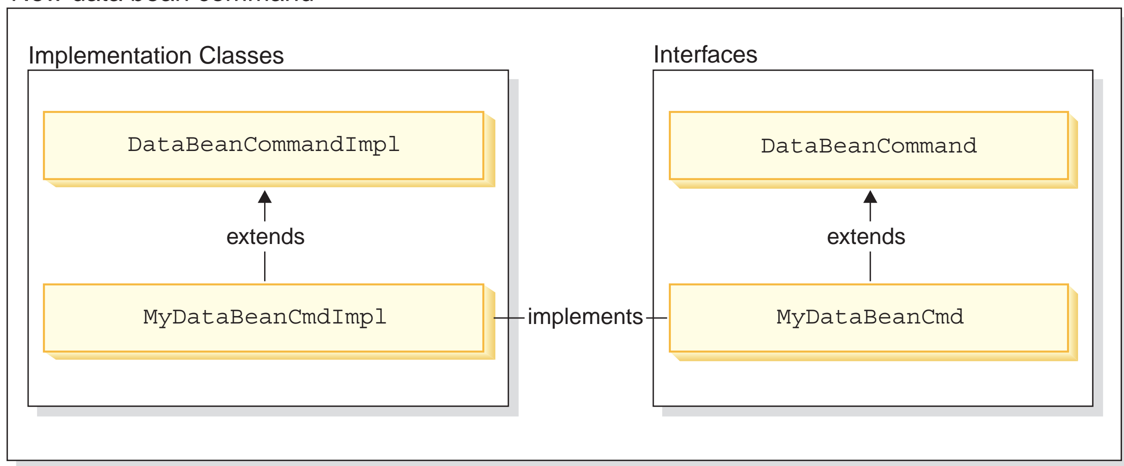


Figure 27.

A view command has two main functions: to format a response and to send the response to the client. A number of generic view commands are provided that send the response to clients, using different protocols. The formatting function is typically handled by the view command invoking a JSP template.

For example, the RedirectViewCommand view command directs the client to a URL to get the response (the response is then formatted by a specified JSP template). The ForwardViewCommand view command forwards the request to the JSP template for formatting and the page is displayed to the client.

Using this view command model, you can create new *views* (the response to the client) by creating new JSP templates. The JSP template should, however, be invoked by one of the existing view commands.

Packaging customized code

When creating customized code, you must follow a particular code organization structure. In general, customized code is maintained in projects in the WebSphere Commerce workspace that are predefined for your customized code. Two predefined projects are provided; the WebSphereCommerceServerExtensionsLogic project and the WebSphereCommerceServerExtensionsData project. The first project is for command and data bean logic, and the second is for any enterprise beans that you create.

When creating new commands, you must place them in a package named appropriately for your business requirements. That is, if the commands apply to a particular store, package them in a package that is unique to the store. If they apply to more than one store, package them accordingly. For example, you might have the following packages:

- `com.bigbusiness.storeA.commands`
- `com.bigbusiness.storeB.commands`
- `com.bigbusiness.commands`

The preceding packaging structure allows for differentiation between business logic at a store level.

When creating new data beans, they must be kept in a package that is separate from command logic, however, this package should be kept within the project (WebSphereCommerceServerExtensionsLogic) that stores the command packages. From the preceding example, you would then place the `com.bigbusiness.databeans` package within the WebSphereCommerceServerExtensionsLogic project.

When creating new entity beans, they should be stored in the WebSphereCommerceServerExtensionsData project. Therefore, you might have the WebSphereCommerceServerExtensionsData project that contains the `com.bigbusiness.objects` package.

This packaging strategy is required for code deployment purposes.

Command context

Commands can obtain information from the Web controller using the command context. Examples of information available include the user's ID, the user object, the language identifier, and the store identifier.

When writing a command, you have access to the command context by calling the getCommandContext() method of the command's superclass. The command context is set to the controller command when the command is invoked by the Web controller. A controller command should propagate the command context to any task or controller commands that are invoked during processing. A command can get the following key information from the command context:

getUserId() and getUser()
: Gets the current user ID or user object. The user ID for the current session is saved in a session context. The session context can be persisted in one of two ways: using the WebSphere Commerce cookie or a WebSphere Application Server persistent session object. The command context hides the complexity of session management from a command.

getStoreId(), getStore(), and getStore(storeId)
: Gets the store associated with the current request. The Web controller returns the store ID in the URL. If the store ID is not specified in the URL, it can be retrieved from the session object that is saved from the previous request. The WebSphere Commerce run-time environment maintains a set of objects that are frequently accessed. For example, it maintains the set of store objects. A command should always get the store object from the command context to take advantage of the object cache in the Web controller. You can get the current store by calling the getStore() method or get a specific store object by calling the getStore(storeId) method from the command context.

getLanguageId()
: Returns the language ID that should be used for the current request. The Web controller implements a Globalization Framework. The concept behind this framework is to determine a language that is preferred by the user and supported by the store. If the URL contains a language ID, the Web controller determines if this language is supported by the store, if so, this is the language ID that gets returned by getLanguageId() method. If no language ID were included in the URL, then the Web controller goes through a decision tree to determine if there is a language ID (that is supported by the store) in the current session object, or in the user's registered preferences, or lastly it will return the default language ID for the store.

getCurrency()
Returns the currency to be used for the current request. Since currency is part of the Globalization Framework, logic behind this method is similar to that of the getLanguageId() method.

getCurrentTradingAgreements() and getTradingAgreement(tradingAgreementId)
Returns the set of trading agreements that are used for the current session. This set may be all of the trading agreements to which the user is entitled, or it can be a subset that was defined by the ContractSetInSession command. A command should always get the trading agreement object from the command context to take advantage of the object cache in the Web controller. You can get the current trading agreement by calling the getCurrentTradingAgreements() method or get a specific trading agreement object by calling the getTradingAgreement(tradingAgreementId) method from the command context.

The command context should be used as a read-only object. You should not call its setter methods. The setter methods are reserved for use by the WebSphere Commerce run-time environment and they may be deprecated in future releases.

For complete details on the command context API (application programming interface), refer to the "Reference" topic in the WebSphere Commerce Production and Development online help.

Temporary changes to contextual information for URL commands

It is possible to override some of the command context information and execute URL commands within the context of another store, or on behalf of another user. URL commands have the following URL input parameters that allow for this temporary switch in command context:

- forStoreId
- forUser
- forUserId

The forStoreId URL input parameter allows you to specify the store ID to be used for this particular URL request. This, in effect, temporarily changes the storeId value in the command context to that of the specified store, but this change is only valid for the duration of the URL command.

Both the forUser and forUserId URL input parameters allow you to specify that the command be executed for the specified user, even though the user that is currently logged in may be different. This is particularly useful when a

customer service representative needs to assist a customer. For example, the customer service representative is able to update a customer's address information on behalf of that customer, by specifying the customer's user name or user ID by using the URL input parameters. This change in user information is only valid for the duration of the URL request for which it was specified.

New controller commands

As previously stated, a new controller command should extend from the abstract controller command class (com.ibm.commerce.command.ControllerCommandImpl). When writing a new controller command, you should override the following methods from the abstract class:

- isGeneric()
- isRetriable()
- setRequestProperties(com.ibm.commerce.datatype.TypedProperty reqParms)
- validateParameters()
- getResources()
- performExecute()

More information on each of the preceding methods is found in the following sections.

isGeneric method

In the standard WebSphere Commerce implementation there are multiple types of users. These include generic, guest, and registered users. Within the grouping of registered users there are customers and administrators.

The generic user has a common user ID that is used across the entire system. This common user ID supports general browsing on the site in a manner that minimizes system resource usage. It is more efficient to use this common user ID for general browsing, since the Web controller does not need to retrieve a user object for commands that can be invoked by the generic user.

The isGeneric method returns a boolean value which specifies whether or not the command can be invoked by the generic user. The isGeneric method of a controller command's superclass sets the value to `false` (meaning that the invoker must be either a registered user or a guest user). If your new controller command can be invoked by generic users, override this method to return `true`.

You should override this method to return `true` if your new command does not fetch or create resources associated with a user. An example of a command that can be invoked by a generic user is the ProductDisplay

command. It is sensible to allow any user to be able to view products. An example of a command for which a user must be either a guest or registered user (and hence, isGeneric returns `false`) is the OrderItemAdd command.

When isGeneric returns a value of `true`, the Web controller does not create a new user object for the current session. As such, commands that can be invoked by the generic user run faster, since the Web controller does not need to retrieve a user object.

The syntax for using this method to enable generic users to invoke a command is as follows:

```
public boolean isGeneric()
{
     return true;
}
```

isRetriable method

The isRetriable method returns a boolean value which specifies whether or not the command can be retried on a transaction rollback exception. The isRetriable method of the new controller command's superclass returns a value of `false`. You should override this method and return a value of `true`, if your command can be retried on a transaction rollback exception.

An example of a command that should not be retried in the case of a transaction exception is the OrderProcess command. This command invokes the third party payment authorization process. It cannot be retried, since that authorization cannot be reversed. An example of a command that can be retried is the ProductDisplay command.

The syntax for enabling the command to be retried in the case of a transaction rollback exception is as follows:

```
public boolean isRetriable()
{
   return true;
}
```

setRequestProperties method

The setRequestProperties method is invoked by the Web controller to pass all input properties to the controller command. The controller command must parse the input properties and set each individual property explicitly within this method. This explicit setting of properties by the controller command itself promotes the concept of type safe properties.

The syntax for using this method is as follows:

```
public void setRequestProperties(
   com.ibm.commerce.datatype.TypedProperty reqParms)
{
```

```
     // parse the input properties and explicitly set each parameter

}
```

validateParameters method

The validateParameters method is used to do initial parameter checking and any necessary resolution of parameters. For example, it could be used to resolve `orderId=*`. This method is called before both the getResources and performExecute methods. Refer to "Access control interactions" on page 101 for more details about this sequence.

getResources method

This method is used to implement resource-level access control. It returns a vector of resource-action pairs upon which the command intends to act. If nothing is returned, no resource-level access control is performed. For more information about access control, refer to Chapter 4, "Access control," on page 89.

performExecute method

The performExecute method contains the business logic for your command. It should invoke the performExecute method of the command's superclass before any new business logic is executed. At the end, it must return a view name.

The following shows example syntax for the performExecute method in a new controller command. In this case, the response uses a redirect view command, it could, however, use a forward view command or direct view command:

```
public void performExecute() throws ECException
{
     super.performExecute();

/////////////////////////////////////////////////
//  your business logic                        //
/////////////////////////////////////////////////

// Create a new TypedProperty for response properties.
TypedProperty rspProp = new TypedProperty();

// set response properties
rspProp.put(ECConstants.EC_VIEWTASKNAME, "MyView");
/////////////////////////////////////////////////////
//  The following line is optional.  The VIEWREG   //
//  table can specify the redirect URL.            //
/////////////////////////////////////////////////////

rspProp.put(ECConstants.EC_REDIRECTURL, MyURL);

////////////////////////////////////////////////////////////////////////////
//   If you are using a forward view, you can set the                     //
//   response properties as follows:                                      //
```

```
//   TypedProperty rspProp = new TypedProperty();                        //
//   rspProp.put(ECConstants.EC_VIEWTASKNAME, "MyView");                 //
//   rspProp.put(ECConstants.EC_DOCPATHNAME, "MyJSP.jsp");               //
//                                                                       //
//   Again, it is optional to explicitly set the name of the JSP template.//
//   The VIEWREG table can specify the JSP template.                     //
/////////////////////////////////////////////////////////////////////////

setResponseProperties(rspProp);
}
```

If you specify the redirect URL within the performExecute method and an entry exists in the VIEWREG table, the value specified in the code takes precedence over the value in the VIEWREG table. The same order of precedence holds true for specification of a JSP template within code.

Long-running controller commands

If a controller command takes a long time to execute, you can split the command into two commands. The first command, which is executed as the result of a URL request, simply adds the second command to the Scheduler, so that it runs as a background job. This is illustrated in the following diagram:

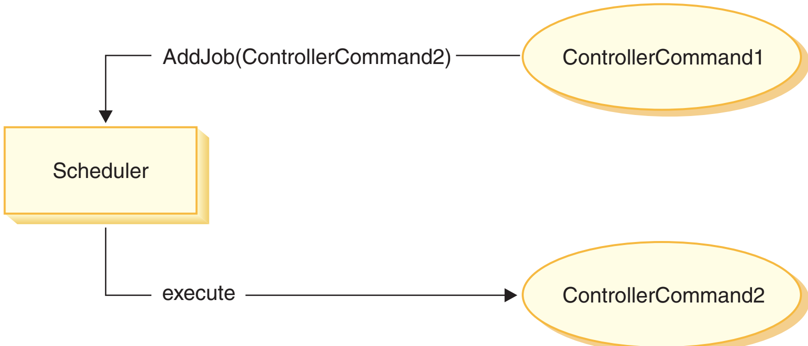


Figure 28.

The flow shown in the preceding diagram is as follows:

1. ControllerCommand1 is executed as a result of a URL request.
2. ControllerCommand1 adds a job to the Scheduler. The job is ControllerCommand2. ControllerCommand1 returns a view, immediately after adding the job to the Scheduler.
3. The Scheduler executes ControllerCommand2 as a background job.

In this scenario, the client typically polls the result from ControllerCommand2. ControllerCommand2 should write the job state to the database.

Formatting of input properties to view commands

When a controller command completes, it returns the name of a view that should be executed. This view may require that several input properties get passed to it. There can be three sources for these input parameters, as described in the following list:

- Default properties that are stored in the PROPERTIES column of the CMDREG table
- Default properties from the PROPERTIES column of the VIEWREG table
- Input properties from the URL

For more information about how these properties are merged and set in the attributes for the JSP template, refer to “Setting JSP attributes - overview” on page 43. This section describes how the input properties to a view command may be formatted.

For redirect view commands, two topics are examined:

- Flattening a query string to support URL redirection
- Dealing with a limit on the length of the redirect URL

For forward view commands, the topic of enumeration of input parameters and setting them as attributes in the HttpServletRequestObject is examined.

Flattening input parameters into a query string for HttpRedirectView

All input parameters that are passed to a redirect view command are flattened into a query string for URL redirection. For example, suppose that the input to the redirect view command contains the following properties:

```
URL = "MyView?p1=v1&p2=v2";
ip1 = "iv1";   // input to orginal controller command
ip2 = "iv2" ;  // input to original controller command
op1 = "ov1";
op2 = "ov2";
```

Based upon the preceding input parameters, the final URL is

```
MyView?p1=v1&p2=v2&ip1=iv1&ip2=iv2&op1=ov1&op2=ov2
```

Note that if the command is to use SSL, then the parameters are encrypted and the final URL appears as

```
MyView?krypto=encrypted_value_of"p1=v1&p2=v2&ip1=iv1&ip2=iv2&op1=ov1&op2=ov2"
```

Handling a limited length redirect URL

By default, all input parameters to the controller command are propagated to the redirect view command. If there is a limit on the number of characters in the redirect URL, this may cause a problem. An example of when the length may be limited is if the client is using the Internet Explorer browser. For this browser, the URL cannot exceed 2083 bytes. If the URL does exceed this limit,

the URL gets truncated. As such, you can encounter a problem if there are a large number of input parameters, or if you are using encryption, because an encrypted string is typically two to three times longer than a string that is not encrypted.

There are two approaches for handling a limited length redirect URL:
1. Override the getViewInputProperties method in the controller command to return only the sets of parameters that are required to be passed to the redirect view command.
2. Use a specified special character in the URL parameters to indicate which parameters can be removed from the input parameter string.

To demonstrate each of the preceding approaches, consider the following set of input parameters to the controller command:

```
URL="MyView";
// All of the following are inputs to the original controller command.
ip1="ipv1";
ip2="ipv2";
ip3="ipv3";
iq1="iqv1";
iq2="iqv2";
ir1="ipr1";
ir2="ipr2";
is="isv";
```

If you are overriding the getViewInputProperties method, the new method can be written so that only the following parameters are passed to the view command:

```
ir2="ipr2";
is="isv";
```

Using the second approach, the view command can be invoked using special parameters to indicate that certain input parameters should be removed. For example, you can achieve the same result by specifying the following as the URL parameter:

```
URL="MyView?ip*=&iq*=&ir1="
```

This URL parameter instructs the WebSphere Commerce run-time framework of the following:
- The `ip*=` specification means that all parameters whose names start with `ip` should be removed.
- The `iq*=` specification means that all parameters whose names start with `iq` should be removed.
- The `ir1=` specification means that the `ir1` parameter should be removed.

Setting attributes in the HttpServletRequest object for HttpForwardView

The default HttpForwardViewCommandImpl enumerates all of the parameters passed to the command and sets them as attributes in the HttpServletRequest object.

For example, suppose that the requestProperties object passed to the forward view command contains the following parameters:

```
p1="pv1";
p2="pv2";
p3=pv3;   // pv3 is an object
```

Then the following attributes are passed to the JSP template using the request.setAttribute() method.

```
request.setAttribute("p1", "pv1");
request.setAttribute("p2", "pv2");
request.setAttribute("p1", pv1);
request.setAttribute("RequestProperties", requestProperties);
request.setAttribute("CommandContext", commandContext);
```

where `requestProperties` is the TypedProperty object that is passed to the command, `commandContext` is the command context object that is passed to the command, and `p1`, `p2`, and `p3` are parameters defined in the requestProperties object.

Database commits and rollbacks for controller commands

Throughout the execution of a controller command, data is often created or updated. In many cases, the database must be updated with the new information at the end of the transaction. The transaction is managed by the Web controller.

The Web controller marks the beginning of the transaction before calling the controller command. When the execution of the controller command is complete, the controller command returns a view name to the Web controller. The Web controller is responsible for marking the end of the transaction. The actual point at which the transaction ends (before or after invoking the view) is dependent upon the type of view used.

There are three types of view commands:
- Forward view command
- Redirect view command
- Direct view command

The Web controller determines the view command to be used for the view, by looking up the view name in the VIEWREG table.

If the entry in the VIEWREG table specifies the use of the ForwardViewCommand, then the Web controller forwards the results of the controller command to the corresponding ForwardViewCommand implementation class (also specified in the VIEWREG). The view command executes within the context of the current transaction. In this case, the database commit or rollback does not occur until the view command completes.

If the entry in the VIEWREG table specifies the use of the RedirectViewCommand, then the Web controller forwards the results of the controller command to the corresponding RedirectViewCommand implementation class. The view command then operates outside of the scope of the current transaction and the database commit or rollback occurs before the redirected view command is called.

If the entry in the VIEWREG table specifies the use of the DirectViewCommand, then the Web controller forwards the results of the controller command to the corresponding DirectViewCommand implementation class. The view command executes within the context of the current transaction. In this case, the database commit or rollback does not occur until the view command completes. (Note that ForwardViewCommand and DirectViewCommand are similar. The ForwardViewCommand forwards the results to a JSP template. In contrast, the DirectViewCommand receives the results as input stream and passes them on as an output stream. It uses either the getRawDocument method that treats the data as bytes, or the getTextDocument that treats the data as text.)

In the cases where the view command executes under the same transaction scope as the controller command, an error in the view command causes a rollback of the entire transaction. This may or may not be the desired outcome, depending upon your business logic.

Example of transaction scope with a controller command

To illustrate the differences in transaction scope for a controller command, depending upon the type of view command used, consider the following examples.

Case 1: Executing the view within the scope of the controller command transaction

Suppose that you have created a new controller command called `YourControllerCmdA`. The command's performExecute method would then include the following:

```
.
.
// Create a new TypedProperty object for output.
TypedProperty rspProp = new TypedProperty();
```

```
//////////////////////
// Business logic   //
//////////////////////

// Return the view
rspProp.put(ECConstants.EC_VIEWTASKNAME, "YourView");
SetResponseProperties(rspProp);
```

In the preceding code snippet, the controller command returns “YourView” as the view. YourView is registered in the VIEWREG table. The following is an example insert statement to register YourView.

```
insert into VIEWREG (ViewName, DeviceFmt_id, storeEnt_id, interfacename,
classname, properties)

values ('YourView',  -1, XX,'com.ibm.commerce.command.ForwardViewCommand',
'com.ibm.commerce.command.HttpForwardViewCommandImpl','docname=YourView.jsp');
```

where XX is the store identifier. Since the view uses the `com.ibm.commerce.command.HttpForwardViewCommandImpl` implementation class, the Web controller uses the generic forward view command.

Based upon the preceding command registration, the Web controller launches the `YourView.jsp` file within the scope of the controller command transaction. If an error occurs in `YourView.jsp`, the transaction fails and a database rollback occurs. As a result, the entire controller command fails.

Case 2: Executing the view outside of the scope of the controller command transaction

Suppose that you would prefer to have information committed to the database, even in the case when an error may occur in the view. In order to have the view execute outside the scope of the controller command’s transaction, the view must be executed as a redirect.

To execute the view as a redirect, the performExecute method of the controller command returns the view in the following manner:

```
.
.
// Create a new TypedProperty object for output.
TypedProperty rspProp = new TypedProperty();

//////////////////////
// Business logic   //
//////////////////////

// Return the view
rspProp.put(ECConstants.EC_VIEWTASKNAME, EC_GENERIC_REDIRECTVIEW);
rspProp.put(EC_Constants.EC_REDIRECTURL, "YourView2");
```

The following example SQL statement supports the redirect strategy:

```
insert into VIEWREG (ViewName, DeviceFmt_id, storeEnt_id, interfacename,
classname, properties)

values ('YourView2',  -1, XX,'com.ibm.commerce.command.ForwardViewCommand',
'com.ibm.commerce.command.HttpForwardViewCommandImpl','docname=YourView2.jsp');
```

where XX is the store identifier.

Since the command passes the EC_GENERIC_REDIRECTVIEW value as a response property parameter, the Web controller uses the generic redirect view command. The generic redirect view is registered in the VIEWREG table with the following information:

- ViewName = RedirectView
- DeviceFmt_Id = -1
- InterfaceName = com.ibm.commerce.command.RedirectViewCommand
- ClassName = com.ibm.commerce.command.HttpRedirectViewCommandImpl

The Web controller invokes the generic redirect view command, which takes the redirect URL as an input property. The response is the redirected to the redirect URL. After the redirect occurs, the YourView2 is invoked. This is then implemented as a generic forward view.

New task commands

A new task command should extend from the abstract task command class (com.ibm.commerce.command.TaskCommandImpl) and implement an interface that extends the com.ibm.commerce.command.TaskCommand interface. As shown in the diagram on page 135, the new task command should be defined as follows:

```
public class MyTaskCmdImpl extends com.ibm.commerce.command.TaskCommandImpl
     implements MyTaskCmd {

}
```

All the input and output properties for the task command must be defined in the command interface, for example MyTaskCmd. The caller programs to the task command interface, rather than the task command implementation class. This enables you to have multiple implementations of the task command (one for each store), without the caller being concerned about which implementation class to call.

All the methods defined in the interface must be implemented in the implementation class. Since the command context should be set by the caller (a controller command), the task command does not need to set the command

context. The task command can, however, obtain information from the Web controller by using the command context.

In addition to implementing the methods defined in the task command interface, you should override the performExecute method from the com.ibm.commerce.command.TaskCommandImpl class.

The performExecute method contains the business logic for the particular unit of work that the task command performs. It should invoke the performExecute method of the task command's superclass, before performing any business logic. The following code snippet shows an example performExecute method for a task command.

```
public void performExecute() throws ECException
{
     super.performExecute();

     // Include your business logic here.

     // Set output properties so the controller command
     // can retrieve the result from this task command.
}
```

The run-time framework calls the getResources method of the controller command to determine which protectable resources the command will access. It may be the case that a task command is executed during the scope of a controller command and it attempts to access resources that were not returned by the getResources method of the controller command. If this is the case, the task command itself can implement a getResources method to ensure that access control is provided for protectable resources.

Note that by default, getResources returns `null` for a task command and resource-level access control checking is not performed. Therefore, you must override this if the task command accesses protectable resources.

Customization of existing commands

This section describes the various ways in which you can customize existing controller, task and data bean commands.

Customizing existing controller commands

A controller command encapsulates the business logic for a business process. Individual units of work within the business process may be performed by task commands. As such, there are several ways in which a controller command can be customized, some of which involve customizing task commands.

When customizing a controller command, you can accomplish the following:

- Add additional processing and logic to an existing controller command. This can be added before existing business logic, after existing logic, or both before and after.
- Replace one or more task commands. This allows you to modify how a particular step in the business process is performed.
- Replace the view called by the controller command.

The following sections provide details on how to make the preceding modifications.

Adding new business logic to a controller command

Suppose there is an existing WebSphere Commerce controller command, called `ExistingControllerCmd`. Following the WebSphere Commerce naming conventions, this controller command would have an interface class named `ExistingControllerCmd` and an implementation class named `ExistingControllerCmdImpl`. Now assume that a business requirement arises and you must add new business logic to this existing command. One portion of the logic must be executed before the existing command logic and another portion must be executed after the existing command logic.

The first step in adding the new business logic is to create a new implementation class that extends the original implementation class. In this example, you would create a new `ModifiedControllerCmdImpl` class that extends the `ExistingControllerCmdImpl` class. The new implementation class should implement the original interface (`ExistingControllerCmd`).

In the new implementation class you must create a new `performExecute` method to override the `performExecute` of the existing command. Within the new `performExecute` method, there are two ways in which you can insert your new business logic: you can either include the code directly in the controller command, or you can create a new task command to perform the new business logic. If you create a new task command then you must instantiate the new task command object from within the controller command.

The following code snippet demonstrates how to add new business logic to the beginning and end of an existing controller command by including the logic directly in the controller command:

```
public class ModifiedControllerCmdImpl extends ExistingControllerCmdImpl
   implements ExistingControllerCmd
   {
      public void performExecute ()
         throws com.ibm.commerce.exception.ECException
         {

            /* Insert new business logic that must be
               executed before the original command.
            */
```

```
            // Execute the original command logic.
            super.performExecute();

            /* Insert new business logic that must be
               executed after the original command.
            */
          }
    }
```

The following code snippet demonstrates how to add new business logic to the beginning of an existing controller command by instantiating a new task command from within the controller command. In addition, you would also create the new task command interface and implementation class and register the task command in the command registry.

```
// Import the package with the CommandFactory
import com.ibm.commerce.command.*;

public class ModifiedControllerCmdImpl extends ExistingControllerCmdImpl
   implements ExistingControllerCmd
   {

      public void performExecute ()
         throws com.ibm.commerce.exception.ECException
         {
            MyNewTaskCmd cmd = null;
            cmd = (MyNewTaskCmd) CommandFactory.createCommand(
               "com.mycompany.mycommands.MyNewTaskCommand",
               getStoreId());

            /*
            Set task command's input parameters, call its
            execute method and retrieve output
            parameters, as required.
            */

            super.performExecute();
         }
   }
```

Regardless of whether you include the new business logic in the controller command, or create a task command to perform the logic, you must also update the CMDREG table in the WebSphere Commerce command registry to associate the new controller command implementation class with the existing controller command interface. The following SQL statement shows an example update:

```
update CMDREG
set CLASSNAME='ModifiedControllerCmdImpl'
where INTERFACENAME='ExistingControllerCmd'
```

Replacing task commands called by a controller command

A controller command often calls several task commands that perform individual tasks. Collectively, these tasks make up the business process represented by the controller command. You may need to change the way in which a particular step in the process is performed, rather than adding new business logic to the beginning or end of the controller command. In this case, you must replace the implementation of the task command that you wish to override, with the implementation of a new task command that performs the task in your desired manner.

As a result of the design of the WebSphere Commerce programming model, you do not need to create a new controller command implementation class to replace the task command. The controller command instantiates the task command by calling the command factory's `createCommand` method. The command factory uses the task command's interface name and then determines the correct implementation class, based upon the command registry. As such, to replace the task command that gets instantiated, you must create a new task command implementation class and then update the command registry so that the original task command interface name is associated with the new task command implementation class. Refer to "Customizing existing task commands" on page 154 for more information.

Replacing the view called by a controller command

To replace the view that is called by a controller command, you create a new implementation class for the controller command. For example, create a new `ModifiedControllerCmdImpl` that extends `ExistingControllerCmdImpl` and implements the `ExistingControllerCmd` interface.

Within the `ModifiedControllerCmdImpl` class, override the `performExecute` method. In the new `performExecute` method, call `super.performExecute` to ensure that all command processing occurs. After the command logic is executed, you can use the response properties to override the view called. The following code snippet displays how to override the view when the view is executed as a redirect:

```
// Import the packages containing TypedProperty, and ECConstants.
import com.ibm.commerce.datatype.*;
import com.ibm.commerce.server.*;

public class ModifiedControllerCmdImplImpl extends ExistingControllerCmdImpl
   implements ExistingControllerCmd
   {
      public void performExecute ()
         throws com.ibm.commerce.exception.ECException
         {

            // Execute the original command logic.
            super.performExecute();
```

```
            // Create a new TypedProperty for response properties.
            TypedProperty rspProp = new TypedProperty();

           // set response properties
           rspProp.put(ECConstants.EC_VIEWTASKNAME, "MyView");
           /////////////////////////////////////////////////////
          //  The following line is optional.  The VIEWREG   //
          //  table can specify the redirect URL.            //
          /////////////////////////////////////////////////////

          rspProp.put(ECConstants.EC_REDIRECTURL, MyURL);

          setResponseProperties(rspProp);

         }
   }
```

The following code snippet displays how to override the view when the view is executed as a forward view:

```
// Import the packages containing TypedProperty, and ECConstants.
import com.ibm.commerce.datatype.*;
import com.ibm.commerce.server.*;

public class ModifiedControllerCmdImplImpl extends ExistingControllerCmdImpl
   implements ExistingControllerCmd
   {
      public void performExecute ()
         throws com.ibm.commerce.exception.ECException
         {

            // Execute the original command logic.
            super.performExecute();

            // Create a new TypedProperty for response properties.
            TypedProperty rspProp = new TypedProperty();

           // set response properties
           rspProp.put(ECConstants.EC_VIEWTASKNAME, "MyView");

           /////////////////////////////////////////////////
          //  It is optional to explicitly set the name   //
          //  of the JSP template. The VIEWREG table can  //
          //  specify the JSP template.                   //
          /////////////////////////////////////////////////

          rspProp.put(ECConstants.EC_DOCPATHNAME, "MyJSP.jsp");

          setResponseProperties(rspProp);

         }
    }
```

To determine which view is used by an existing controller command, refer to the "Reference" topic of the WebSphere Commerce Production and Development online help.

Customizing existing task commands

There are two standard ways to modify existing WebSphere Commerce task commands. With these methods of modification, you can accomplish the following:

- Add additional processing and logic to an existing task command. This can be added before existing business logic, after existing logic, or both before and after.
- Completely replace the existing business logic with your own business logic.

To accomplish the above modifications, you actually create a new task command implementation class. More detail is provided in the following sections.

Adding new business logic to a task command

Suppose there is an existing WebSphere Commerce task command, called `ExistingTaskCmd`. Following the WebSphere Commerce naming conventions, this task command would have an interface class named `ExistingTaskCmd` and an implementation class named `ExistingTaskCmdImpl`. Now assume that a business requirement arises and you must add new business logic to this existing command. One portion of the logic must be executed before the existing command logic and another portion must be executed after the existing command logic.

The first step in adding the new business logic is to create a new implementation class that extends the original implementation class. In this example, you would create a new `ModifiedTaskCmdImpl` class that extends the `ExistingTaskCmdImpl` class. The new implementation class should implement the original interface (`ExistingTaskCmd`).

Within the new command, you override the existing `performExecute` method and include the new logic before and after calling the `super.performExecute` method.

The following code snippet demonstrates how to add new business logic to an existing task command:

```
public class ModifiedTaskCmdImpl extends ExistingTaskCmdImpl
     implements ExistingTaskCmd {


         /* Insert new business logic that must be
            executed before the original command.
         */
```

```
        // Execute the original command logic.
        super.performExecute();

        /* Insert new business logic that must be
           executed after the original command.
        */
}
```

You must also update the CMDREG table to associate the new implementation class with the existing interface. The following SQL statement shows an example update:

```
update CMDREG
set CLASSNAME='ModifiedTaskCmdImpl'
where INTERFACENAME='ExistingTaskCmd'
```

Replacing business logic of an existing task command

To replace the business logic of an existing task command, you must create a new implementation class for the task command. This new implementation class must extend from the existing task command but it should not implement the existing interface. Additionally, in the new implementation class, do not call the `performExecute` method of the superclass.

While extending from the exact command that you are replacing may seem counterintuitive, the reason for taking this approach is related to support for future versions of WebSphere Commerce. This approach shields your code from changes that may be made to command interfaces in future versions of WebSphere Commerce.

As an example, suppose you wanted to replace the business logic of the `OrderNotifyCmdImpl` task command. In this case, you would create a new task command called `CustomizedOrderNotifyCmdImpl`. This command extends `OrderNotifyCmdImpl`. In the new `CustomizedOrderNotifyCmdImpl`, you create the new business logic, but do not call the `performExecute` method from the superclass. If a future version of WebSphere Commerce then introduces a new method, called `newMethod` in the interface, the corresponding version of the `OrderNotifyCmdImpl` command will include a default implementation of the `newMethod` method. Then, since your new command extends from `OrderNotifyCmdImpl`, the compiler will find the default implementation of this new method in the `OrderNotifyCmdImpl` command and your new command is shielded from the interface change.

Refer to the "Reference" topic of the WebSphere Commerce Production and Development online help to ensure that the new implementation class provides the same external behavior as the existing class.

Data bean customization

A data bean normally extends an access bean. The access bean, which can be generated by WebSphere Studio Application Developer, provides a simple way to access information from an entity bean. When modifications are made to an entity bean (for example, adding a new field, a new business method or a new finder), the update is reflected in the access bean as soon as the access bean is regenerated. Since the data bean extends the access bean, it automatically inherits the new attributes. As a result of this relationship, no coding is required to enable the data bean to use new attributes from the entity bean.

If you need to add new attributes to a data bean that are not derived from an entity bean, you can extend the existing data bean using Java inheritance. For example, if you want to add a new field to the OrderDataBean, define MyOrderDataBean as follows:

```
public class MyOrderDataBean extends OrderDataBean
{
     public String myNewField () {
     //  implement the new field here
     }

}
```

The new data bean must also have a BeanInfo class. A sample of the declaration for this class follows:

```
public class MyOrderDataBeanInfo extends java.beans.SimpleBeanInfo
{

}
```

WebSphere Studio Application Developer provides a tool that allows you to generate this BeanInfo class.

Chapter 7. Trading agreements and business policies (Business Edition)

This chapter only applies to WebSphere Commerce Business Edition.

Introduction

One of the key elements of B2B (business-to-business) commerce is relationship management. A trading agreement is used to manage a business relationship between a buyer and a seller organization. The trading agreement model used by WebSphere Commerce Business Edition supports various types of trading agreements, such as Contract and RFQ (request for quote).

The main element of a trading agreement is a set of terms and conditions. Each term and condition defines a specific business rule to be used during trading. Using WebSphere Commerce Business Edition, a set of terms and conditions can be negotiated using an RFQ online process, or negotiated offline and then captured using the business relationship management interfaces in WebSphere Commerce Accelerator.

There are several ways to model a term and condition:

- A term and condition that selects one of the predefined business policies, such as a Price list and a Return policy. Or it can select a business policy that you have created. One term and condition object can also refer to multiple business policy objects.
- A term and condition that applies a specific adjustment to the business policy, such as an adjustment to the standard pricing.
- A term and condition that defines a set of parameters that govern a business process. For example, it could specify that a particular fulfillment center is to be used by a specific contract.

A contract is made up of a set of terms and conditions. This is shown in the following diagram.

© Copyright IBM Corp. 2000, 2003

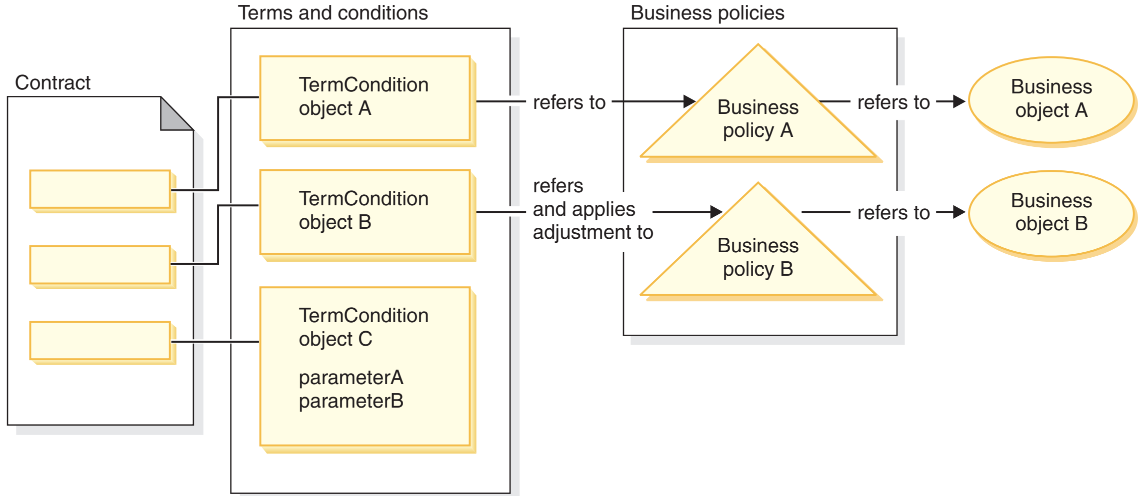


Figure 29.

From the preceding diagram, note the following:

- The term "adjustment" refers to a modification to the business policy. As an example, it can be used to apply a discount to the result of a business policy such that a 10% discount gets applied to the standard price. It can also be used to influence the business policy with a set of parameters.
- As an example, in the diagram the TermCondition object A may represent a shipping term and condition object. In this case, the business policy A may represent a shipping mode business policy and the business object A represents the shipping mode "A3" of shipping carrier XYZ.
- An another example, in the diagram the TermCondition object B may represent a price term and condition object that applies a 50% discount of the price defined by business policy B. In this case, business policy B is a price policy and business object B is a trading position container that defines the trading position for the master catalog.

This chapter provides guidelines for programmers on how to create new business policies and new terms and conditions.

The ToolTech sample store demonstrates a shipping term and condition object and a price term and condition object in its business flow. For more information about the contract data that supports these examples, refer to "ToolTech sample contract data" on page 160.

Business policy objects and commands

A business policy object contains the following information:

- Policy ID
  This is the primary key for the business policy object.
- Policy type
  This defines the business policy type. `Price` and `ProductSet` are examples of policy types.
- Policy name
  Each business policy must have a unique name.
- Store entity
  The store or store group in which the business policy is deployed.
- Properties
  A set of default properties that can be passed to the business policy command. The commands associated with the business policy object are stored in the BusinessPolicyCmd table.
- Effective period
  The period for which the business policy object is effective.
- Business policy command
  Zero or more business policy commands that implement the business policy. A business policy command is typically invoked by a business process that can be either a task command or a controller command. For example, the `getContractPrice()` command gets the price for a term and condition. This price term and condition refers to a particular price policy command and this price policy command is used to calculate the price.

Multiple business policy commands can be associated with a single business policy object. Each business policy command must implement the same interface defined by the business policy type object. The structure of a new business policy command is depicted in the following diagram:

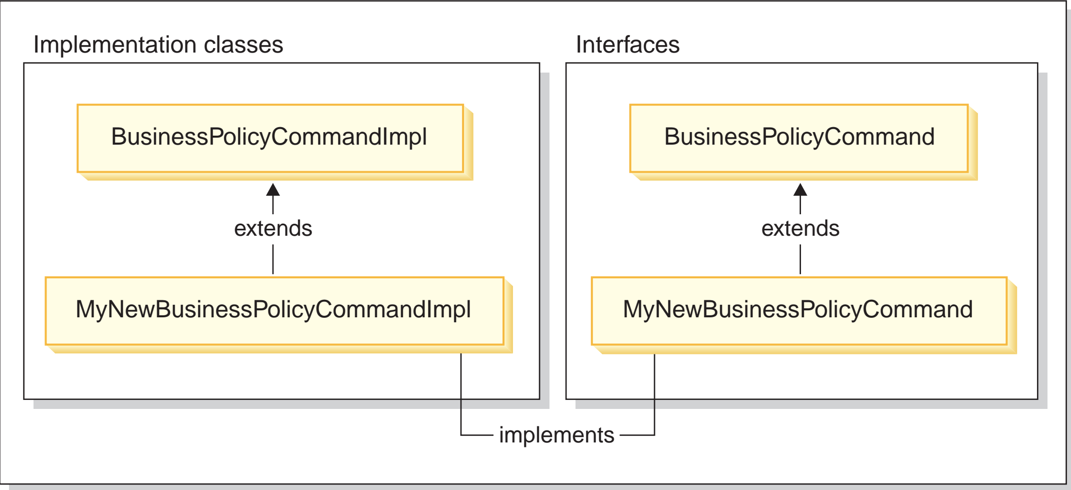


Figure 30.

As shown in the preceding diagram, in order to create a new business policy command, you create a new implementation class that extends the WebSphere Commerce BusinessPolicyCmdImpl implementation class. You also create a new interface that extends the BusinessPolicyCmd interface.

ToolTech sample contract data

This section provides an introduction to some of the contract data that is used in the ToolTech sample store.

The sample data in the following sections is organized by database table. Only the relevant rows and columns are displayed. Also note that when the sample is installed any unique identifiers (such as CONTRACT_ID) may have different values than what is shown here.

CONTRACT table sample data

The following table shows relevant sample data from the CONTRACT database table. Note that for display purposes, the database column headings are shown the first column and the row of sample data from the table is shown in the second column.

Column name	Sample data
CONTRACT_ID	10007
MAJORVERSION	1
MINORVERSION	0

Column name	Sample data
NAME	ToolTechContractNumber 4567
MEMBER_ID	-2001
ORIGIN	0
STATE	3
USAGE	1
MARKFORDELETE	0

TERMCOND table sample data

The following table shows relevant sample data from the TERMCOND database table. Note that for display purposes, the database column headings are shown the first column and the rows of sample data from the table are shown in the second and third columns.

Column name	Sample data row 1	Sample data row 2
TERMCOND_ID	10025	10030
TCSUBTYPE_ID	PriceTCPriceListWith SelectiveAdjustment	ShippingTCShippingMode
TRADING_ID	10007	10007
STRINGFILED1	ProductSet2	
INTEGERFIELD2	10002	
INTEGERFIELD3	1	
BIGINTFIELD1	10051	
FLOATFIELD1	-50.0	
SEQUENCE	1	6

POLICYTC table sample data

The following table shows relevant sample data from the POLICYTC database table. This table establishes the relationship between a policy and a terms and conditions object.

	Column name	
	POLICY_ID	TERMCOND_ID
Sample data row 1	10053	10025
Sample data row 2	10056	10030

POLICY table sample data

The following table shows relevant sample data from the POLICY database table.

Column name	Sample data row 1	Sample data row 2
POLICY_ID	10053	10056
POLICYNAME	MasterCatalogPriceList	A3
POLICYTYPE_ID	Price	ShippingMode
STOREENT_ID	10051	10051
PROPERTIES	name=ToolTech& member_id=-2001	shippingMode=A3
STARTTIME	null	null
ENDTIME	null	null

TRADEPOSCN table sample data

The following table shows relevant sample data from the TRADEPOSCN database table.

	Column name			
	TRADEEPOSCN_ID	**MEMBER_ID**	**NAME**	**TYPE**
Sample data row	10051	-2001	ToolTech	S

SHIPMODE table sample data

The following table shows relevant sample data from the SHIPMODE database table.

	Column name			
	SHIPMODE_ID	**STOREENTITY_ID**	**CODE**	**CARRIER**
Sample data row	10053	10051	A3	XYZ Carrier

Extending the existing contract model

A contract can be made up of one or more terms and conditions objects, each of which refers to a policy. As such, the subsequent sections describe the steps necessary for creating a new business policy and integrating this into your business flow.

As a brief overview, the following are the high-level steps for performing this task:

1. Creating a new business policy.
   The following tasks are related to creating a new business policy command:
   a. Create a new business policy type (if required).
      Several business policy types are provided, but if the standard types do not suit your business requirements, create a new business policy type.
   b. Creating a new business policy command.
   c. Register the new business policy and business policy command.
2. Relate a terms and conditions object to the new business policy.
   This can be done by relating an existing terms and conditions object to the new business policy, or by creating a new terms and condition object. If you create a new terms and conditions object, then you must perform the following steps:
   a. Register the new term and condition in the database
   b. Register the new term and condition in the contract XSD (XML schema definition)
   c. Creating a new CMP enterprise bean for the term and condition
   d. Updating WebSphere Commerce Accelerator to reflect the new term and condition
3. Invoking the new business policy during the business flow.

WebSphere Commerce Version 5.5 introduces new types of contracts. For information about the types of contracts available, refer to the “Business accounts” subtopic of the the “Concepts” topic in the WebSphere Commerce Production online help.

The following sections use the BuyerContract as the contract type in the extension example. Similar extension methodologies are used for the other contract types.

Creating a new business policy

Creating a new business policy typically involves registering a unique business policy in the database, as well as creating a new business policy command.

Creating a new business policy command involves the following high-level steps:
1. Creating a new business policy type (if required).
2. Writing the new business policy command.
3. Registering the new business policy and business policy command in the database.

Each of the preceding steps is described in more detail in the subsequent sections.

Creating a new business policy type

This section describes how to create a new business policy type. A business policy type indicates the realm of the transaction to which a policy applies. Examples of business policy types include:

- Price
- ProductSet
- ShippingMode
- ShippingCharge
- Payment
- ReturnCharge
- ReturnApproval
- ReturnPayment
- InvoiceFormat

If the existing business policy types do not satisfy your business requirements, you should create a new business policy type. Creating the new business policy type consists of defining and registering the business policy type.

When defining and registering a new policy type, you must update the following database tables:

- POLICYTYPE
- PLCYTYCMIF
- PLCYTYPDSC

The POLICYTYPE table specifies the type of business policy that you are creating. It contains a single column, POLICYTYPE_ID, that is the primary key. An example value is `Price`. If you create a new business policy type, ensure that you specify a unique POLICYTYPE_ID.

The PLCYTYCMIF table is the business policy type to command interface relationship specification table. That is, for each business policy type, it specifies the Java command interface for the business policy object. While there can be zero or more business policy commands that implement a business policy, each of the business policy commands must implement the interface specified here.

The PLCYTYPDSC table specifies a description of the business policy type. It includes a language identifier of the description and the description of the business policy type.

To create a new business policy type, create an entry in each of these tables for the new business policy type. The following SQL statements provides an example:

```
insert into POLICYTYPE (POLICYTYPE_ID) values ('MyNewPolicyType');
insert into PLCYTYCMIF (POLICYTYPE_ID, BUSINESSCMDIF)
   values ('MyNewPolicyType',
   'com.mycompany.mybusinesspolicycommands.MyNewPolicy');
insert into  PLCYTYPDSC (POLICYTYPE_ID, LANGUAGE_ID, DESCRIPTION)
   values ('MyNewPolicyType', -1,
   'My new policy type for example purposes.');
```

As the final step of creating the new business policy type, you may code one or more new business policy type interfaces. These interfaces are then implemented by any business policy command that falls under the realm of this business policy type. For example, in the ToolTech sample store, Price is defined as a business policy type. As such, there are the `com.ibm.commerce.price.commands.ResolvePriceListsCmd` and `com.ibm.commerce.price.commands.RetrievePricesCmd` interfaces that are implemented by all price-related business policy commands.

If you will not have a business policy command that performs operations on the new business policy type, then you are not required to create a new interface. This is rare, and in most cases when creating a new business policy type, you must create a new business policy type interface as well.

When you create a business policy type interface, the new interface must extend the `com.ibm.commerce.command.BusinessPolicyCommand` interface.

Writing the new business policy command

To create a new business policy command, you must create a new command that implements the interface of the business policy type to which the command relates. The new command must also extend `com.ibm.commerce.command.BusinessPolicyCommandImpl` implementation class. This is very similar to creating a new controller or task command.

There are two different approaches by which you can pass input properties to a business policy command. The first way is to have default input properties specified in the PROPERTIES column of the POLICY table. For more information about this table, refer to the following section.

The second approach is to create a new field in the command for each of the input properties. For each field, create a new pair of getter and setter methods.

Setting requestProperties in business policy commands

There are two ways in which requestProperties are set in a business policy command object. The first way uses the PROPERTIES column of the POLICY

table to set the default properties. This is accomplished by the setRequestProperties method. The second way to set properties is to have the command (controller or task) that calls the business policy command explicitly set other required properties.

When creating a new business policy command, you should override the default setRequestProperties method to include the logic to explicitly set each of the parameters that are included in the requestProperties object.

Consider an example of a new business policy command that has an interface name of MyNewBusinessPolicyCmd and implementation class name of MyNewBusinessPolicyCmdImpl.

Assume that the entry in the POLICY table for this new business policy command includes the following values in the PROPERTIES column:

- defaultProperty1=apple
- defaultProperty2=orange
- defaultProperty3=banana

The interface for this new business policy command is defined as follows:

```
public interface MyNewBusinessPolicyCmd extends
   com.ibm.commerce.command.BusinessPolicyCmd {
      java.lang.String defaultCommandClassName =
         'com.mycompany.mycommands.MyNewBusinessPolicyCmdImpl';
      public void setProperty1();
      public void setProperty2();
}
```

The implementation class for this new business policy command is defined as follows:

```
public class MyNewBusinessPolicyCmdImpl extends
   com.ibm.commerce.command.BusinessPolicyCmdImpl
   implements com.mycompany.mycommands.MyNewBusinessPolicyCmd {
      // Establish default properties that are stored in the POLICY table

      private java.lang.String defaultProperty1;
      private java.lang.String defaultProperty2;
      private java.lang.String defaultProperty3;

      // Begin to establish properties that must be set
     // by the calling command.

      //  *** property1   ***
      private java.lang.String property1;
      public java.lang.String getProperty1() {
         return property1;
         }
      public void setProperty1(java.lang.String newProperty1) {
         property1 = newProperty1;
```

```
      }

      //  *** property2   ***
      private java.lang.String property2;
      public java.lang.String getProperty2() {
         return property2;
         }
      public void setProperty1(java.lang.String newProperty2) {
         property2 = newProperty2;
         }

      // End establishing properties that must be set
      // by the calling command.

      /* Upon instantiation the business policy command sets all
         default properties from the POLICY table into the
         requestProperties object.  The calling command
         is responsible for setting any other required properties.
      */

      public void setRequestProperties(com.ibm.commerce.datatype.TypedProperty
         requestProperties) {
         //  Get the default properties defined in the POLICY table
         setDefaultProperty1(requestProperties.get("defaultProperty1"));
         setDefaultProperty2(requestProperties.get("defaultProperty2"));
         setDefaultProperty3(requestProperties.get("defaultProperty3"));
         }

}
```

The command that calls the new business policy command could be defined in a manner similar to the following:

```
public class MyCallerCommandImpl
   extends com.ibm.commerce.command.TaskCommandImpl
   implements com.mycompany.mycommands.MyCallerCommand {

   /* Include all elements and processing required for the
      task command.
   */

   // Determine the policy ID and setPolicyId

   // Call the business policy command.

   cmd = (MyNewBusinessPolicyCmd) CommandFactory
            createPolicyCommand(policyId);

   // Set required properties

   cmd.setProperty1("Fruit salad");
   cmd.setProperty2("Favorite food");

 cmd.execute();
}
```

Registering the new business policy and business policy command

After you have created the new business policy command, you must register both the business policy and the business policy command in the database.

Business policies are registered in the POLICY table. This table contains the following columns:

- POLICY_ID
  The primary key. This is the policy identifier.
- POLICYNAME
  A unique policy name.
- POLICYTYPE_ID
  The policy type identifier. This is the foreign key to the POLICYTYPE table.
- STOREENT_ID
  The store or store group to which the policy applies.
- PROPERTIES
  Default properties that can be set to the business policy command. Specified as name-value pairs, for example, `parm1=val1&parm2=val2`.
- STARTDATE
  The starting date (specified as a timestamp) of the policy. If NULL, the starting date is immediate.
- ENDDATE
  The ending date (specified as a timestamp) of the policy. If NULL, there is no end date.

Once the new policy is registered in the POLICY table, you must register a relationship between the policy and the business policy command that implements the business policy. The POLICYCMD table is used for this purpose. The POLICYCMD table contains the following columns:

- POLICY_ID
  Foreign key reference to the POLICY table.
- BUSINESSCMDCLASS
  The business policy command that implements the policy.
- PROPERTIES
  Default properties that can be set to the business policy command. Specified as name-value pairs, for example, `parm1=val1&parm2=val2`.

Relating a terms and conditions object to a new business policy

In the WebSphere Commerce contracts and policies framework, terms and conditions (also referred to as *terms*) provide a way to describe an agreement between a buyer and a seller. Terms and conditions can be used in various types of trading agreements, such as contract and RFQ (request for quotation). Terms and conditions objects usually refer to business policies with an

optional adjustment. For example, a price terms and conditions object is created by choosing one of the price policy objects. In the price term, an account manager can make adjustments to the store standard price, such as:

- A percentage discount over the standard price list
- A percentage discount on a specified set of the products

Each of the adjustments is specified as a term and condition.

When you create a new business policy, there must be at least one terms and conditions object that refers to this business policy, if the policy is to be used in a contract. You can either relate an existing term and condition object to the new business policy (this is done by capturing the relationship between the existing terms and conditions object and the new business policy in the XSD (XML schema definition) files), or you can create a new terms and conditions object that is related to the new business policy.

Creating new terms and conditions

Within the WebSphere Commerce architecture, new terms and conditions objects are created by performing the following steps:

1. Updating the database schema to include the new term and condition.
2. Updating XSD files to reflect the new term and condition.
3. Creating a new enterprise bean for the term and condition.
4. Updating WebSphere Commerce Accelerator to reflect the new term and condition, or using the contract load command to create a new contract using the new term and condition.

In the following sections, the example of `MyTC` is the new term and condition object.

Registering the new term and condition in the database

When you are creating a new terms and condition object, you must update the database schema to include this object. The database tables that must be updated are TCTYPE and TCSUBTYPE.

The following SQL statement shows an example of how to register the new term and condition in the database:

```
insert into TCTYPE (TCTYPE_ID) values ('MyTC');
insert into TCSUBTYPE (TCSUBTYPE_ID, TCTYPE_ID, ACCESSBEANNAME,
   DEPLOYCOMMAND)
values ('MySubTC', 'MyTC',
   'com.ibm.commerce.contract.objects.MySubTCAccessBean',
   'packagename.MySubTCDeployCmd');
```

Register the new term and condition in the contract XSD

To make the new term and condition available in contracts, you must create a new XSD file that defines the new term and condition. You must also update the Package.xsd file to include the new XSD file.

To create the new XSD file, do the following:

1. Navigate to the following directory:
   - 400 *WC_userdir*/instances/*instanceName*/xml/trading/xsd
   - AIX Linux Solaris *WC_installdir*/xml/trading/xsd
   - Windows *WC_installdir*\xml\trading\xsd

   where *instanceName* is the name of your WebSphere Commerce instance.
2. In this directory, create a new XSD file. The following shows the XSD that would be used for the example MyTC. The file is CustomizedBuyerContract.xsd:

```
<?xml version="1.0"?>
<schema targetNamespace="http://www.ibm.com/WebSphereCommerce"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wc="http://www.ibm.com/WebSphereCommerce"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

  <!-- include basic trading agreement xsd -->

  <include schemaLocation="BuyerContract.xsd" />
  <complexType name="MyTCType">
        <complexContent>
            <extension base="wc:TermConditionType"/>
        </complexContent>
    </complexType>
    <element name="MySubTC" substitutionGroup="wc:AbstractCustomizedTC">
        <complexType>
            <complexContent>
                <extension base="wc:MyTCType">
                    <sequence>
                        <element ref="wc:ProductSetPolicyRef"/>
                    </sequence>
                    <attribute name="attr1" type="normalizedString"
                         use="required"/>
                    <attribute name="attr2" type="int" use="required"/>
                </extension>
            </complexContent>
        </complexType>
    </element>
</schema>
```

3. Save the new file.

Next, you must update the Package.xsd file to remove the BuyerContract.xsd file and instead include the CustomizedBuyerContract.xsd file , as follows:

1. Navigate to the following directory:
   - Windows `WC_installdir\xml\trading\xsd`
   - AIX Solaris Linux `WC_installdir/xml/trading/xsd`

- 400 *WC_userdir*/instances/*instanceName*/xml/trading/xsd

where *instanceName* is the name of your WebSphere Commerce instance.
2. Open the Package.xsd file in a text editor.
3. Locate the section about the BuyerContract.xsd and modify it as follows:

```
<!--include schemaLocation="BuyerContract.xsd"/-->
<include schemaLocation="CustomizedBuyerContract.xsd"/>
```

Creating a new CMP enterprise bean for the term and condition

You must create a new CMP enterprise bean for the term and condition object. The bean is created for the term and condition subtype.

Note that typically when creating new enterprise beans, you would place the beans into the WebSphereCommerceServerExtensionsData project, rather than including them into one of the EJB groups that contain WebSphere Commerce entity beans. In this case however, since all new entity beans for terms and conditions must inherit from the WebSphere Commerce TermCondition bean, you must place your new term and condition beans into the Enablement-RelationshipManagementData project. The following sections describe how to create the new enterprise bean, using the tools in WebSphere Studio Application Developer.

Create a new enterprise bean: To create the new CMP enterprise bean for the new term and condition, do the following:

1. Within the J2EE Hierarchy view, expand **EJB Modules**.
2. Right-click the **Enablement-RelationshipManagementData** module and select **New > Enterprise Bean**.
   The Enterprise Bean Creation wizard opens.
3. From the **EJB Project** drop-down list, the **Enablement-RelationshipManagementData** is already selected. Click **Next**.
4. In the Create an Enterprise Bean window, do the following:
   a. Select **Entity bean with container-managed persistence (CMP) fields**
   b. In the **Bean name** field, enter an appropriate name for your bean. For this example, enter `MySubTC`
   c. In the **Source folder** field, leave the default value that is specified (ejbModule).
   d. In the **Default package** field, enter `com.ibm.commerce.contract.objects`.
   e. Click **Next**.
5. In the Enterprise Bean Details window, do the following:
   a. From the **Bean supertype** drop-down list, select **TermCondition**.

b. Click **Add** to add a new CMP attributes.
   The Create CMP Attribute window opens. In this window, do the following:
   1) In the **Name** field, enter an appropriate name for the new CMP field. For this example, enter `attr1`.
   2) In the **Type** field, enter the appropriate data type for the field. For this example, enter `String`.
   3) Select the **Access with getter and setter methods** check box.
   4) Select **Promote getter and setter methods to remote interface** check box.
   5) Clear the **Make getter read-only** check box.
   6) Click **Apply**.
   7) Create another attribute. In the **Name** field, enter `attr2`.
   8) In the **Type** field, enter the appropriate data type for the field. For this example, enter `Integer`.
   9) Clear the **Make getter read-only** check box.
   10) Click **Apply**.
   11) Click **Close** to close this window.

6. Click **Finish**.

Map the fields from the new bean into the TERMCOND table: The next step is to map the fields from the new bean to columns in the TERMCOND table. To create this mapping, do the following:

1. Switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the following folders: **Enablement-RelationshipManagementData > ejbModule > META-INF**.
3. Double-click the **Map.mapxmi** file.
4. In the Enterprise Beans pane, expand the **TermCondition** bean, then expand the **MySubTC** bean, so that its attributes can be viewed.
5. In the Tables pane, expand the **TERMCOND** table so that its columns can be viewed.
6. Drag the **attr1** field from the MySubTC onto the **STRINGFIELD3** column in the TERMCOND table.
7. Drag the **attr2** field from the MySubTC onto the **INTEGERFIELD3** column in the TERMCOND table.
8. Save your changes.

Adding a new ejbCreate method: In this step, you add a new ejbCreate method to the MySubTC bean, by doing the following:

1. In the J2EE Hierarchy view, double-click the **MySubTCBean** class to open it and view its source code.
2. Create a new ejbCreate(Long, Element) method, by adding the following code into the class:

```
public com.ibm.commerce.contract.objects.TermConditionKey
   ejbCreate(java.lang.Long argTradingId,
   org.w3c.dom.Element argElement)
      throws javax.ejb.CreateException,
     javax.ejb.FinderException,
     javax.naming.NamingException,
     javax.ejb.RemoveException {
          _initLinks();
          super.ejbCreate (argTradingId, argElement);
           this.attr1= null;
           this.attr2 = null;
           return null;
}
```

   Save the code changes.
3. You must add the new ejbCreate(Long, Element) method to the home interface. This makes the method available in the generated access bean. To add the method to the home interface, do the following:
   a. Right-click the **ejbCreate(Long, Element)** method in the Outline view and select **Enterprise Bean > Promote to Home Interface**.

Adding a new ejbPostCreate method: Next create a new ejbPostCreate(Long, Element) method so that it has the same parameters as the ejbCreate(Long, Element) method, by doing the following:

1. Double-click the **MySubTCBean** class to open it and view its source code.
2. Create a new ejbPostCreate(Long, Element) method, by adding the following code into the class:

```
public void ejbPostCreate(java.lang.Long argTradingId,
   org.w3c.dom.Element argElement)
      throws javax.ejb.CreateException,
     javax.ejb.FinderException,
     javax.naming.NamingException,
     javax.ejb.RemoveException
        {
           parseXMLElement(argElement);
        }
```

   Save the code changes.

Adding a parseXMLElement method: In this step, you must create the parseXMLElement method in the MySubTCBean, as follows:

1. Double-click the **MySubTCBean** class to open it and view its source code.
2. Update the method as follows:

```
public void parseXMLElement(org.w3c.dom.Element argElement) throws
    javax.ejb.CreateException,
    javax.ejb.FinderException,
    javax.naming.NamingException,
    javax.ejb.RemoveException
{
 super.parseXMLElement(argElement);
 if (argElement == null)
  return;
 String nodeName = argElement.getNodeName();
 if (nodeName.equals("TCCopy"))
  return;

 this.attr1 = argElement .getAttribute("attr1").trim();
 this.attr2 = new Integer (argElement.
     getAttribute("attr2").trim());
 // get element "ProductSetPolicyRef" from "MySubTC"
 Element ePolicyReference = null;
 ePolicyReference = ContractUtil.getElementByTag(
      argElement, "ProductSetPolicyRef");

 parseElementPolicyReference(ePolicyReference);

}
```

3. Save your work.

Adding a createNewVersion method: In this step you must create a new createNewVersion method in the MySubTCBean, as follows:

1. Double-click the **MySubTCBean** class to open it and view its source code.
2. Update the method as follows:

```
public Long createNewVersion(Long argNewTradingId) throws
  javax.ejb.CreateException,
  javax.ejb.FinderException,
  javax.naming.NamingException,
  javax.ejb.RemoveException,
  org.xml.sax.SAXException,
  java.io.IOException
{
   // Contract a seqElement since tcSequence can not be null
   Element seqElement = ContractUtil.
      getSeqElementFromTCSequence(this.tcSequence);
   MySubTCAccessBean newTC = new MySubTCAccessBean(
      argNewTradingId, seqElement);
   Long newTCId = newTC.getReferenceNumberInEJBType();
   newTC.setInitKey_referenceNumber(newTCId);
   newTC.setMandatoryFlag(this.mandatoryFlag);
   newTC.setChangeableFlag(this.changeableFlag);
   // set columns for this specific TC
   newTC.setAttr1(this.attr1);
   newTC.setAttr2(this.attr2);
   newTC.commitCopyHelper();
   return newTCId;
}
```

3. Save your work.

Adding a getXMLString method: In this step you must create a new getXMLString method in the MySubTCBean, as follows:

1. Double-click the **MySubTCBean** class to open it and view its source code.
2. Override the method as follows:

```
public String getXMLString() throws javax.ejb.CreateException,
   javax.ejb.FinderException,
   javax.naming.NamingException
{
 return getXMLString(false);
}


public String getXMLString(boolean tcdata) throws
 javax.ejb.CreateException,
 javax.ejb.FinderException,
 javax.naming.NamingException
{
 String xmlTC = " <MySubTC %TC_DATA% " +
  "  attr1=\"" + this.attr1 +
  "\" attr2=\"" + this.attr2.toString() + "\">" +
  "%TC_DESC%" +
  "%PARTICIPANT%" +
  "%XML_POLICYREFERENCE%" +
  " </MySubTC>";
 xmlTC = ContractUtil.replace( xmlTC, "%TC_DATA%",
      getXMLStringForTCData(tcdata));
 String xmlPolicy = getXMLStringForElementPolicyReference(
     "ProductSet");
 xmlTC = ContractUtil.replace( xmlTC, "%XML_POLICYREFERENCE%",
     xmlPolicy);
 xmlTC = ContractUtil.replaceAll( xmlTC, "%POLICY_REF_TYPE%",
     "ProductSetPolicyRef");
 return xmlTC;
}
```

3. Save your work.

Adding a markForDelete method: In this step you must create a new markForDelete method in the MySubTCBean, as follows:

1. Double-click the **MySubTCBean** class to open it and view its source code.
2. Override the method as follows:

```
public void markForDelete() throws
 javax.ejb.CreateException,
 javax.ejb.FinderException,
 javax.naming.NamingException
{
 // code: remove entries from associated tables which
 // cannot be deleted though delete cascade
}
```

3. Save your work.

Updating the remote interface: You must ensure that the following methods have been added to the remote interface, as follows:

1. Switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the **Enablement-RelationshipManagementData** project.
3. Expand the **com.ibm.commerce.contract.objects** package.
4. Double-click the **MySubTCBean** bean.
5. In the Outline view, right-click the **getXMLString()** method and select **Enterprise Bean > Promote to Remote Interface**
6. In the Outline view, right-click the **getXMLString(boolean tcdata)** method and select **Enterprise Bean > Promote to Remote Interface**
7. In the Outline view, right-click the **parseXMLElement(org.w3c.dom.Element argElement)** method and select **Enterprise Bean > Promote to Remote Interface**
8. In the Outline view, right-click the **createNewVersion(Long argNewTradingId)** method and select **Enterprise Bean > Promote to Remote Interface**
9. In the Outline view, right-click the **markForDelete()** method and select **Enterprise Bean > Promote to Remote Interface**
10. Save your changes.

Creating an access bean for MySubTC: To create the access bean, do the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**, then right-click **Enablement-RelationshipManagementData** and select **New > Access Bean**.
   The Add an Access Bean window opens.
2. Select **Copy Helper** and click **Next**.
3. Select the ***MySubTC***bean and click **Next**.
4. From the Constructor method drop-down list, select **findByPrimaryKey(com.ibm.commerce.contract.objects.***MySubTC***Key)** as the constructor method.
5. Select all attributes in the Attribute Helpers section.
6. Click **Finish**.
7. Save your work.

Adding new string converters: You must add new string converters, as follows:

1. In the J2EE Hierarchy view, expand **EJB Modules > Enablement-RelationshipManagementData > ejbModule > META-INF**.
2. Double-click the **ibm-ejb-access-bean.xmi** file.
3. Locate the section about MySubTC.
4. For each copyHelperProperties element, add the following attribute:
   ```
   converterClassName="com.ibm.commerce.base.objects.WCSStringConverter"
   ```
5. Save your work.
6. Next you must regenerate the access bean for MySubTC, as follows:
   a. Right-click **Enablement-RelationshipManagementData** and select **Access Beans > Regenerate Access Beans**.
   b. Select **MySubTC** and click **Finish**.

Generating deployed code: You must generate the deployed code for both the MySubTC bean as well as the TermCondition bean, as follows:
1. In the J2EE Hierarchy view, expand **EJB Modules** .
2. Right-click **Enablement-RelationshipManagementData** and select **Generate > Deploy and RMIC Code**.
3. Select **MySubTC** and click **Finish**.
4. Right-click **Enablement-RelationshipManagementData** and select **Generate > Deploy and RMIC Code**.
5. Click **Select All** and click **Finish**.

   Note: Technically, you must regenerate the deployed code for the parent bean (the TermCondition bean) and all of the sibling beans (all of the other beans in the Enablement-RelationshipManagementData group that contain “TC” in their names). Note that if you had added a new field or modified the remote interface of the existing TermCondition bean, then you would have to regenerate the access beans for itself and all of its child beans as well. For simplicity, the preceding instructions select all of the beans in this project.

Overriding methods in the validateContract task command: The next step is to override methods that are in the `ValidateContractCmd` task command. In this command, there are three methods that you may want to override to support the new term and condition object. They are:
- `validateTCType()`
  This method checks what type of term can be in a contract. For example, the InvoiceTC belongs to account and therefore, it cannot appear in a contract.
- `validateTCOccurrence()`
  This method checks the occurrence of the terms. For example, in the default implementation of this method, a contract has to have at least one PriceTC.

- otherValidateCheck()
  The default implementation of this method is empty. You can add any additional validation that does not fall into the first two methods.

For details about how to make this modification, refer to "Customizing existing task commands" on page 154.

Creating a new deployment command: If the term and condition must be deployed, you must create a new deployment command and register this command in the database. If required, do the following:

1. In this example, the new deployment command interface is called MySubTCDeployedCmd and the implementation class is called MySubTCDeployedCmdImpl. In addition, the command is packaged in the packagename package. To register this command, issue the following SQL command:

```
insert into CMDREG (STOREENT_ID, INTERFACENAME, CLASSNAME, TARGET)
values (0, 'packagename.MySubTCDeployCmd',
'packagename.MySubTCDeployCmdImpl', 'Local');
```

2. In the packagename package, create the new MySubTCDeployedCmd interface. This interface must extend the com.ibm.commerce.contract.commands.DeployTCCmd command interface. The following describes the new command interface:

```
public interface MySubTCDeployCmd extends
   com.ibm.commerce.contract.commands.DeployTCCmd
   {
       // customized code
   }
```

   There is a protected parameter abTC and a method called getTargetStoreId() in DeployTCCmd. The value of abTC is MySubTCAccessBean and the getTargetStoreId() method returns the identifier of the store to which the contract is being deployed.

3. In the same package, create the MySubTCDeployCmdImpl implementation class. This implementation class must extend com.ibm.commerce.contract.commands.DeployTCCmdImpl. The following describes the new command implementation class:

```
public class MySubTCDeployCmdImpl
   extends com.ibm.commerce.contract.commands.DeployTCCmdImpl
   implements MySubTCDeployCmd
   {
     // customer code
   }
```

Updating WebSphere Commerce Accelerator to use a new term and condition

Once you have created new terms and conditions, you can update WebSphere Commerce Accelerator so that it can be used to create new contracts that

include those new terms and conditions. Updating WebSphere Commerce Accelerator for this purpose includes the following steps:

1. Creating a new JavaScript™ file for the new terms and conditions. For the purpose of the example in this section, this file is referred to as `Extensions.js`.
2. Creating a new JSP template that includes an HTML section in which a user can enter required information for the new terms and conditions. For the purpose of the example in this section, this file is referred to as `ContractMyTC.jsp`.
3. Creating a new data bean for the new terms and conditions. For the purpose of the example in this section, this file is referred to as `MyTCDataBean`.
4. Registering the new view in the VIEWREG table.
5. Updating the `ContractRB_`*`locale`*`.properties` file to include the new resources.
6. Editing the `ContractNotebook.xml` file to include the new page.

Each of these steps is described in more detail in the following sections.

Creating the new JavaScript file: The first step to updating the WebSphere Commerce Accelerator to use new terms and conditions is to create a new JavaScript file for them. For reference, you can refer to the following sample file:

- 400 AIX Linux Solaris *WC_installdir*/samples/contract/Extensions.js
- Windows *WC_installdir*\samples\contract\Extensions.js
- Developer *WCDE_installdir*\samples\contract\Extensions.js

In order to use this sample file, copy it to the following directory:

- 400 *WAS_userdir*/installedApps/*cell_name*/WC_*instanceName*.ear/CommerceAccelerator.war/tools/contract
- AIX Linux Solaris *WAS_installdir*/installedApps/*cell_name*/WC_*instanceName*.ear/CommerceAccelerator.war/tools/contract
- Windows *WAS_installdir*\installedApps*cell_name*\WC_*instanceName*.ear\CommerceAccelerator.war\tools\contract
- Developer *workspace_dir*\CommerceAccelerator\Web Content\tools\contract

where *instanceName* is the name of your WebSphere Commerce instance and *cell_name* is the WebSphere Application Server cell name.

In this new file, you must create a JavaScript object to store the data for the new term and condition. This is shown in the following code snip:

```
function ContractMyTCModel() {

   this.tcReferenceNumber = "";
   this.policyReferenceNumber = "";

   this.attr1 = "";
   this.attr2 = "";

   this.policyList = new Array();
   this.selectedPolicyIndex = "0";
}
```

You should also create a new JavaScript object to submit the new term and condition. This must be done in a manner consistent with the extensions that you made to the XSD files. This is shown in the following code snip:

```
function submitMyTC(contract) {

  var tcModel = get("ContractMyTCModel");

  if (tcModel != null) {

 var myTC = new Object();
 myTC.attr1 = tcModel.attr1;
 myTC.attr2 = tcModel.attr2;

 myTC.ProductSetPolicyRef = new Object();
 myTC.ProductSetPolicyRef.policyName = tcModel.policyList[
      tcModel.selectedPolicyIndex].policyName;
 myTC.ProductSetPolicyRef.StoreRef = new Object();
 myTC.ProductSetPolicyRef.StoreRef.name = tcModel.policyList[
      tcModel.selectedPolicyIndex].storeIdentity;
 myTC.ProductSetPolicyRef.StoreRef.Owner = new Object();
 myTC.ProductSetPolicyRef.StoreRef.Owner = tcModel.policyList[
      tcModel.selectedPolicyIndex].member;

 if (tcModel.tcReferenceNumber != "") {
  // Change the term and condition
  myTC.action = "update";
  myTC.referenceNumber = tcModel.tcReferenceNumber;
 }
 else {
  // Create a new term and condition
  myTC.action = "new";
 }

 contract.MySubTC = myTC;
  }

  return true;
}
```

Creating the new JSP template: The next step is to create a new JSP template that includes an HTML section in which the user can enter information required by the new term and condition. For reference, you can refer to the following sample file:

- 400 AIX Linux Solaris *WC_installdir*/samples/contract/ContractMyTC.jsp
- Windows *WC_installdir*\samples\contract\ContractMyTC.jsp
- Developer *WCDE_installdir*\samples\contract\ContractMyTC.jsp

In order to use this sample file, copy it to the following directory:

- 400 *WAS_userdir*/installedApps/*cell_name*/WC_*instanceName*.ear/CommerceAccelerator.war/javascript/tools/contract
- AIX Linux Solaris *WAS_installdir*/installedApps/*cell_name*/WC_*instanceName*.ear/CommerceAccelerator.war/ javascript/tools/contract
- Windows *WAS_installdir*\installedApps*cell_name*\WC_*instanceName*.ear\CommerceAccelerator.war\ javascript\tools\contract
- Developer *workspace_dir*\CommerceAccelerator\Web Content\tools\contract

where *instanceName* is the name of your WebSphere Commerce instance and *cell_name* is the name of the WebSphere Application Server cell.

The following code snip shows an example HTML section of a JSP template that can be used for MyTC.

```
<!--
///////////////////////////////////////
// HTML SECTION
///////////////////////////////////////
-->

<BODY onLoad="onLoad()" class="content">

   <H1>
   <%= contractsRB.get("MyTCHeading") %>
   </H1>

<FORM NAME="MyTCForm">

   <%= contractsRB.get("MyTCAttr1Label") %>
   <BR>
   <INPUT type=text name=Attr1 value="" size=10 maxlength=10>
   <BR>

   <%= contractsRB.get("MyTCAttr2Label") %>
   <BR>
   <INPUT type=text name=Attr2 value="" size=10 maxlength=10>
   <BR>
```

```
   <%= contractsRB.get("MyTCPolicyLabel") %>
   <BR>
   <SELECT NAME="PolicyList" SIZE="1">
   </SELECT>

</FORM>
```

Creating the new data bean: In this step, you create a new data bean that loads the necessary data from the MySubTC access bean. The relevant sections of code are shown in the following code snip:

```
public class MyTCDataBean extends MySubTCAccessBean
   implements SmartDataBean, Delegator {
      private java.lang.Long contractId;
      private boolean hasMyTC = false;
      private CommandContext iCommandContext;


      /**
       * MyTCDataBean default constructor.
       */
      public MyTCDataBean() {
      }
      /**
       * MyTCDataBean constructor.
       */
      public MyTCDataBean(Long newContractId) {
       contractId = newContractId;
      }


      /*
       * populate the attributes from TermConditionAccessBean
       */
      public void populate() throws Exception {

            Enumeration myTCEnum = new TermConditionAccessBean().
                     findByTradingAndTCSubType(contractId, "MySubTC");
                  if (myTCEnum != null) {
                        // assume a contract only has one MyTC for this example

                        setEJBRef(((TermConditionAccessBean)
                              myTCEnum.nextElement()).getEJBRef());
                        refreshCopyHelper();
                        hasMyTC = true;
                  }

}
```

Registering the new view in the VIEWREG table: You must register your newly created view in the VIEWREG table. The following is an example SQL statement to register the new view.

```
insert into VIEWREG(VIEWNAME,DEVICEFMT_ID,STOREENT_ID, INTERFACENAME,
   CLASSNAME, PROPERTIES, HTTPS, INTERNAL)
values ('ContractMyTCPanelView', -1, 0,
   'com.ibm.commerce.tools.command.ToolsForwardViewCommand',
   'com.ibm.commerce.tools.command.ToolsForwardViewCommandImpl',
   'docname=tools/contract/ContractMyTC.jsp', 1, 1)
```

Updating the ContractRB_*locale*.properties file: You must update the following properties file with information specific to the new term and condition:

- 400 *WAS_userdir*/installedApps/*cell_name*/WC_*instanceName*.ear/properties/com/ibm/commerce/tools/contract/properties/ContractRB_*locale*.properties
- AIX Linux Solaris *WAS_installdir*/installedApps/*cell_name*/WC_*instanceName*.ear/properties/com/ibm/commerce/tools/contract/properties/ContractRB_*locale*.properties
- Windows *WAS_installdir*\installedApps*cell_name*\WC_*instanceName*.ear\properties\com\ibm\commerce\tools\contract\properties\ContractRB_*locale*.properties
- Developer *workspace_dir*\WebSphereCommerceServer\properties\com\ibm\commerce\tools\contract\properties\ContractRB_*locale*.properties

where *instanceName* is the name of your WebSphere Commerce instance and *cell_name* is the name of the WebSphere Application Server cell.

The following is an example of the information that you would add to the file.

```
MyTCHeading=My TC
attr1Empty=Attribute One must be entered.
attr2Empty=Attribute Two must be entered.
attr1TooLong=Attribute One is too long.
attr2TooLong=Attribute Two is too long.
MyTCAttr1Label=Attribute One (required)
MyTCAttr2Label=Attribute Two (required)
MyTCPolicyLabel=Policy
```

Editing the ContractNotebook.xml file: The last step for including new terms and conditions in the WebSphere Commerce Accelerator is to update the following file to include the new page.

- 400 *WC_installdir*/xml/tools/contract/ContractNotebook.xml
- AIX Linux Solaris *WC_installdir*/xml/tools/contract/ContractNotebook.xml
- Windows *WC_installdir*\xml\tools\contract\ContractNotebook.xml

- Developer *WCDE_installdir*\Commerce\xml\tools\contract\ContractNotebook.xml

The following is an example snip of code that is used to include the new page in this example.

```
<panel name="MyTCHeading"
    url="ContractMyTCPanelView"
    parameters="contractId,accountId"
    helpKey="MC.contract.MyTCPanel.Help" />
```

Importing the new contract using the new term and condition

As an alternative to updating the WebSphere Commerce tools to use a new term and condition, you can use the contract import command (refer to the WebSphere Commerce online help for information about this command) to import a new contract that includes this new term and condition. After importing, the relevant section in the Contract.xml file appears as follows:

```
 <MySubTC attr1="abc" attr2="123">
        <ProductSetPolicyRef policyName = "Product Set 1">
            <StoreRef name = "StoreGroup1">
                <Owner>
                    <OrganizationRef distinguishName = "o=Root Organization"/>
                </Owner>
            </StoreRef>
        </ProductSetPolicyRef>
    </MySubTC>
```

Invoking the new business policy

Once you have created a new business policy and this business policy has been associated with at least one terms and conditions object, you can must update your application logic to invoke the new business policy commands.

Business policy commands are invoked from within controller and task commands.

The command factory is used to invoke business policy commands. There are two `create` methods that can be used to invoke business policy commands. The first is used to invoke a business policy command when there is only one business policy command associated with the business policy. This is shown in the following code snippet:

```
CommandFactory createBusinessPolicyCommand(Long policyId);
```

The second method is used to invoke a business policy command when there is more than one business policy command associated with the business policy. This is shown in the following code snippet:

```
CommandFactory createBusinessPolicyCommand(Long policyId, String cmdIfName);
```

In the preceding example, `cmdIfName` is used to specify the interface name of the business policy command to be created.

The command factory looks up the policy object in the POLICYCMD table to determine the command that implements this policy. It also fetches any default properties from the table and sets them as requestProperties in the business policy command.

The following code snippet shows an example of invoking a refund policy:

```
RefundPolicyCmd  cmd;

   ///////////////////////////////////////////////////////////
   // Get the refund policy id from the refundTC object     //
   //  and use it to create the policy command.            //
   ///////////////////////////////////////////////////////////
 cmd = (RefundPolicyCmd) CommandFactory
        createPolicyCommand (refundTC.getRefundPolicy);

  cmd.execute()
```

Creating a contract

The next step to fully integrate the extension to the contract model into your business process is to create a contract that includes the terms and condition which refers to the new business policy. A contract can be created using the WebSphere Commerce Accelerator or by using one of the contract URL commands (ContractImportApprovedVersion and ContractImportDraftVersion). For more information about creating contracts, refer to the WebSphere Commerce Production and Development online help.

Contract customization scenarios

This section provides an overview of the steps involved for the following contract customization scenario:

- Enabling a rebate

Rebate scenario

In this example scenario, a flat rate rebate is created. Since the ToolTech sample store includes neither a term and condition nor a policy type that matches the rebate scenario, these must be created. Additionally, a new business policy must be created, as well as a database table to store the rebate codes.

Implementing this rebate scenario includes the following high-level steps:

1. Creating the XREBATECODE database table and a corresponding XRebateCodeBean entity bean that is used to access information from this table.

2. Creating a new 5DollarRebate business policy by performing the following sub-tasks:
   a. Create the corresponding new business policy type. This defines the interface (RebatePolicyCmd) that the new business policy command will implement.
   b. Create the new CalculateRebateCmdImpl business policy command.
   c. Register the new business policy command and business policy type in the database.
3. Create a new term and condition (RebateTC) for the rebate by performing the following sub-tasks:
   a. Register the RebateTC term and condition in the database.
   b. Update the XSD files to reflect the new RebateTC.
   c. Create a new enterprise bean for the RebateTC.
   d. Update WebSphere Commerce Accelerator to reflect the new RebateTC.
4. Create a new contract that uses the RebateTC.
5. Integrating the new business policy into the shopping flow.

Each of these steps is described in more detail in subsequent sections.

Step 1: Creating the new table and enterprise bean

Since the existing database schema does not include the specification of a rebate amount and code, a new table must be created. In general, when a new table is created, a new entity bean is also created that is used to when accessing information contained in this table.

For the purpose of this example, assume that the following XREBATECODE database table is created.

Table 2. XREBATECODE database table

	Column name		
	REBATECODE_ID	AMOUNT	CURRENCY
Sample data	201	5	CAD
	202	10	CAD

Additionally, a new CMP entity bean (XRebateCodeBean) would be created. For detailed information about creating this bean, refer to “Creating a new CMP enterprise bean” on page 59.

Step 2: Creating the “5DollarRebate” business policy

In order to create this new business policy, you must perform the following steps:

1. Create the new business policy type interface. This is the RebatePolicyCmd interface that the CalculateRebateCmdImpl will implement.
2. Create the new CalculateRebateCmdImpl business policy command.
3. Register the new business policy and business policy command in the database.

Creating the "Rebate" business policy type: Since there is not an existing business policy type that corresponds to rebates, a new one must be created. Creating a new business policy type involves defining and registering a policy type in the database. The following tables must be updated:

- POLICYTYPE
- PLCYTYCMIF
- PLCYTYPDSC

For this scenario, to create the new REBATE policy type, the following SQL statements would be used:

```
insert into POLICYTYPE (POLICYTYPE_ID) values ('Rebate');
insert into PLCYTYCMIF (POLICYTYPE_ID, BUSINESSCMDIF)
   values ('Rebate',
   'com.mycompany.mybusinesspolicycommands.RebatePolicyCmd');
insert into  PLCYTYPDSC (POLICYTYPE_ID, LANGUAGE_ID, DESCRIPTION)
   values ('Rebate', -1,
   'Rebate policy type.');
```

As a result, the following table shows the relevant columns of the PLCYTYCMIF table that shows the relationship between the policy type and the business policy command to which it is related.

Table 3. Updates made to the PLCYTYCMIF table

	Column name	
	POLICYTYPE_ID	BUSINESSCMDIF
Sample data	Rebate	com.mycompany.mybusinesspolicycommands. RebatePolicyCmd

You must also code the new RebatePolicyCmd interface. This interface must extend the com.ibm.commerce.command.BusinessPolicyCommand interface. As suggested by the previous table, package this interface into your own package.

Creating the CalculateRebateCmdImpl business policy command: To create the new business policy command, you must create a new command called CalculateRebateCmdImpl that extends the

com.ibm.commerce.command.BusinessPolicyCommandImpl implementation class. This command should implement the RebatePolicyCmd interface created in the previous step.

Note that in this example, the interface name and the command name are dissimilar. These names were chosen to intentionally show that there may be many business policy commands that implement the rebate type of business policy. Each implementation (that is, each business policy command) would then implement the rebate in a unique manner.

The logic of the command depends upon the particular implementation of how the customer is to pick up the goods. Additionally, this CalculateRebateCmdImpl should be invoked by a separate controller or task command in your application.

Registering the new business policy and new business policy command: The new business policy must be registered in the database. You must also register the relationship between the new business policy and the new business policy command.

To register this information, you can use the com.ibm.commerce.contract.commands.PolicyAddCmd command. The following shows an example usage of the PolicyAdd command for this scenario:

```
http://localhost:8080/webapp/wcs/stores/servlet/PolicyAdd?
   type=Rebate&name=5DollarRebate&plcyStoreId=-1
   &cmd_1=com.mycompany.mybusinesspolicycommands.CalculateRebateCmdImpl
   &startDate=2002-05-08%2000:00:00&endDate=2003-05-09%2000:00:00
   &commonProps=rebatecode_id%3D501&URL=aRedirectURL
```

Note that URL reserved characters must be replaced by their ASCII codes for input properties. As such, the typical = (equals) symbol is replaced with "%3D", the & (ampersand) is replaced by "%26", and the space character is replaced by "%20". The date format used in the preceding example is yyyy-mm-dd hh:mm:ss, with ASCII code replacing URL reserved characters.

The following tables show the relevant columns of the affected database tables, after performing the updates.

Table 4. Updates made to the POLICY table

	Column name				
	POLICY _ID	POLICY NAME	POLICYTYPE _ID	STOREENT _ID	PROPERTIES
Sample data	301	5Dollar Rebate	Rebate	-1	rebatecode_id= 201

Note that it is also presumed that the start date and end date values are set to

null.

Table 5. Updates made to the POLICYCMD table

	Column name		
	POLICY _ID	BUSINESS CMDCLASS	PROPERTIES
Sample data	301	com.mycompany. mybusinesspolicycommands. CalculateRebateCmdImpl	null

As a result, you now have a new business policy called "5DollarRebate" that is related to the CalculateRebateCmd business policy command.

Step 3: Creating the "RebateTC" term and condition

Creating the "RebateTC" term and condition requires that the following steps be performed:

1. Register the RebateTC term and condition in the database.
2. Update the XSD files to reflect the new RebateTC.
3. Create a new enterprise bean for the RebateTC.
4. Update the WebSphere Commerce Accelerator to reflect the new RebateTC.

Registering the"RebateTC" term and condition in the database: When you are creating a new terms and condition object, you must update the database schema to include this object. The database tables that must be updated are TCTYPE and TCSUBTYPE.

The following SQL statement shows an example of how to register the RebateTC in the database:

```
insert into TCTYPE (TCTYPE_ID) values ('RebateTC');
insert into TCSUBTYPE (TCSUBTYPE_ID, TCTYPE_ID, ACCESSBEANNAME,
   DEPLOYCOMMAND)
values ('RebateTC', 'RebateTC',
   'com.ibm.commerce.contract.objects.RebateTCAccessBean',
   null);
```

The following tables show an extract of the relevant columns in the TCTYPE and TCSUBTYPE tables.

Table 6. Updates made to the TCTYPE table

	Column name
	TCTYPE_ID
Sample data	RebateTC

Table 7. Updates made to the TCSUBTYPE table

	Column name			
	TCSUBTYPE _ID	TCTYPE _ID	ACCESSBEAN NAME	DEPLOY COMMAND
Sample data	RebateTC	RebateTC	com.ibm.commerce. contract.objects. RebateTCAccessBean	null

Register the rebate term and condition in the contract document type definition: To make the rebate term and condition available in contracts, you must create a new XSD file that defines the new term and condition. You must also update the Package.xsd file to include the new XSD file.

To create the new XSD file, do the following:

1. Navigate to the following directory:
   - Windows *WC_installdir*\xml\trading\xsd
   - AIX Solaris Linux *WC_installdir*/xml/trading/xsd
   - 400 *WC_userdir*/instances/*instanceName*/xml/trading/xsd

   where *instanceName* is the name of your WebSphere Commerce instance.
2. In this directory, create a new XSD file. The following shows the XSD that would be used for the example RebateTC. The file is RebateCustomizedBuyerContract.xsd:

```
<?xml version="1.0"?>
<schema targetNamespace="http://www.ibm.com/WebSphereCommerce"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wc="http://www.ibm.com/WebSphereCommerce"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

  <!-- include basic trading agreement xsd -->

  <include schemaLocation="BuyerContract.xsd" />
  <complexType name="RebateTCType">
        <complexContent>
            <extension base="wc:TermConditionType"/>
        </complexContent>
    </complexType>
    <element name="RebateTC" substitutionGroup="wc:AbstractCustomizedTC">
        <complexType>
            <complexContent>
                <extension base="wc:RebateTCType">
                    <sequence>
                        <element ref="wc:RebatePolicyRef"/>
                    </sequence>
                </extension>
```

```
            </complexContent>
        </complexType>
    </element>

    <element name="RebatePolicyRef" type="wc:BusinessPolicyRef" />

</schema>
```

3. Save the new file.

Next, you must update the Package.xsd to remove the BuyerContract.xsd and instead include the RebateCustomizedBuyerContract.xsd, as follows:

1. Navigate to the following directory:
   - Windows *WC_installdir*\xml\trading\xsd
   - AIX Solaris Linux *WC_installdir*/xml/trading/xsd
   - 400 *WC_userdir*/instances/*instanceName*/xml/trading/xsd

   where *instanceName* is the name of your WebSphere Commerce instance.
2. Open the Package.xsd file in a text editor.
3. Locate the section about the BuyerContract.xsd and modify it as follows:

```
<!--include schemaLocation="BuyerContract.xsd"/-->
<include schemaLocation="RebateCustomizedBuyerContract.xsd"/>
```

Creating a new enterprise bean for the RebateTC: You must create a new enterprise bean for the new RebateTC. This new bean should inherit from the WebSphere Commerce TermCondition bean.

A new enterprise bean for a term and condition is typically named after the subtype. Note that in this case, the term and condition subtype is the same as the term and condition type, and as such, the name of the bean is the same as the term and condition type.

The following table shows some of the general information about the new bean that must be created. For more details about the bean, including which methods to override, refer to “Creating a new CMP enterprise bean for the term and condition” on page 171.

Table 8.

Attribute	Value
EJB Project	Enablement-RelationshipManagementData
Bean type	Entity bean with container-managed persistence fields
Bean name	RebateTC
Package	com.ibm.commerce.contract.objects

Table 8. (continued)

Attribute	Value
Bean supertype	`TermCondition`
Bean class	`RebateTCBean`

In the new bean, create three CMP fields that are used for the following values:

- rebate code ID
- amount
- currency

Updating WebSphere Commerce Accelerator to include the RebateTC: Once you have created new terms and conditions, you can update WebSphere Commerce Accelerator so that it can be used to create new contracts that include those new terms and conditions. For information about how to update this tool, refer to "Updating WebSphere Commerce Accelerator to use a new term and condition" on page 178.

Step 4: Creating a new contract

You must create a new contract that includes the "RebateTC" term and condition and that refers to the "5DollarRebate" business policy. You can use either the WebSphere Commerce Accelerator or XML to create a new contract. Each of these methods for creating new contracts are described in the WebSphere Commerce Production and Development online help.

The following tables show the updates to the relevant columns of the TERMCOND and POLICYTC database tables, after the contract has been created.

Table 9. Updates made to the TERMCOND table

	Column name		
	TRADING _ID	TERMCOND _ID	TCSUBTYPE _ID
Sample data	25	901	RebateTC

Table 10. Updates made to the POLICYTC table

	Column name	
	POLICY _ID	TERMCOND_ID
Sample data	301	901

Step 5: Integrating the new business policy into the shopping flow

In this scenario, it is presumed that a new page is added to the store that allow customers to log on and claim their rebates. When the customer would click to claim the rebate, a command that invokes the new `RebatePolicyCmd` interface should be invoked. For example, there could be a new ClaimRebateCmd controller command that invokes the `RebatePolicyCmd`. The correct business policy is then found and (in this case) the "5DollarRebate" business policy is applied.

Part 3. Development environment

Chapter 8. Development environment

This chapter introduces the main development tools used for customizing a WebSphere Commerce application.

Typical development environment

Business The recommended development package for creating customized code to be used with WebSphere Commerce Business Edition is the WebSphere Commerce Studio, Business Developer Edition product. Professional The recommended development package for creating customized code to be used with WebSphere Commerce Professional Edition is the WebSphere Commerce Studio, Professional Developer Edition product. Express The recommended development package for creating customized code to be used with WebSphere Commerce - Express is the WebSphere Commerce - Express Developer Edition product. All of these packages include the tools you require to create customized code and perform Web development tasks. In general, this book refers to these products collectively as the WebSphere Commerce development environment.

There are four main components to the WebSphere Commerce development environment:

1. The WebSphere Commerce workspace that is used within WebSphere Studio Application Developer
2. The development database
3. File system assets
4. WebSphere Commerce plug-ins to WebSphere Studio Application Developer

With this development environment, you can create customized code and test it within the context of the WebSphere test environment.

To create stores in the development environment, you simply launch the Administration Console running on the locally defined WebSphereCommerceServer test server in WebSphere Studio Application Developer and use that tool to publish a store based on one of the sample stores. Alternatively, you can create your own store.

In addition to the WebSphere Commerce workspace that is provided, the WebSphere Commerce development environment provides additional tools and plug-ins. The following plug-ins are provided:

© Copyright IBM Corp. 2000, 2003

- A Configuration Manager plug-in that assists with managing your WebSphere Commerce (and WebSphere Commerce Payments) instances.
- A WebSphere Commerce online help plug-in allows you to access the WebSphere Commerce online help from within WebSphere Studio Application Developer. Additionally, with this plug-in, the WebSphere Commerce context-sensitive online help can be launched by pressing F1 when WebSphere Commerce tools (for example, the Administration Console) are running.
- A WebSphere Commerce API reference information plug-in that allows you to access the WebSphere Commerce API reference information from within WebSphere Studio Application Developer.

* A WebSphere Commerce enterprise bean conversion tool is also provided.
* Using this tool, you can develop enterprise beans using a development
* database that is different from your target production database. For example,
* this tool allows you to use a local DB2 database for development, yet use a
* DB2 database on the iSeries platform, a DB2 for OS/390 and z/OS database,
* or even an Oracle database for production.

WebSphere Studio Application Developer

The WebSphere Commerce development environment package includes WebSphere Studio Application Developer which is the core development environment from IBM. It helps you to optimize and simplify Java 2 Enterprise Edition (J2EE) and Web services development by offering best practices, templates, code generation, and the most comprehensive development environment in its class. This sophisticated integrated development environment (IDE) includes integrated support for Java components, enterprise beans, servlets, JSP files, HTML, XML, and Web services all in one development environment.

Amongst many other exciting features, it also includes local test tools that allow you to quickly generate test clients. It also includes a full WebSphere Application Server test environment that allows you to test your code end-to-end in a local environment.

Development environment for iSeries

400 In short, a special development environment is not required in order to create customized code for iSeries. The development workstation is set up following the same procedure that is outlined in the Business Professional *WebSphere Commerce Studio Installation Guide* or Express *WebSphere Commerce - Express Developer Edition Installation Guide*. The local development database used should be DB2. Using this configuration, customized code can be created and tested using the local DB2 database and local WebSphereCommerceServer test server.

After testing determines that the customized code functions to your satisfaction within the context of the test server, you must then deploy it to a target WebSphere Commerce Server running on the iSeries platform. In order to account for differences between the database on the Windows and iSeries platforms, a WebSphere Commerce enterprise bean conversion tool is provided. More information about this tool is provided in “Overview of the WebSphere Commerce enterprise bean conversion tool.”

Using different database management systems for development and production

It is possible for developers to use a local DB2 database on their development machines even though the database for the production environment will be a DB2 for OS/390 and z/OS or Oracle database. In this case, the WebSphere Commerce enterprise bean conversion tool is used to convert the meta-data of the beans from the DB2 format to the DB2 for OS/390 and z/OS or Oracle format. More information about this tool is provided in “Overview of the WebSphere Commerce enterprise bean conversion tool.”

Overview of the WebSphere Commerce enterprise bean conversion tool

In general, there are three scenarios in which the enterprise bean conversion tool is used:

- If the target production environment runs on the iSeries platform
- If the target production database is a DB2 for OS/390 and z/OS database while the development machines use local DB2 databases
- If the target production database is an Oracle database while the development machines use local DB2 databases

This WebSphere Commerce-specific tool allows you to develop against one database type and later deploy to another database type. Using this tool, the meta-data for the enterprise beans is converted to the appropriate format and information for the target database, and the deployed code is also generated using this new meta-data. For step-by-step details on how to use this tool, refer to “Creating an EJB JAR file with conversion” on page 204.

Payment options within the development environment

In previous versions of the WebSphere Commerce development environment, a test payment method was provided to allow developers to complete a purchase within the test environment store, without calling out to a remote payment provider. Now, with the WebSphere Commerce development environment Version 5.5, the WebSphere Commerce Payments component can run within the test environment.

This means that you now have the option of using either the local WebSphere Commerce Payments instance, or you can configure your WebSphere Commerce development instance to use a remote WebSphere Commerce Payments instance.

By default, the local WebSphere Commerce Payments instance is created when you install the WebSphere Commerce development environment. Additionally, if this instance is running at the time you publish a sample store, the store is automatically configured to use that local WebSphere Commerce Payments instance.

Information about configuring the payments options is provided in the *WebSphere Commerce development environment Installation Guide*.

Chapter 9. Deployment details

After you have created customized code in the WebSphere Commerce development environment and have tested it within the WebSphere test environment, you must deploy it to a target WebSphere Commerce Server running outside of the WebSphere test environment. This target WebSphere Commerce Server can run locally on your development machine, or it can be on another machine (using the same, or a different operating system).

This chapter describes the steps required for deployment of customized code to a target WebSphere Commerce Server running outside of the WebSphere test environment.

This chapter is divided into sections that describe how to deploy the various types of code that your customized application may contain. The following tasks are described:

- Deploying enterprise beans
- Deploying commands and data beans
- Deploying store assets
- Updating the target database

Each of the preceding tasks are described in more detail in subsequent sections.

User permission requirements for deployment steps

AIX Linux Solaris You should perform all of the deployment steps (except for the access control updates) on the target WebSphere Commerce Server using the non-root user ID that you created in preparation for the WebSphere Commerce installation process. In addition, ensure that your file assets (for example, JAR files) and directories into which these assets are placed have read, write and execute file permissions granted for this user.

For information about user permissions that are required for performing access control updates, refer to the “Loading your XML changes into the database” topic in the *WebSphere Commerce Security Guide*.

Incremental deployment

The type of deployment described in this chapter is an *incremental* deployment. In an incremental deployment, you must already have a WebSphere Commerce enterprise application installed on the target WebSphere Commerce Server. Then in the deployment process, you only deploy those assets (commands, data beans, enterprise beans and more) to the existing enterprise application.

To create the initial enterprise application on your target WebSphere Commerce Server, you must install WebSphere Commerce and then use the Configuration Manager to create your WebSphere Commerce instance (and hence, create the WebSphere Commerce enterprise application).

Deploying enterprise beans

This section describes how to deploy enterprise beans. These beans may be either new enterprise beans that you have created for your e-commerce application, or they may be WebSphere Commerce entity beans that you have modified. In either case, the deployment steps are basically the same.

One of the key factors to understanding the deployment process for the WebSphere Commerce application is understanding the packaging scheme that is used for customized WebSphere Commerce code. In particular, you do not need to create a new EJB project within the WebSphere Commerce workspace. New enterprise beans are placed into the WebSphereCommerceServerExtensionsData project and the customized code for modified WebSphere Commerce entity beans remains in the original WebSphere Commerce EJB project.

There are two main steps in this deployment process:
- Creating the EJB JAR file
- Updating the EJB JAR file in the target WebSphere Commerce Server

Creating the EJB JAR file

There are two different approaches to creating the EJB JAR file, depending upon your deployment scenario, as follows:
- If you are creating an EJB JAR file that is being deployed to a target WebSphere Commerce Server that uses the same type of database as your development environment, follow the instructions contained in “Creating an EJB JAR file without conversion” on page 203.
- If you are creating an EJB JAR file that is being deployed to target WebSphere Commerce Server that uses a different type of database from your development environment, follow the instructions contained in “Creating an EJB JAR file with conversion” on page 204.

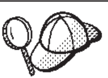
Creating an EJB JAR file without conversion

To create the EJB JAR file, do the following:

1. Open the WebSphere Commerce development environment (**Start > Programs > IBM WebSphere Commerce development environment > WebSphere Commerce development environment**) and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the EJB project that contains the bean or beans that you are deploying, as follows:
   - If you have created new enterprise beans, expand the **WebSphereCommerceServerExtensionsData** EJB project.
   - If you have modified WebSphere Commerce entity beans, expand the project that contains the modified bean. For example, if you have modified the User bean, expand the **Member-MemberManagementData** EJB project.
3. Double-click **EJB Deployment Descriptor**.
4. With the Overview tab selected, scroll to the bottom of the pane, to locate the **WebSphere Bindings** section.
5. In the **DataSource JNDI** name field, enter the data source JNDI name of the target WebSphere Commerce Server. The following is an example value:

   DB2 `jdbc/WebSphere Commerce DB2 DataSource demo`
   where the target WebSphere Commerce Server is using a DB2 database, and the WebSphere Commerce instance name is "demo"

   Oracle `jdbc/WebSphere Commerce Oracle DataSource demo`
   where the target WebSphere Commerce Server is using an Oracle database, and the WebSphere Commerce instance name is "demo".



The value for the DataSource JNDI name is created by adding "`jdbc/`" to the data source name of the target WebSphere Commerce Server. You can verify the data source name by opening the *instanceName*.xml file on the target WebSphere Commerce Server and searching for `DatasourceName=` in the file.

6. Save your deployment descriptor changes (Ctrl+S).
7. In the Business Professional J2EE Navigator view Express Project Navigator view, right-click the EJB project (either WebSphereCommerceServerExtensionsData or the project that contains the modified WebSphere Commerce entity bean) and select **Export**.
   The Export wizard opens.
8. In the Export wizard, do the following:
   a. Select **EJB JAR file** and click **Next**.

b. Business Professional The value for **What resources do you want to export?** is prepopulated with the name of the EJB project.

   Express The EJB project name is prepopulate.
   Leave this value as is.

c. Business Professional In the **Where do you want to export resources to?** field, enter the fully-qualified JAR file name to use.

   Express For the destination, enter the fully-qualified JAR file name to use.
   For example, enter `C:\ExportTemp\`*`JarFileName`*`.jar` where *JarFileName* is the name of your JAR file. If you have created new enterprise beans, you should enter *`yourDir`*`\WebSphereCommerceServerExtensionsData.jar`. If you have modified an existing WebSphere Commerce public entity bean, you must use the predefined JAR file name for this EJB group. For example, if your modification was in the Member-MemberManagementData EJB module, enter *`yourDir`*`\Member-MemberManagementData.jar`.

d. Ensure that **Export source files** is not selected.

e. Click **Finish**.

9. After the JAR file has been created, open the EJB deployment descriptor and restore the modifications that were made in step 5, back to the setting that is required for your local test server. Save your changes.

Creating an EJB JAR file with conversion

To convert the meta-data and create the EJB JAR file, do the following:

1. Open the WebSphere Commerce development environment (**Start > Programs > IBM WebSphere Commerce development environment > WebSphere Commerce development environment**) and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the EJB project that contains the bean or beans that you are deploying, as follows:
   - If you have created new enterprise beans, expand the **WebSphereCommerceServerExtensionsData** project.
   - If you have modified WebSphere Commerce entity beans, expand the project that contains the modified bean. For example, if you have modified the User bean, expand the **Member-MemberManagementData** EJB project.
3. Double-click **EJB Deployment Descriptor**.
4. With the Overview tab selected, scroll to the bottom of the pane, to locate the **WebSphere Bindings** section.
5. In the **DataSource JNDI** name field, enter the data source JNDI name of the target WebSphere Commerce Server. The following is an example value:

DB2 `jdbc/WebSphere Commerce DB2 DataSource demo`
where the target WebSphere Commerce Server uses a DB2 database and the WebSphere Commerce instance name is `demo`

Oracle `jdbc/WebSphere Commerce Oracle DataSource demo`
where the target WebSphere Commerce Server uses an Oracle database and the WebSphere Commerce instance name is `demo`

The value for the DataSource JNDI name is created by adding "`jdbc/`" to the data source name of the target WebSphere Commerce Server. You can verify the data source name by opening the *instanceName*.xml file on the target WebSphere Commerce Server and searching for `DatasourceName=` in the file.

---

6. Save your deployment descriptor changes (Ctrl+S).
7. Close WebSphere Studio Application Developer.
8. At a command prompt, navigate to the following directory:

   WCStudio_installdir\Commerce\bin
9. Enter the following command:

```
ejbDeploy.bat projName outputJarName mapFile workspace_dir eclipseDir
   ejbDeployXmlFile WCStudio_commercedir
```

```
ejbDeploy.bat projName outputJarName mapFile workspace_dir eclipseDir
   ejbDeployXmlFile WCStudio_commercedir zOS
```

where:

- *projName* is the name of the EJB project that is to be converted
- *outputJarName* is the fully-qualified name of the output JAR file. Note that you should be using the name of the JAR file that already exists in the WebSphere Commerce enterprise application. The following is an example:

  `C:\ExportTemp\WebSphereCommerceServerExtensionsData.jar`
- *mapFile* is the name of the mapping file. Examples of this are the following files:

```
WCDE_installdir\Commerce\properties\com\ibm\commerce\
   metadata\conversion\oracle.mapping
WCDE_installdir\Commerce\properties\com\ibm\commerce\
   metadata\conversion\as400.mapping
WCDE_installdir\Commerce\properties\com\ibm\commerce\
   metadata\conversion\zSeries.mapping
```

- *workspace_dir* is the directory for your current development workspace.
- *eclipseDir* is the path to the eclipse directory. By default, this is

  WCDE_installdir\Studio5

- *ejbDeployXmlFile* is the fully-qualified ejbDeploy.xml file. By default, this is

  ```
  WCDE_installdir\Commerce\xml\ejbDeploy.xml
  ```

  Note: When you run this command, you may see errors indicating that it was unable to expand some files. This is not a concern.
- *WCStudio_commercedir* is the path to the Commerce directory that is created when the WebSphere Commerce development environment is installed. By default, this is

  ```
  WCDE_installdir\Commerce
  ```

  or more specifically

  ```
  C:\WebSphere\CommerceDev55
  ```

The following is an example usage of this command with all values specified:

```
ejbDeploy.bat WebSphereCommerceServerExtensionsData
WebSphereCommerceServerExtensionsData.jar
C:\WebSphere\CommerceStudio55\Commerce\properties\com\ibm\
   commerce\metadata\conversion\oracle.mapping
C:\WebSphere\workspace_db2 C:\WebSphere\Studio5
C:\WebSphere\CommerceStudio55\Commerce\xml\ejbDeploy.xml
C:\WebSphere\CommerceStudio55\Commerce
```

Note that line breaks are for presentation purposes only.

10. Open the WebSphere Commerce development environment and open the EJB deployment descriptor and restore the modifications that were made in step 5, back to the setting that is required for your local test server. Save your changes.
11. If you are collecting all assets to be deployed into a single directory (for example C:\ExportTemp), copy the newly created EJB JAR file into that directory.

Updating the EJB JAR file on the target WebSphere Commerce Server

The next step is to copy the newly created EJB JAR file into the appropriate place on the target WebSphere Commerce Server.

To update the EJB JAR file on the target WebSphere Commerce Server, do the following:

1. Stop the WebSphere Commerce instance that is running within WebSphere Application Server. Refer to the *WebSphere Commerce Installation Guide* for your platform and database for details about how to stop this instance.
2. Locate the original EJB JAR file in the WebSphere Commerce instance. For example, locate the following file:

- 400 *WAS_userdir*/installedApps/*WAS_node_name*/ WC_*instance_name*.ear/*JarFileName*.jar
- AIX Linux Solaris *WAS_installdir*/installedApps/*cellName*/ WC_*instance_name*.ear/*JarFileName*.jar
- Windows *WAS_installdir*\installedApps*cellName*\ WC_*instance_name*.ear*JarFileName*.jar

where
- *instance_name* is the name of your WebSphere Commerce instance.
- 400 *WAS_node_name* represents the iSeries system where the WebSphere Application Server product is installed.
- *JarFileName* is the name of the JAR file containing the customized code

3. Make a copy of the original EJB JAR file.
4. Copy the new EJB JAR file from the development machine into the location from step 2.
5. If you have modified EJB deployment descriptors, do the following:
   a. Locate the deployment repository (META-INF directory) for this WebSphere Application Server cell. This typically takes the following form:
      - 400 *WAS_userdir*/config/cells/*WAS_node_name*/applications/ WC_*instance_name*.ear/deployments/WC_*instance_name*/ *EJBModuleName*.jar/META-INF
      - AIX Linux Solaris *WAS_installdir*/config/ cells/*cellName*/applications/WC_*instance_name*.ear/ deployments/WC_*instance_name*/*EJBModuleName*.jar/META-INF
      - Windows *WAS_installdir*\config\cells*cellName* \applications\WC_*instance_name*.ear\deployments\ WC_*instance_name**EJBModuleName*.jar\META-INF

      where
      - *instance_name* is the name of your WebSphere Commerce instance.
      - 400 *WAS_node_name* represents the iSeries system where the WebSphere Application Server product is installed.
      - *EJBModuleName* is the name of the EJB module that has been modified

      The following are platform specific examples of the META-INF directory:
      - 400 *WAS_userdir*/config/cells/myNode/applications/ WC_demo.ear/deployments/WC_demo/Member-MemberManagementData.jar/META-INF

- AIX Linux Solaris *WAS_installdir*/config/cells/myCell/applications/WC_demo.ear/deployments/WC_demo/Member-MemberManagementData.jar/META-INF
- Windows *WAS_installdir*\config\cells\myCell\applications\WC_demo.ear\deployments\WC_demo\Member-MemberManagementData.jar\META-INF

where

- myCell is the name of the WebSphere Application Server cell
- demo is the name of the WebSphere Commerce instance
- Member-MemberManagementData is the name of the EJB module that has been modified
- 400 myNode is the name of the WebSphere Application Server node

b. Backup all of the files in the preceding directory.

c. Use a tool to open the *JarFileName*.jar file and view its contents.

d. Extract the contents of the META-INF directory from the *JarFileName*.jar file into the directory from step 5a.

6. Restart the WebSphere Commerce instance as described in the *WebSphere Commerce Installation Guide* for your platform and database.

Deploying commands and data beans

When you do any of the following tasks, the customized code should be packaged into the WebSphereCommerceServerExtensionsLogic project:

- Create new commands
- Create new data beans
- Modify existing WebSphere Commerce commands
- Modify existing WebSphere Commerce data beans

You are not permitted to make modifications directly into an existing class (or into the project containing the existing class) for commands or data beans.

Deploying customized commands and data beans involves the following steps:

- Creating the JAR file
- Updating the JAR file in the target WebSphere Commerce Server

The preceding steps are described in more detail in subsequent sections.

Ensure that you refer to "Updating the target database" on page 212 if your customized code requires updates to the command registry, new access control policies, or other database updates.

Creating the JAR file

To create the JAR file, do the following:

1. Open the WebSphere Commerce development environment (**Start > Programs > IBM WebSphere Commerce development environment > WebSphere Commerce development environment**) and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Right-click the **WebSphereCommerceServerExtensionsLogic** project and select **Export**.
   The Export wizard opens.
3. In the Export wizard, do the following:
   a. Select **JAR file** and click **Next**.
   b. The left pane under **Select the resources to export?** is prepopulated with the name of the project. Leave this value as is.
   c. In the right pane under **Select the resources to export?** ensure that only the following resources are selected:
      - .classpath
      - .project
      - .serverPreference
   d. Ensure that **Export generated class files and resources** is selected.
   e. Do *not* select **Export Java source files and resources**.
   f. In the **Select the export destination** field, enter the fully-qualified JAR file name to use. For example, enter `C:\ExportTemp\WebSphereCommerceServerExtensionsLogic.jar`.
      Note that the JAR file name must be WebSphereCommerceServerExtensionsLogic.jar.
   g. Click **Finish**.

Once the JAR file has been successfully created, the next step is to transfer the file into the appropriate location on your target WebSphere Commerce Server.

Updating the JAR file on the target WebSphere Commerce Server

The next step is to copy the newly created JAR file into the appropriate place on the target WebSphere Commerce Server.

To update the JAR file, do the following:

1. Stop the WebSphere Commerce instance that is running within WebSphere Application Server. Refer to the *WebSphere Commerce Installation Guide* for your platform and database for details about how to stop this instance.
2. Locate the original EJB JAR file in the WebSphere Commerce instance. For example, locate the following file:
   - 400 *WAS_userdir*/installedApps/*WAS_node_name*/WC_*instance_name*.ear/WebSphereCommerceServerExtensionsLogic.jar

- AIX Linux Solaris *WAS_installdir*/installedApps/*cellName*/ WC_*instance_name*.ear/WebSphereCommerceServerExtensionsLogic.jar
- Windows *WAS_installdir*\installedApps*cellName*\ WC_*instance_name*.ear\WebSphereCommerceServerExtensionsLogic.jar

where

- *instance_name* is the name of your WebSphere Commerce instance.
- 400 *WAS_node_name* represents the iSeries system where the WebSphere Application Server product is installed.

3. Make a copy of the original JAR file to a backup location.
4. Copy the JAR file from the development machine into the location from step 2.
5. Restart the WebSphere Commerce instance as described in the *WebSphere Commerce Installation Guide* for your platform and database.

Deploying store assets

Store assets include assets such as the following:

- JSP templates
- HTML files
- Image files
- XML files
- Properties files and resource bundles

These assets must be deployed from your development environment onto your target WebSphere Commerce Server. This includes the following steps:

- Exporting store assets from WebSphere Studio Application Developer
- Transferring assets to the target WebSphere Commerce Server

The preceding steps are described in more detail in subsequent sections.

Exporting store assets

To export the store assets from the development environment, do the following:

1. Open the WebSphere Commerce development environment (**Start > Programs > IBM WebSphere Commerce development environment > WebSphere Commerce development environment**) and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the **Stores** folder.
3. Right-click the **Web Content** folder and select **Export**.
   The Export Wizard opens.
4. In the Export wizard, do the following:

a. Select **File system** and click **Next**.
b. Select all of the resources that you want to deploy. That is, select all of the JSP templates, HTML files, images, property files and other store assets that need to be deployed.
c. Select **Create directory structure for selected files**.
d. In the **Directory** field, enter a temporary directory into which these resources will be placed. For example, enter `C:\ExportTemp\StoreAssets`
e. Click **Finish**.

The next step is to copy these resources into the appropriate location on your target WebSphere Commerce Server.

Transferring store assets

To transfer store assets from your development machine to your target WebSphere Commerce Server, do the following:

1. Depending upon the types of assets you are deploying and your specific configuration details, you may need to stop the WebSphere Commerce instance that is running within WebSphere Application Server. If you are uncertain about whether a restart will be required for your particular deployment scenario, you should stop the instance. Refer to the*WebSphere Commerce Installation Guide* for your platform and database for details about how to stop this instance.
2. On the target machine, locate the Stores.war directory. The following is an example of this directory:
   - 400 *WAS_userdir*/installedApps/*WAS_node_name*/ WC_*instance_name*.ear/Stores.war
   - AIX Linux Solaris *WAS_installdir*/installedApps/*cellName*/ WC_*instance_name*.ear/Stores.war
   - Windows *WAS_installdir*\installedApps*cellName*\ WC_*instance_name*.ear\Stores.war

   where
   - *instance_name* is the name of your WebSphere Commerce instance.
   - 400 *WAS_node_name* represents the iSeries system where the WebSphere Application Server product is installed.
3. Copy the files exported in "Exporting store assets" into the Stores.war directory.
4. If you previously stopped your WebSphere Commerce instance, start the instance. Refer to the*WebSphere Commerce Installation Guide* for your platform and database for details about how to start this instance.

Updating the target database

As your target WebSphere Commerce Server uses a different database than your development machine, you must perform all updates that were done to the development database on the database that is used by the target WebSphere Commerce Server. This includes any updates for the registration of new or modified commands or views, additional tables that have been created, and the creation of access control policies for any new resources that have been created.

400 You are responsible for having a utility for executing SQL statements. One way to do this is to use IBM iSeries Access for Windows. To open this utility, do the following:

1. Open the **iSeries Navigator**.
2. After the operations navigator opens, you are required to sign onto a particular system. Ensure that you select the target iSeries machine and use the WebSphere Commerce instance user profile and password. This ensures that the WebSphere Commerce instance user profile owns all new tables that are created.
3. In the panel on the left, expand your iSeries system and then **DATABASES**. Right-click on the Relational Database and select **Run SQL Scripts** from the drop-down list list.
   The Run SQL Scripts window opens. Using this window, you can cut and paste in SQL statements or open an SQL script. You can set your default schema by using the **JDBC Setup** option under the **Connection** selection.

* DB2 390 z/OS To log on to the z/OS system in order to perform
* operations on the target database, such as checking the table space status or
* executing SQL statements using SPUFI, you may use the IBM Personal
* Communications package. To utilize this package, do the following:

* 1. Install IBM Personal Communications and configure it to connect to z/OS.
* 2. Log on to TSO using the user ID provided by your z/OS system
* administrator.
* 3. Perform the operations.

Access control updates

While access control information is contained in the database, this is a special type of information that may not necessarily be a straight replication from the development environment to the target environment. In particular, in the development environment, you may decide to use very liberal access control policies that are not appropriate for a production (or next level of testing) environment. As an example, within the confines of the development environment, policies for new commands could be set such that all users can execute the command, but this may not be appropriate elsewhere.

As a result, before copying access control information from the development to target environment, you should give consideration to the access control requirements in the new environment and adjust your policies accordingly.

For information about loading access control policies (including command syntax for various platforms and directory permission requirements), refer to the *WebSphere Commerce Security Guide*.

Part 4. Tutorials

The following tutorials are designed to introduce the various tasks related to creating customized code for WebSphere Commerce applications. The development-related steps are to be performed in either a WebSphere Commerce Business Edition, a WebSphere Commerce - Express development environment, or a WebSphere Commerce Studio, Professional Developer Edition. The deployment steps are to be performed on WebSphere Commerce (either Business, Express, or Professional edition, corresponding to your development environment) running on Windows 2000.

© Copyright IBM Corp. 2000, 2003

Chapter 10. Tutorial: Creating new business logic

This tutorial is designed to show you the steps involved in creating new business logic. The types of assets created include a new view, a new controller command, a new task command, new data beans, and a new entity bean. This tutorial uses the scenario of developing a small interface to let users modify a balance of bonus points. It is for demonstration purposes only and does not reflect the logic required to build a loyalty program application. Instead, from this tutorial, you will learn the development steps that are common to creating each of the previously listed types of code assets.

This tutorial is divided into the following sub-tasks:

1. Preparing your workspace
2. Creating a new view
3. Creating a new controller command
4. Passing information from the controller command to the view
5. Parsing and validating URL parameters in the controller command
6. Creating a new task command
7. Modifying the new task command
8. Creating a new enterprise bean:
   a. Creating a new table
   b. Creating a new CMP enterprise bean
   c. Mapping the new bean to the table and creating the schema
   d. Creating an associated access bean
   e. Generating deployed code
   f. Testing the bean with the universal test client
9. Integrating the Bonus bean into MyNewControllerCmd
   a. Modifying the performExecute method of the MyNewTaskCmdImpl class to calculate the new bonus points and save the points to the XBONUS table.
   b. Adding a getResources method to the MyNewControllerCmdImpl class to return a list of resources that the command uses. This method is included for access control purposes.
   c. Creating the BonusDataBean so that bonus points can be easily displayed on a JSP template.
   d. Creating a new access control policy for the new resources.
   e. Modifying the MyNewJSPTemplate.jsp file to allow users to enter bonus points and display results.

© Copyright IBM Corp. 2000, 2003

f. Testing the integrated code.
10. Deploying all of the preceding code, access control policies, JSP templates, images, and resource bundles to a target WebSphere Commerce Server running in WebSphere Application Server.

Each of the preceding steps is described in step-by-step details in subsequent sections.

Locating the sample code

Before beginning this tutorial, download the WC_SAMPLE_55.zip package that contains the starting point for these programming tutorials. Save this file onto your development machine. As an example, you might save the file into the *WCDE_installdir* directory:

This package is located with the *WebSphere Commerce Programming Guide and Tutorials* on the following Web site:

http://www.ibm.com/software/commerce/library/

Preparing your workspace

In this step, you import sample code into your WebSphere Commerce workspace. This sample code is the starting point for the tutorial.

To prepare your workspace, do the following:

1. Ensure that you have installed the WebSphere Commerce development environment Version 5.5 and that you have completed configuring your development environment. You should have also published a store based upon the FashionFlow sample store (which is an example of the consumer direct model) within your development environment. Instructions on how to publish a store are found in the WebSphere Commerce Production and Development online help.
2. Import the sample code into your workspace, by doing the following:
   a. Start WebSphere Commerce development environment as follows:
      - Business Professional **Start > Programs > IBM WebSphere Commerce Studio > WebSphere Commerce development environment**
      - Express Express **Start > Programs > IBM WebSphere - Express Developer Edition > WebSphere Commerce development environment**
   b. Switch to the J2EE Perspective (**Window > Open Perspective > J2EE**) and then select the Business Professional J2EE Navigator view Express Project Navigator view.

c. Expand the **WebSphereCommerceServerExtensionsLogic** project.

d. Right-click the **src** folder and select **Import**.
The Import wizard opens.

e. From the **Select an import source** list, select **Zip file** and click **Next**.

f. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located as follows:
yourDirectory\WC_SAMPLE_55.zip
where *yourDirectory* is the directory into which you downloaded the package.

g. Click **Deselect All**, then expand the directories and select to import the following files:
   - com\ibm\commerce\sample\commands\MyNewControllerCmd.java
   - com\ibm\commerce\sample\commands\MyNewControllerCmdImpl.java
   - com\ibm\commerce\sample\commands\MyNewTaskCmd.java
   - com\ibm\commerce\sample\commands\MyNewTaskCmdImpl.java
   - com\ibm\commerce\sample\databeans\MyNewDataBean.java

h. In the **Folder** field, the `WebSphereCommerceServerExtensionsLogic/src` folder is already specified. Keep this value.

i. Click **Finish**.

3. Right-click the **WebSphereCommerceServerExtensionsLogic** project and select **Rebuild project**.

4. Import the JSP template for the tutorials into the appropriate directory, by doing the following:

a. In the Business Professional J2EE Navigator view Express Project Navigator view, expand the **Stores** project. Then expand **Web Content > *FashionFlow_name*** where *FashionFlow_name* is the name of your store based upon the FashionFlow sample store.

If you just published your store, the *FashionFlow_name* directory may not be displayed. If this is the case, right-click the Stores project and select **Refresh**. The *FashionFlow_name* directory will now be available in the workspace.

b. Right-click the ***FashionFlow_name*** directory and select **Import**.
The Import wizard opens.

c. From the **Select an import source** list, select **Zip file** and click **Next**.

d. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located as follows:

yourDirectory\WC_SAMPLE_55.zip
where *yourDirectory* is the directory into which you downloaded the package.

e. Click **Deselect All**, then select the MyNewJSPTemplate_All.jsp file.

f. In the **Folder** field, the `Stores/Web Content/`*`FashionFlow_name`* folder is already specified. Keep this value.

g. Click **Finish**.

5. Import the properties file for the tutorial into the appropriate directory, by doing the following:

   a. In the Business Professional J2EE Navigator view Express Project Navigator view, expand the following directories:
   Stores > Web Content> WEB-INF> classes > ***FashionFlow_name*** directory.

   b. Right-click the ***FashionFlow_name*** directory and select **Import**.
   The Import wizard opens.

   c. From the **Select an import source** list, select **Zip file** and click **Next**.

   d. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located, as follows:
   yourDirectory\WC_SAMPLE_55.zip
   where *yourDirectory* is the directory into which you downloaded the package.

   e. Click **Deselect All**, then select the Tutorial_All_en_US.properties file.

   f. In the **Folder** field, the `Stores/Web Content/WEB-INF/classes/`*`FashionFlow_name`* folder is already specified. Keep this value.

   g. Click **Finish**.

6. There are four files that are used to load access control policies for new resources that are created in the tutorials. These are:

   - MyNewViewACPolicy.xml — This XML file contains the access control policy used when a new view is created.
   - MyNewControllerCmdACPolicy.xml— This XML file contains the access control policy used when a new controller command is created.
   - SampleACPolicy_template.xml— This XML file contains the access control policy used when a new enterprise bean is created.
   - SampleACPolicy_template_en_US.xml— This XML file contains the access control policy description when a new enterprise bean is created.

   Copy the preceding files into the appropriate directory, by doing the following:

   a. On the file system, navigate to the following directory:
   yourDirectory\WC_SAMPLE_55.zip.

b. Expand the ZIP file and extract the preceding four files into the following directory:
   WCDE_installdir\Commerce\xml\policies\xml
7. Test your environment to ensure you are ready to start the tutorials, by doing the following:
   a. In WebSphere Studio Application Developer, switch to the Server perspective.
   b. Start your payment server. If you are running a local payment server, right-click **WebSphereCommercePaymentsServer** and select **Start** (or **Restart**).
   c. Right-click **WebSphereCommerceServer** and select **Start** (or **Restart**).
   d. Watch the console to see when the WebSphereCommerceServer server has finished its startup process. You will see information similar to the following when the server has started:

```
[4/2/03 12:56:06:286 EST] 66adf8d1 ApplicationMg A WSVR0221I:
   Application started: WebSphereCommerceServer
[4/2/03 12:56:06:777 EST] 66adf8d1 HttpTransport A SRVE0171I:
   Transport http is listening on port 9,080.
[4/2/03 12:56:10:742 EST] 66adf8d1 HttpTransport A SRVE0171I:
   Transport https is listening on port 9,443.
[4/2/03 12:56:10:762 EST] 66adf8d1 HttpTransport A SRVE0171I:
   Transport http is listening on port 8,080.
[4/2/03 12:56:10:762 EST] 66adf8d1 HttpTransport A SRVE0171I:
   Transport http is listening on port 80.
[4/2/03 12:56:11:123 EST] 66adf8d1 HttpTransport A SRVE0171I:
   Transport https is listening on port 443.
[4/2/03 12:56:11:183 EST] 66adf8d1 HttpTransport A SRVE0171I:
   Transport https is listening on port 9,043.
[4/2/03 12:56:11:653 EST] 66adf8d1 RMIConnectorC A ADMC0026I:
   RMI Connector available at port 2809
[4/2/03 12:56:12:575 EST] 66adf8d1 WsServer
   A WSVR0001I: Server server1 open for e-business
```

   e. In the Business Professional J2EE Navigator view Express Project Navigator view, expand the **Stores** project. Then expand **Web Content > *FashionFlow_name***.
   f. Right-click the **index.jsp** file and select **Run on Server**.
      The sample store opens.
   g. Select a product and ensure that you can purchase it.

You are now ready to proceed with the tutorials.

Creating a new view

The first step of this tutorial is to create a new view. This new view is called MyNewView and has a corresponding JSP template, MyNewJSPTemplate.jsp.

In this section of the tutorial, you will learn the following:
- Where to place JSP templates and graphic files that apply to a specific store.
- How to use WebSphere Studio Application Developer to create the JSP template.
- How to create the properties file that contains the text for the JSP template.
- How to update the view registry (VIEWREG table) with the new "MyNewView"
- How to set up access control for the new view.
- How to use the WebSphere test environment (WTE) to test the new view.

In general, creating a new view includes the following steps:
1. Naming the view and registering it in the view registry.
2. Creating new properties files in which translatable text for JSP templates are stored.
3. Creating a new JSP template for the new view.
4. Creating and loading access control policies for the view.

Registering MyNewView

In this scenario, the view you create is called MyNewView. This view must be registered in the VIEWREG table, which is part of the command registry. To register a new view, you only need to use a simple SQL statement to create a new entry in the VIEWREG table.

Before proceeding with this step, you must know the unique identifier for your store. You can determine this by running the following SQL query against your development database:

```
select STOREENT_ID from STOREENT where IDENTIFIER = 'FashionFlow_name'
```

where *FashionFlow_name* is the name of your store. Make note of the value here, as it is required in the next section:

____

DB2 If you are using a DB2 database, do the following to register MyNewView:
1. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Line Tools > Command Center**).
2. From the **Tools** menu, select **Tools Settings**.
3. Select the **Use statement termination character** check box and ensure the character specified is a semicolon (**;**)
4. Close the tools settings.

5. With the Script tab selected, create the required entry in the VIEWREG table, by entering the following information in the script window:

```
connect to developmentDB user dbuser using dbpassword;
insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID, INTERFACENAME,
   CLASSNAME, PROPERTIES, DESCRIPTION, HTTPS, LASTUPDATE)
values ('MyNewView', -1, FF_storeent_ID,
'com.ibm.commerce.command.ForwardViewCommand',
'com.ibm.commerce.command.HttpForwardViewCommandImpl',
'docname=MyNewJSPTemplate.jsp', 'This is my new view for tutorial one',
0, null)
```

where

- *developmentDB* is the name of your development database
- *dbuser* is the database user
- *dbpassword* is the password for your database user
- *FF_storeent_ID* is the unique identifier for your store that is based on the FashionFlow sample store.

Click the **Execute** icon.

You should see a message indicating that the SQL command completed successfully.

Oracle If you are using an Oracle database, do the following to register your view in the database:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

```
insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID, INTERFACENAME,
   CLASSNAME, PROPERTIES, DESCRIPTION, HTTPS, LASTUPDATE)
values ('MyNewView',-1, FF_storeent_ID,
   'com.ibm.commerce.command.ForwardViewCommand',
   'com.ibm.commerce.command.HttpForwardViewCommandImpl',
   'docname=MyNewJSPTemplate.jsp','This is my new view for tutorial 1',
   0, null);
```

where

- `FF_storeent_ID` is the unique identifier for your store that is based on the FashionFlow sample store.

Press Enter to run the SQL statement.

6. Enter the following to commit your database changes:

```
commit;
```

and press Enter to run the SQL statement.

MyNewView is now registered.

Creating a properties file for the tutorial

In this step, you create the new properties file to hold any translatable text that is used in the JSP template. Separating translatable text from the JSP template itself makes the task of translation much simpler and is also key to having a globalized Web site.

To create the properties file, do the following in WebSphere Studio Application Developer:

1. Open the Web perspective (**Window > Open Perspective > Web**).
2. Within the **Stores** Web project, expand the **Web Content > WEB-INF > classes >** ***FashionFlow_name*** folders. You will find the Tutorial_All_en_US.properties file.
3. Right-click **Tutorial_All_en_US.properties** and select **Open With > Properties File Editor**.
4. Right-click the ***FashionFlow_name*** folder and select **New > Other > Simple > File > Next** to create a new properties file.
   The New File window opens.
5. In the **File name** field, enter `TutorialNLS_en_US.properties`, then click **Finish**.
   The new empty file opens.
6. Copy section 1 from the Tutorial_All_en_US.properties file into the new TutorialNLS_en_US.properties file. This introduces the following name-value pairs into the TutorialNLS_en_US.properties file:

```
# -- SECTION 1 -- #

ProgrammerGuide=Programmer's Guide
Tutorial=Tutorial: Creating new business logic
ParametersFromCmd= List of parameter-value pairs sent from the
   controller command
CalledByControllerCmd=MyNewView was called by a controller command
CalledByWhichControllerCmd=MyNewView was called by the controller
   command which is -
ControllerParm1=ControllerParm1=
ControllerParm2=ControllerParm2=
Example=This is an example of using the <if> tag from JSP Standard
   Tag Library (JSTL)
UserName=UserName=
Points=Points=
Greeting=Greeting=
UserId=UserId=
FirstInput=Your first input parameter
RegisteredUser=is a registered user
ReferenceNumber=The member refernce number of this user is
NotRegisteredUser=is not a registered user
```

```
BonusAdmin=Bonus Administration
PointBeforeUpdate=The bonus point before update is
PointAfterUpdate=The bonus point after update is
EnterPoint=Please enter the points, then submit it to the controller
   command

# -- END OF SECTION 1 -- #
```

Note that line breaks within a value are for presenation purposes only.

7. Save the TutorialNLS_en_US.properties file (Ctrl+S).

The new TutorialNLS_en_US.properties file is saved under the Stores\Web Content\WEB-INF\classes*FashionFlow_name* directory and will be used as a resource bundle by your new JSP template.

Note: When the content of a properties file is changed, the server must be restarted before the changes can be seen in testing.

Creating MyNewJSPTemplate

In this step, you use the Page Designer tool in WebSphere Studio to create a new JSP template. In particular, you create MyNewJSPTemplate.jsp that is used with MyNewView. When creating this template, you create a new blank JSP template and then add appropriate sections from another JSP template (MyNewJSPTemplate_All.jsp).

To create your new JSP template, do the following:

1. In the Web perspective, switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Right-click the **Stores** Web project and select **Properties**.
3. Business Professional Select **Web** in the left pane and then from the list of Available Web Project Features, select **Include the JSP Standard Tag Library**.

   Express Select **Web Project Features** in the left pane and then from the list of Available Web Project Features, select **JSP Standard Tag Library**.

   Click **Apply**. When the update is complete, click **OK** to close the properties editor.
4. Expand the **Web Content*FashionFlow_name*** directory.
5. Right-click **MyNewJSPTemplate_All.jsp** file and select **Open With > Page Designer**.
6. Right-click the ***FashionFlow_name*** folder and select **New > JSP File** to create the new JSP template in this folder.
   The New JSP File window opens.
7. Specify values for the new file, as follows:
   a. In the **File name** field, enter `MyNewJSPTemplate.jsp`.

b. From the **Markup Language** drop-down list, select **XHTML** and click **Next**.

c. Business Professional Click **Add Tag Library**.
The Select a Tag Library window opens..

Express Select **Configure advance options** > click **Next** > click **Add** (next to the Tag Libraries table).

d. Select the following tag libraries:

- http://java.sun.com/jstl/core
- http://java.sun.com/jstl/fmt

Click **OK**, then **Next**.

e. Click **Next**.

f. Clear the **(Use workbench default) Workbench Encoding** check box.

g. From the **Encoding** drop-down list, select **ISO Latin -1**.

h. From the **Document Type** drop-down list, select **XHTML 1.0 Transitional**.

i. Click **Finish**.
The MyNewJSPTemplate.jsp file opens. Click the Design, Source, and Preview tabs for different views of the file.

8. With the Design tab selected, click on the **Place MyNewJSPTemplate.jsp's content here** text. Replace this text with `Hello world!`.

9. Switch to the Source and then Preview tabs. Notice that the text has changed.

10. Now you must copy a preparation section from the MyNewJSPTemplate_All.jsp file into your new MyNewJSPTemplate.jsp file. This section sets the place holders for updates you will make to the file. Copy the text between the `<%--PREPARATION SECTION` and `END OF PREPARATION SECTION --%>` markers into your new JSP template. When copying this text into your JSP template, overwrite the following text:

```
<title> MyNewJSPTemplate.jsp </title>
</head>
<body>
<p> Hello World! </p>
</body>
</html>
```

Note: Do not copy the `<%--PREPARATION SECTION` and `END OF PREPARATION SECTION --%>` markers into your new JSP template. Only copy the text contained between those markers.

11. Copy sections 1A and 2 from MyNewJSPTemplate_All.jsp file into your new MyNewJSPTemplate.jsp file. Place the new text between the `<!--`

SECTION 1A -->, <!-- END OF SECTION 1A -->, <!-- SECTION 2 -->, and <!-- END OF SECTION 2 --> markers. This introduces the following text into MyNewJSPTemplate.jsp:

```
<!-- SECTION 1A -->

    <%@ include file="include/EnvironmentSetup.jsp"%>

<!-- END OF SECTION 1A -->

<!-- SECTION 2 -->

<fmt:setLocale value="${CommandContext.locale}" />
<fmt:setBundle basename="${sdb.directory}/TutorialNLS" var="tutorial" />

<!-- END OF SECTION 2-->
```

The first section includes the EnvironmentSetup.jsp file that is used to set up environment variables. The second section is used to create the resource bundle object that is used to retrieve information from the properties file and it sets the locale.

12. Now add a graphic and text to the JSP template. Again, this step is accomplished by copying text from the MyNewJSPTemplate_All.jsp file into the MyNewJSPTemplate.jsp file. This time, copy section 3 from MyNewJSPTemplate_All.jsp file into the MyNewJSPTemplate.jsp file. This introduces the following text into the JSP template:

```
<!-- SECTION 3 -->

<table cellpadding="0" cellspacing="0" border="0">
 <tr>
  <td bgcolor="#ff2d2d" >
   <img src="<c:out value="${sdb.jspStoreDirFilePath}
                           images/male_blueshirt.gif" />" border="0"/>
  </td>
 </tr>
</table>

<h1><fmt:message key="ProgrammerGuide" bundle="${tutorial}" /> </h1>

<h2><fmt:message key="Tutorial" bundle="${tutorial}" /> </h2>

<!-- END OF SECTION 3 -->
```

Section 3 introduces an image that is located in the store-specific image sub-folder (Stores\Web Content*FashionFlow_name*\images). It also retrieves text from the properties file.

13. Save the changes you made to the MyNewJSPTemplate.jsp file (Ctrl+S).

Creating and loading access control policies for MyNewView

Command-level access control must be specified for the new view. In this case, the command-level access control policy specifies that all users are

allowed to execute the view. Note that this type of access control policy is acceptable for the development environment, but it may not be suitable for other circumstances. For more advanced access control requirements, refer to the *WebSphere Commerce Security Guide*.

The access control policy is defined by the MyNewViewACPolicy.xml file, which you placed into the following directory, as part of the preparatory steps:
WCDE_installdir\Commerce\xml\policies\xml

To load the new policy, do the following:

1. At a command prompt, navigate to the following directory: *WCDE_installdir*\Commerce\bin
2. You must issue the acpload command, which has the following form:

   ```
   acpload db_name db_user db_password inputXMLFile
   ```

   where
   - *db_name* is the name of your development database.
   - *db_user* is the name of the database user.
   - *db_password* is the password for your database user.
   - *inputXMLFile* is the XML file containing the access control policy specification. In this case, specify `MyNewViewACPolicy.xml`.

   The following is an example of the command, with variables specified:

   ```
   acpload Demo_Dev db2user db2user MyNewViewACPolicy.xml
   ```

Testing MyNewView

The final step of creating a new view is to test the new view in the WebSphere test environment. Note that when testing the new view (and later when testing the new commands) you must first launch the home page of your store. The reason that launching the store is required is because the new view is registered specifically to your store. As a result, you first launch the store home page to set the store ID value in the command context, before attempting to access the new view. Similarly, the controller command you create later is also registered specifically to your store and requires the store ID from the command context.

To test your new view, do the following:

1. In WebSphere Studio Application Developer, open the Server Perspective (**Window > Open Perspective > Server**).
2. Right-click the **WebSphereCommerceServer** server and select **Start** (or **Restart**).

3. Right-click **index.jsp** under the Stores\Web Content*FashionFlow_name* directory and select **Run on Server**.
   The store home page is displayed in the Web browser.
4. In the Web browser enter the following URL:

   ```
   http://localhost/webapp/wcs/stores/servlet/MyNewView
   ```

   After a few seconds, the new JSP template is displayed, as shown in the following screen shot:

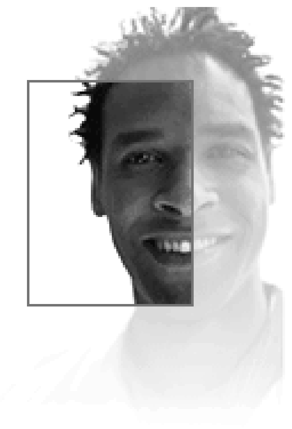


Figure 31.

Creating a new controller command

In this step, you create a new controller command, called MyNewControllerCmd. Initially, this command only returns the MyNewView view.

In this section of the tutorial, you will learn the following:
- The minimum requirements for code contained in a controller command
- How to create the new controller command interface and implementation class
- How to set up a controller command to return a view
- How to register a controller command in the command registry
- How to set up access control for a controller command

In general, creating a new controller command involves the following steps:
1. Registering the new command in the command registry.
2. Creating an interface for the command.
3. Creating an implementation class for the command.
4. Creating and loading access control policies for the command.
5. Testing the command.

Registering MyNewControllerCmd

In this part of the tutorial, you create a new controller command called MyNewControllerCmd. This command must be registered in the command registry. In particular, the interface must be registered in the URLREG table, and the association between the interface and its implementation class gets registered in the CMDREG table.

DB2 If you are using a DB2 database, do the following to register MyNewControllerCmd:
1. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Line Tools > Command Center**).
2. With the Script tab selected, create the required entry in the URLREG table, by entering the following information in the script window:
```
connect to developmentDB user dbuser using dbpassword;
insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS, DESCRIPTION,
   AUTHENTICATED) values ('MyNewControllerCmd',FF_storeent_ID,
   'com.ibm.commerce.sample.commands.MyNewControllerCmd',
   0, 'This is a new controller command for tutorial one.',null);
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION,
   CLASSNAME, TARGET)
values (FF_storeent_ID,
   'com.ibm.commerce.sample.commands.MyNewControllerCmd',
   'This is a new controller command for tutorial one.',
   'com.ibm.commerce.sample.commands.MyNewControllerCmdImpl',
   'local');
```

where

- *developmentDB* is the name of your development database
- *dbuser* is the database user
- *dbpassword* is the password for your database user
- *FF_storeent_ID* is the unique identifier for your store that is based on the FashionFlow sample store.

Click the **Execute** icon.

You should see a message indicating that the SQL command completed successfully.

Oracle If you are using an Oracle database, do the following to register MyNewControllerCmd:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

```
insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS, DESCRIPTION,
   AUTHENTICATED) values ('MyNewControllerCmd',FF_storeent_ID,
   'com.ibm.commerce.sample.commands.MyNewControllerCmd',
   0, 'This is a new controller command for tutorial one.',null);
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION,
   CLASSNAME, TARGET)
values (FF_storeent_ID,
   'com.ibm.commerce.sample.commands.MyNewControllerCmd',
   'This is a new controller command for tutorial one.',
   'com.ibm.commerce.sample.commands.MyNewControllerCmdImpl','local');
```

   where

   - *FF_storeent_ID* is the unique identifier for your store that is based on the FashionFlow sample store.

   Press Enter to run the SQL statement.
6. Enter the following to commit your database changes:

```
commit;
```

   and press Enter to run the SQL statement.

Note that the second insert statement into the CMDREG table is not absolutely necessary. In this scenario, the interface will use the default implementation, and as such, this association between the interface and

implementation class does not really need to be specified in the command registry. It is included here for the purpose of completeness.

Creating the MyNewControllerCmd interface

According to the WebSphere Commerce programming model, all new controller commands must have an interface, as well as an implementation class. For this tutorial, a base for the interface is provided in the sample code. It is split into a number of different sections that are currently in the code as comments. As you progress through the tutorial, you uncomment various sections of the code.

To create the MyNewControllerCmd interface, do the following:

1. In WebSphere Studio Application Developer, open the Java perspective (**Window > Open Perspective > Java**).
2. Expand the **WebSphereCommerceServerExtensionsLogic** project.
3. Navigate to the **src** directory and then expand the **com.ibm.commerce.sample.commands** package.
4. Double-click the **MyNewControllerCmd.java** interface to open the file.
5. In the source, uncomment section 1 (delete the “/*” before the section and the “*/”after the section). This introduces the following code into the interface:

   ```
   /// Section 1 ///////////////////////////////////////////////

   // set default command implement class

    static final String defaultCommandClassName =
        "com.ibm.commerce.sample.commands.MyNewControllerCmdImpl";


   /// End of section 1////////////////////////////////////////
   ```

   This section of code specifies that by default, the interface should use the MyNewControllerCmdImpl implementation class.
6. Save the change to the interface (Ctrl+S).

Creating the MyNewControllerCmdImpl implementation class

Once the interface is created, the next step is to create the implementation class for the command. For this tutorial, a base for the implementation class is provided in the sample code. It is split into a number of different sections that are currently in the code as comments. As you progress through the tutorial, you uncomment various sections of the code.

To create the MyNewControllerCmdImpl implementation class, do the following:

1. Double-click the **MyNewControllerCmdImpl.java** class to open it.

2. In the Outline view, select the **performExecute** method to view its source code.
3. In the source code for the performExecute method, uncomment Section 1. This introduces the following code into the method:

```
/// Section 1 ////////////////////////////////////////////////

    /// create a new TypedProperties for output purpose.

  TypedProperty rspProp = new TypedProperty();

/// End of section 1 /////////////////////////////////////////
```

   This creates a new TypedProperty object that is used to hold the command's response properties.
4. In the source code for the performExecute method, uncomment Section 5. This introduces the following code into the method:

```
/// Section 5 /////////////////////////////////////////////////

    /// see how controller command call a JSP

 rspProp.put(ECConstants.EC_VIEWTASKNAME, "MyNewView");
 setResponseProperties(rspProp);

/// End of section 5//////////////////////////////////////////
```

   This section of code accomplishes two main tasks. First, it is a requirement of the WebSphere Commerce programming model that all controller commands return a view. In this section, it specifies that the view to be returned is the MyNewView, that you previously created. Additionally, it sets the command's response properties to be the new rspProp object.
5. Save your changes (Ctrl+S).
6. In order to compile the changes you have made to your code, right-click the **WebSphereCommerceServerExtensionsLogic** project and select **Build Project**.

Creating and loading access control policies for the command

Command-level access control must be specified for the new command. In this case, the command-level access control policy specifies that all users are allowed to execute the command. Note that this type of access control policy is acceptable for the development environment, but it may not be suitable for other circumstances. For more advanced access control requirements, refer to the *WebSphere Commerce Security Guide*.

The access control policy is defined by the MyNewControllerCmdACPolicy.xml file, which you placed into the following directory, as part of the preparatory steps: *WCDE_installdir*\Commerce\xml\policies\xml

To load the new policy, do the following:

1. At a command prompt, navigate to the following directory: *WCDE_installdir*\Commerce\bin
2. You must issue the acpload command, which has the following form:

   ```
   acpload db_name db_user db_password inputXMLFile
   ```

   where
   - *db_name* is the name of your development database.
   - *db_user* is the name of the database user.
   - *db_password* is the password for your database user.
   - *inputXMLFile* is the XML file containing the access control policy specification. In this case, specify `MyNewControllerCmdACPolicy.xml`.

   The following is an example of the command, with variables specified:

   ```
   acpload Demo_Dev db2user db2user MyNewControllerCmdACPolicy.xml
   ```

Testing MyNewControllerCmd

Now that the interface, implementation class, command registration, and access control information have all been created, you can test the new controller command.

When Java code has been modified, the test server must be restarted before changes are recognized.

To test your new code, do the following:

1. Switch to the Server perspective (**Window > Open Perspective > Server**).
2. Right-click the **WebSphereCommerceServer** server and select **Start** (or **Restart**).
3. Right-click **index.jsp** under the Stores\Web Content*FashionFlow_name* directory and select **Run on Server**.
   The store home page is displayed in the Web browser.
4. In the Web browser enter the following URL:

   ```
   http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd
   ```

   After a few seconds, the new JSP template is displayed, as shown in the following screen shot:

Figure 32.

Passing information from MyNewControllerCmd to MyNewView

In this step, you modify MyNewControllerCmd so that it passes information to MyNewView. Two different ways of passing information to the view are shown. First, you will learn how to use the TypedProperties object for response properties and how to extract information from this object onto the JSP template. Second, you will learn how to create a new data bean that is used to pass information to the JSP template.

Passing information using a TypedProperties object

In this section, you modify MyNewControllerCmdImpl to pass information to the JSP template. In particular, you modify the command to add additional name-value pairs into the existing rspProp TypedProperties object that is used for response properties from the command. Within the JSP template, you use

the JSTL expression language to extract the information from the response properties.

In this section of the tutorial, you will learn the following:
- How to modify the controller command to include additional response properties in the TypedProperty
- How to modify the JSP template to retrieve information from the response properties using the JSTL expression language

To enable the display of information from the TypedProperties object in your JSP template, do the following:

1. The first step is to modify the MyNewControllerCmdImpl class, as follows:
   a. Switch to the Java perspective.
   b. Expand these directories: **WebSphereCommerceServerExtensionsLogic > src > com.ibm.commerce.sample.commands**.
   c. Double-click **MyNewControllerCmdImpl.java** and select its **performExecute** method in the Outline view.
   d. In the source code for the performExecute method, uncomment Section 2. This introduces the following code into the method:
```
/// Section 2 ////////////////////////////////////////////////

    /// see how the controller command pass in variables to JSP

 /// add additional parameters in controller command to rspProp
  ///  for response
    String message1 = "Hello from IBM!";

 rspProp.put("controllerParm1", message1);
 rspProp.put("controllerParm2", "Have a nice day!");

/// End of section 2//////////////////////////////////////////
```
      The preceding code snippet creates two new parameters that are put into the response properties object. This object is eventually passed to the view.
   e. Save your changes.
   f. Compile the command by right-clicking the **WebSphereCommerceServerExtensionsLogic** project and selecting **Build Project**.
2. Next you must update the MyNewJSPTemplate.jsp file, by doing the following:
   a. If the JSP template files are not already open, do the following:

1) In the Web perspective, switch to the Business Professional J2EE Navigator view Express Project Navigator view and expand the **Stores** Web project.
2) Navigate to the **Web Content*FashionFlow_name*** directory.
3) Right-click **MyNewJSPTemplate_All.jsp** and select **Open With > Page Designer**.
4) Right-click **MyNewJSPTemplate.jsp** and select **Open With > Page Designer**.

b. Copy section 4 from MyNewJSPTemplate_All.jsp file into your new MyNewJSPTemplate.jsp file. Place the new text between the `<!-- SECTION 4 -->` and `<!-- END OF SECTION 4 -->` markers. This introduces the following text into MyNewJSPTemplate.jsp:

```
<!-- SECTION 4 -->

<h3><fmt:message key="ParametersFromCmd" bundle="${tutorial}" /> </h3>

<fmt:message key="ControllerParm1" bundle="${tutorial}" />
<c:out value="${controllerParm1}"/> <br />

<fmt:message key="ControllerParm2" bundle="${tutorial}" />
<c:out value="${controllerParm2}"/> <br /> <br />

<!-- END OF SECTION 4 -->
```

This section uses the JSTL expression language to get and display the values that were passed in from the controller command.

c. Save your changes.

3. The next step is to test the modifications to the controller command and to the JSP template, by doing the following:

a. Switch to the Server Perspective (**Window > Open Perspective > Server**).

b. Right-click the **WebSphereCommerceServer** server and select **Start** (or **Restart**).

c. Right-click **index.jsp** under the Stores\Web Content*FashionFlow_name* directory and select **Run on Server**.
The store home page is displayed in the Web browser.

d. In the Web browser enter the following URL:

```
http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd
```

After a few seconds, the new JSP template is displayed, as shown in the following screen shot:

Hello World!.

Programmer's Guide

Tutorial: Creating new business logic

List of parameter-value pairs sent from the controller command

ControllerParm1= Hello from IBM!
ControllerParm2= Have a nice day!

Figure 33.

Passing information using a data bean

In this section, you add code to determine if the view was called directly, or if it was called by the controller command. In the latter case, the JSP template should also display the name of the command that called it.

In order to pass this information to the view, a new data bean, called MyNewDataBean is created. The MyNewJSPTemplate is also modified so that it can display the new information.

The MyNewDataBean is used strictly for making information available from the controller command to the JSP template. Contrast this to making information available from the database. Later in the tutorial, you will learn how to create a new data bean whose purpose is to make information from the database available to the JSP template.

You may wonder why this tutorial is using a data bean just to make information available from the controller command. There are two reasons for creating this bean: first, it is a good programming practice that allows a logical grouping of attributes, and second it makes it simpler for Web page developers to add information to the Web page using a data bean, rather than by using the TypedProperties response properties object.

In this section of the tutorial, you will learn the following:

- How to create a new data bean
- How to modify the controller command to instantiate a data bean
- How to set attributes in a data bean, using the controller command
- How to pass the instantiated data bean to the JSP template
- How to modify the JSP template to retrieve information from the data bean
- See an example of using the <if> tag in the JSP template

Creating MyNewDataBean

The MyNewDataBean is used to pass information to the MyNewJSPTemplate.jsp page. As with other sections of the tutorial, a base for the data bean is provided in the sample code. It is split into a number of different sections that are currently in the code as comments. As you progress through the tutorial, you uncomment various sections of the code.

To create MyNewDataBean, do the following:

1. Open the Java perspective and use the Package Explorer view.
2. Expand these directories: **WebSphereCommerceServerExtensionsLogic > src > com.ibm.commerce.sample.databeans**.
3. Double-click **MyNewDataBean.java** to view its source code.
4. In source code for the main class, uncomment Section 1. This introduces the following code into the class:

```
 /// Section 1 ///////////////////////////////////////////////
/// create fields and accessors (setter/getter methods)

 private java.lang.String callingCommandName = null;
 private boolean calledByControllerCmd = false;


 public java.lang.String getCallingCommandName() {
  return callingCommandName;
 }
```

```
public void setCallingCommandName(java.lang.String newCallingCommandName)
 {
 callingCommandName = newCallingCommandName;
}

public boolean getCalledByControllerCmd() {
 return calledByControllerCmd;
}

public void setCalledByControllerCmd(boolean newCalledByControllerCmd)
 {
 calledByControllerCmd = newCalledByControllerCmd;
}

/// End of Section 1 ////////////////////////////////////
```

The preceding code introduces two variables that are used to display information when the view was returned by a controller command, rather than being called directly by the URL for the view.

5. Save your changes.

Instantiating MyNewDataBean and setting its attributes using MyNewControllerCmd

In this step you modify MyNewControllerCmdImpl to instantiate MyNewDataBean and set the attributes of this bean.

To modify MyNewControllerCmdImpl, do the following:

1. In the Java perspective, expand these directories: **WebSphereCommerceServerExtensionsLogic > src > com.ibm.commerce.sample.commands** .
2. Double-click **MyNewControllerCmdImpl.java** to view its source code.
3. In the code for the main class, uncomment Import Section 1 to make the new data bean available in this class. This introduces the following code into the class:

```
/// Import Section 1 ///////////////////////////////
import com.ibm.commerce.sample.databeans.*;
/// End of Import Section 1 ////////////////////////
```

4. In the Outline view, select its **performExecute** method.
5. In the source code for the performExecute method, uncomment Sections 3A and 3B. This introduces the following code into the method:

```
/// Section 3A/////////////////////////////////////////

 /// instantiate the MyNewDataBean databean and set the properties,
  ///  then add the instance to resProp for response

 MyNewDataBean mndb = new MyNewDataBean();
 mndb.setCallingCommandName(this.getClass().getName());
 mndb.setCalledByControllerCmd(true);
```

```
/// end of section 3A/////////////////////////////////////////

/// Section 3B///////////////////////////////////////////
 rspProp.put("mndbInstance", mndb);

/// end of section 3B/////////////////////////////////////////
```

The preceding code snippet instantiates the MyNewDataBean object, sets two parameters in the object (indicating that it was called by a controller command and which command called it), and it then puts the data bean object into the response properties so that it will be made available to the JSP template.

6. Save your changes.
7. Compile the code changes by right-clicking on the **WebSphereCommerceServerExtensionsLogic** project and selecting **Build Project**.

Using MyNewDataBean in MyNewJSPTemplate

In this section, you modify MyNewJSPTemplate to indicate whether or not the view was returned by a controller command. If it was returned by a controller command, it should also display the name of that controller command. The JSP template uses JSTL tags for the conditional logic to determine this, based upon values from MyNewDataBean.

To modify the JSP template, do the following:

1. If the JSP template files are not already open, do the following:
   a. In the Web perspective, switch to the Business Professional J2EE Navigator view Express Project Navigator view and expand the **Stores** Web project.
   b. Navigate to the **Web Content*FashionFlow_name*directory.
   c. Highlight both the **MyNewJSPTemplate_All.jsp** and **MyNewJSPTemplate.jsp** files, right-click and select **Open With > Page Designer**.
2. Copy Section 5 from the MyNewJSPTemplate_All.jsp file into the MyNewJSPTemplate.jsp file. This introduces the following text into the JSP template:

```
<!-- SECTION 5 -->

<c:if test="${mndbInstance.calledByControllerCmd}">
   <fmt:message key="Example" bundle="${tutorial}" /> <br />
   <fmt:message key="CalledByControllerCmd" bundle="${tutorial}" />
      <br />
   <fmt:message key="CalledByWhichControllerCmd" bundle="${tutorial}" />
   <b><c:out value="${mndbInstance.callingCommandName}" /></b> <br />
```

```
      <br />
</c:if>

<!-- END OF SECTION 5 -->
```

This section of code uses the JSTL <if> tag to determine whether to display information about the calling controller command. It also retrieves translatable text from the resource bundle for the tutorial.

3. Save your changes.

Testing the modified JSP template

To test the modified JSP template, do the following:

1. Switch to the Server perspective (**Window > Open Perspective > Server**).
2. Right-click the **WebSphereCommerceServer** server and select **Start** (or **Restart**).
3. Right-click **index.jsp** under the Stores\Web Content*FashionFlow_name* directory and select **Run on Server**.
   The store home page is displayed in the Web browser.
4. In the Web browser enter the following URL:

   ```
   http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd
   ```

   After a few seconds, the JSP template is displayed, as shown in the following screen shot:

Hello World!.

Programmer's Guide

Tutorial: Creating new business logic

List of parameter-value pairs sent from the controller command

ControllerParm1= Hello from IBM!
ControllerParm2= Have a nice day!

This is an example of using the tag from JSP Standard Tag Library (JSTL)
MyNewView was called by a controller command
MyNewView was called by the controller command which is -
com.ibm.commerce.sample.commands.MyNewControllerCmdImpl

Figure 34.

5. Next, enter the URL to call the view directly, and notice the difference in the information displayed:

```
http://localhost/webapp/wcs/stores/servlet/MyNewView
```

Parsing and validating URL parameters in MyNewControllerCmd

In this step, you modify the controller command to make use of parameters that are passed in via the URL that calls the controller command. Validation logic is also included in the command to ensure that required parameters are included, as well as ensuring appropriate values are used for these parameters.

The validateParameters method that is currently in your new command is actually just a command stub. It consists of the following code:

```
public void validateParameters() throws ECApplicationException {
}
```

You must now add customized parameter checking to the command and pass the URL parameters to the JSP template. When modifying the validateParameters method, you add new fields that correspond to the URL parameters.

In this section of the tutorial, you will learn the following:

- How to add new fields to a command to correspond to the URL parameters
- How to use the getRequestProperties method to enable populating these fields with the URL input parameters
- How to catch missing parameter exceptions
- The appropriate way to pass URL parameters to a view
- See examples of various test cases of missing or incorrect parameter values

Adding new fields to MyNewControllerCmd

In this step, you create two new fields for the URL parameters. They are added to both the command interface and implementation class. One URL parameter is a string value that holds a user name and the second is an integer that will be used to accept an input value of bonus points.

To add these new fields, do the following:

1. Switch to the Java perspective.
2. Expand these directories: **WebSphereCommerceServerExtensionsLogic > src > com.ibm.commerce.sample.commands**.
3. Double-click the **MyNewControllerCmd.java** interface to view its source code.
4. Uncomment Section 2 to add the new fields and corresponding getter methods to the interface. This introduces the following code to the class:

```
/// Section 2 ////////////////////////////////////////////////

// set interface methods
```

```
 public java.lang.Integer getPoints() ;

 public java.lang.String getUserName() ;

 public void setPoints(java.lang.Integer newPoints) ;

 public void setUserName(java.lang.String newUserName)  ;


/// End of section 2////////////////////////////////////////////
```

5. Save your changes.
6. Double-click the **MyNewControllerCmdImpl.java** class to view its source code.
7. Uncomment Section 1 in the main class to add the new fields as well as their corresponding getter and setter methods to the class. This introduces the following code into the class:

```
/// Section 1 ///////////////////////////////////////////////

/// create and implement controller command's fields and accessors
/// (setter/getter methods)

 private java.lang.String userName = null;
 private java.lang.Integer points;


 public java.lang.Integer getPoints() {
  return points;
 }

 public java.lang.String getUserName() {
  return userName;
 }

 public void setPoints(java.lang.Integer newPoints) {
  points = newPoints;
 }

 public void setUserName(java.lang.String newUserName) {
  userName = newUserName;
 }

/// End of Section 1 ////////////////////////////////////////
```

8. Save your changes.

Passing URL parameters to the view

In this step, you include the code to pass the input parameters to the JSP template. This is done by setting fields in the data bean with the values of the input parameters.

To pass the URL parameters, do the following:

1. Double-click **MyNewControllerCmdImpl.java**.

2. In the Outline view, select the **performExecute** method.
3. In the source code of the performExecute method, uncomment Section 3C. This introduces the following code into the method:

```
/// Section 3C/////////////////////////////////////////

 // pass the input information to the databean
 mndb.setUserName(this.getUserName());
 mndb.setPoints(this.getPoints());

/// end of section 3C/////////////////////////////////////////
```

   This code sets the values in the data bean object so that they will be available to the JSP template.
4. Save your changes.

Catching missing parameters and validating values

In this step, you modify the validateParameters method to introduce error checking and parameter validation logic. Once modified, the code checks for the following:

- If the first input parameter is not provided, a "parameter not found" exception is thrown. This is due to the fact that the first input parameter is a required parameter. In this case, the generic error page is displayed to the customer.
- The second input parameter is optional. As such, if the second input parameter is not provided, a "parameter not found exception" is *not* thrown. Instead, the value for the second input parameter is defaulted to zero and the customer is not affected by the error. Processing continues.

To add this error checking, do the following:

1. Double-click **MyNewControllerCmdImpl.java**.
2. In the Outline view, select its **validateParameters** method.
3. In the source code of the validateParameters method, uncomment Section 1. This introduces the following code into the method:

```
/// Section 1 ///////////////////////////////////////////////////

/// uncomment to check parameters

    final String strMethodName = "validateParameters";

 TypedProperty prop = getRequestProperties();

 /// retrieve required parameters
 try {
  setUserName(prop.getString("input1"));

 } catch (ParameterNotFoundException e) {
  /// the next exception uses _ERR_CMD_MISSING_PARAM ECMessage object
```

```
    /// defined in ECMessage class
   throw new ECApplicationException(ECMessage._ERR_CMD_MISSING_PARAM,
        this.getClass().getName(), strMethodName,
      ECMessageHelper.generateMsgParms(e.getParamName()));
  }

  /// retrieve optional Integer
  // set input2 = 0 if no input value
  setPoints(prop.getInteger("input2",0));

/// End of section 1/////////////////////////////////////////////////
```

The preceding code snippet checks the two input parameters. The try block determines if the first parameter exists; if not, an exception is thrown. Since the second parameter is optional, this code sets the value to zero if it is either missing or an incorrect value.

4. Save your changes.

Adding new fields to MyNewDataBean

In this step, you add new fields and their associated getter methods to the MyNewDataBean data bean, so that the URL parameters will be available to the JSP template.

To modify MyNewDataBean, do the following:

1. Double-click **MyNewDataBean.java** to view its source code.
2. Uncomment Section 2 to introduce the following code into the class:

```
/// Section 2 ////////////////////////////////////////////

 private java.lang.String userName = null;
 private java.lang.Integer points;


 public String getUserName() {
  return userName;
 }

 public void setUserName(java.lang.String newUserName) {
  userName = newUserName;
 }


 public Integer getPoints() {
  return points;
 }

 public void setPoints(java.lang.Integer newPoints) {
  points = newPoints;
 }


/// End of Section 2 ////////////////////////////////////
```

3. Save your changes.
4. Compile the code changes by right-clicking the **WebSphereCommerceServerExtensionsLogic** and selecting **Build Project**.

Modifying MyNewJSPTemplate to display the URL parameters

In this step, you modify the MyNewJSPTemplate.jsp file to add a new section that displays the URL input parameters, by doing the following:

1. If the JSP template files are not already open, do the following:
   a. In the Web perspective, switch to the Business Professional J2EE Navigator view Express Project Navigator view and expand the **Stores** Web project.
   b. Navigate to the Web Content*FashionFlow_name* sub-folder.
   c. Highlight both the **MyNewJSPTemplate_All.jsp** and **MyNewJSPTemplate.jsp** files, right-click and select **Open With > Page Designer**.
2. Copy Section 6 from the MyNewJSPTemplate_All.jsp file into the MyNewJSPTemplate.jsp file. This introduces the following text into the JSP template:

```
<!-- SECTION 6 -->

<fmt:message key="UserName" bundle="${tutorial}" />
<c:out value="${mndbInstance.userName}"/> <br>

<fmt:message key="Points" bundle="${tutorial}" />
<c:out value="${mndbInstance.points}"/> <br>

<!-- END OF SECTION 6 -->
```

3. Save your changes.

Testing URL parameter values

The next step is to test if the new error checking is working properly, by doing the following:

1. Switch to the Server perspective (**Window > Open Perspective > Server**).
2. Right-click the **WebSphereCommerceServer** server and select **Start** (or **Restart**).
3. Right-click **index.jsp** under the Stores\Web Content*FashionFlow_name* directory and select **Run on Server**.
   The store home page is displayed in the Web browser.
4. Case 1: The first test case will be to exclude both parameters from the URL. After the store home page is displayed, enter the following URL:
   `http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd`

Since no parameters are passed to the command, a generic application error is shown.

Generic Error

The store is currently experiencing problems. Try again later.

Store developers:
To view detailed error information, see the comments in the HTML source for this page.

Figure 35.

The console in WebSphere Studio Application Developer shows information similar to the following:

```
timeStamp 6730e546 CommerceSrvr  E
com.ibm.commerce.sample.commands.MyNewControllerCmdImpl
validateParameters CMN0206E Please check all fields. "input1" is a
required field.
```

5. Case 2: The next test case is to use a valid first parameter, but omit the second parameter. In this case, you expect that no error should be detected, since by default, a zero value will be used for the missing second parameter. Enter the following URL:

   ```
   http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd?input1=abc
   ```

   The result of this command is that the MyNewJSPTemplate page is displayed and a zero value is shown for input2.

Programmer's Guide

Tutorial: Creating new business logic

List of parameter-value pairs sent from the controller command

ControllerParm1= Hello from IBM!
ControllerParm2= Have a nice day!

This is an example of using the tag from JSP Standard Tag Library (JSTL)
MyNewView was called by a controller command
MyNewView was called by the controller command which is - **com.ibm.commerce.sample.commands.MyNewControllerCmdImpl**

UserName= abc
Points= 0

Figure 36.

6. Case 3: In this test case, a valid parameter is provided for the first input parameter, and an invalid parameter is provided for the second parameter (a string is used rather than an integer). Similar to the last case, you should not see an error, since the error handline changes the second input parameter to a zero. Enter the following URL:

```
http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd?
   input1=abc&input2=abc
```

   The result of this command is that the MyNewJSPTemplate page is displayed and a zero value is shown for input2.

Programmer's Guide

Tutorial: Creating new business logic

List of parameter-value pairs sent from the controller command

ControllerParm1= Hello from IBM!
ControllerParm2= Have a nice day!

This is an example of using the tag from JSP Standard Tag Library (JSTL)
MyNewView was called by a controller command
MyNewView was called by the controller command which is -
com.ibm.commerce.sample.commands.MyNewControllerCmdImpl

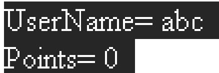


Figure 37.

7. Case 4: In this case, valid parameters are used for both of the URL input parameters. Enter the following URL:

```
http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd?
   input1=abc&input2=1000
```

The result of this command is that the MyNewJSPTemplate page is displayed and 1000 points are displayed for the user.

Programmer's Guide

Tutorial: Creating new business logic

List of parameter-value pairs sent from the controller command

ControllerParm1= Hello from IBM!
ControllerParm2= Have a nice day!

This is an example of using the tag from JSP Standard Tag Library (JSTL)
MyNewView was called by a controller command
MyNewView was called by the controller command which is -
com.ibm.commerce.sample.commands.MyNewControllerCmdImpl

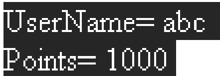


Figure 38.

Creating a new task command

A controller command typically represents a business process or complex function. For example, all of the business logic related to processing orders is encapsulated in the OrderProcessCmd controller command. A business process can often be divided up into smaller, more specific tasks. For example, within the OrderProcessCmd controller command, there are several task commands that get called to perform individual units of work.

MyNewControllerCmdImpl does not currently call any task commands. This section is divided into two steps. In the first step, you create the new task command. In the second step, you modify the performExecute method of the controller command to call the new task command.

In this step, you create a new task command interface and its associated implementation class. Initially, the new task command does very little except handle view parameters. It only has fields for defaultCommandClassName, URL parameters, and the current value of bonus points. It has a method for getting the current value of bonus points.

In this step of the tutorial, you will learn the following:
• How to create a new task command interface and its associated implementation class
• The minimum amount of code that is required in a task command
• How to add fields and methods to the task command

Creating MyNewTaskCmd

This step shows you how to write a new task command. Creating a completely new task command involves creating an interface and an implementation class. When creating a task command, the interface should extend `com.ibm.commerce.commands.TaskCommand`. The implementation class should extend `com.ibm.commerce.command.TaskCommandImpl`.

Upon completion of this exercise, you will have a task new command, called MyNewTaskCmd. This command is used by a your store that is based upon the FashionFlow sample.

To create MyNewTaskCmd, do the following:

1. Switch to the Java perspective and select the **WebSphereCommerceServerExtensionsLogic** project.
2. Expand the **src** directory, and then the **com.ibm.commerce.sample.commands** directory.
3. Double-click the **MyNewTaskCmd.java** interface to view its source code.
4. In the source code for this interface, uncomment Section 1 to create a field that specifies the default implementation class to be used by the interface. This introduces the following code into the interface:

```
/// Section 1 ////////////////////////////////////////////////

// set default command implement class

 static final String defaultCommandClassName=
  "com.ibm.commerce.sample.commands.MyNewTaskCmdImpl";


/// End of section 1///////////////////////////////////////////
```

   Since the same implementation class is used for the entire site and no default properties are passed to the command, you can specify the default implementation right in the code. If you have a command that

either has multiple implementations, or has default properties (which are stored in the CMDREG table), you must register the command in the CMDREG table to create the mapping between the interface and implementation class.

5. Next, uncomment Section 2 to create getter and setter methods that will be used in the MyNewTaskCmdImpl implementation class. These methods are for fields corresponding to the following types of information:
   - A customer's user ID.
   - A value of bonus points
   - A greeting message

   By uncommenting Section 2, the following code is introduced into the interface:

```
/// Section 2 ////////////////////////////////////////////////
// set interface methods

 public void setInputUserName(java.lang.String inputUserName);
 public void setInputPoints(Integer inputPoints);
 public void setGreetings(java.lang.String greeting);

 public java.lang.String getInputUserName();
 public java.lang.Integer getInputPoints();
 public java.lang.String getGreetings();


/// End of section 2//////////////////////////////////////////
```

6. Save your changes.
7. Double-click the **MyNewTaskCmdImpl.java** implementation class to view its source code.
8. Uncomment Sections 1A and 1B to create fields and their corresponding getter and setter methods in the implementation class. This introduces the following code into the class:

```
//// Section 1A ///////////////////////////////////////////////

 private java.lang.String inputUserName;
 private java.lang.String greetings;
 private java.lang.Integer inputPoints;


////End of Section 1A /////////////////////////////////////////


//// Section 1B ///////////////////////////////////////////////

 public void setInputUserName(java.lang.String newInputUserName) {
  inputUserName = newInputUserName;
 }
```

```
public void setInputPoints(Integer newInputPoints) {
 inputPoints = newInputPoints;
}

public void setGreetings(java.lang.String newGreetings) {
 greetings = newGreetings;
}

public java.lang.String getInputUserName() {
 return inputUserName;
}

public Integer getInputPoints() {
 return inputPoints;
}

public java.lang.String getGreetings() {
 return greetings;
}


////End of Section 1B ////////////////////////////////////////
```

Save your work.

9. In the Outline view, select the **performExecute** method of the MyNewTaskCmdImpl class.
10. In the source code of this performExecute method, uncomment Section 1 to introduce the following code into the method:

```
/// Section 1 /////////////////////////////////////////////////

/// modify the greetings and see it in the NVP list

 setGreetings( "Hello ! " + getInputUserName() );

/// End of section 1 //////////////////////////////////////////
```

This updates the greetings value. The greetings value is then made available then to other objects via the getGreetings() method. It will be added to the name-value pair (NVP) list.

11. Save your work.

Calling the task command

Once you have created your task command, you need to call the command from within your controller command. The following steps show how to modify your controller command in this manner:

1. In the Java perspective, double-click the **MyNewControllerCmdImpl.java** class
2. In the Outline view, select its **performExecute** method.
3. In the source code for the performExecute method, navigate to Area 4.

4. Uncomment Sections 4A, 4B, and 4C to introduce the following code into the method:

```
/// Section 4A
    /// see how the controller command call a task command

 MyNewTaskCmd cmd = null;

 try {

  cmd = (MyNewTaskCmd) CommandFactory.createCommand(
         "com.ibm.commerce.sample.commands.MyNewTaskCmd", getStoreId());

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

  /// set input parameters to task command
  cmd.setInputUserName(getUserName());
  cmd.setInputPoints(getPoints()); // change to Integer

/// End Section 4A ///////////////////////////////////////

/// Section 4B ///////////////////////////////////

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

  /// retrieve output parameter from task command, then put it to
     /// response properties
  rspProp.put("taskOutputGreetings", cmd.getGreetings());

/// End Section 4B /////////////////////////////////////

/// Start Section 4C /////////////////////////////////////
 } catch (ECException ex) {
  /// throw the exception as is
  throw (ECException) ex;
 }

/// End Section 4C //////////////////////////////////////
```

Section 4A creates the new task command object using the command factory. It then sets the command context, and sets the input parameters of the task command. Section 4B calls the validateParameters method for access control purposes, before invoking the execute method of the task command. It then retrieves the greetings value from the task command. Section 4C is a simple catch block for exceptions.

5. Save your changes.
6. Compile the code changes by right-clicking the **WebSphereCommerceServerExtensionsLogic** project and selecting **Build Project**.

Modifying MyNewJSPTemplate to add the greetings message

In this step, you modify the MyNewJSPTemplate.jsp file to add a new section that displays the greetings message, by doing the following:

1. If the JSP template files are not already open, do the following:
   a. In the Web perspective, switch to the Business Professional J2EE Navigator view Express Project Navigator view and expand the **Stores** Web project.
   b. Navigate to the Web Content*FashionFlow_name* sub-folder.
   c. Highlight both the **MyNewJSPTemplate_All.jsp** and **MyNewJSPTemplate.jsp** files, right-click and select **Open With > Page Designer**.
2. Copy Section 7 from the MyNewJSPTemplate_All.jsp file into the MyNewJSPTemplate.jsp file. This introduces the following text into the JSP template:

```
<!-- SECTION 7 -->

<fmt:message key="Greeting" bundle="${tutorial}" />
<c:out value="${taskOutputGreetings}"/> <br /> <br />

<!-- END OF SECTION 7 -->
```

3. Save your changes.

Testing MyNewTaskCmd

The next step is to test if the new task command is working properly, by doing the following:

1. Switch to the Server perspective (**Window > Open Perspective > Server**).
2. Right-click the **WebSphereCommerceServer** server and select **Start** (or **Restart**).
3. Right-click **index.jsp** under the Stores\Web Content*FashionFlow_name* directory and select **Run on Server**.
   The store home page is displayed in the Web browser.
4. Next, enter the following URL:

```
http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd?
   input1=abc&input2=1000
```

   The MyNewJSPTemplate is displayed. It includes the greeting message that was created by the task command.

Programmer's Guide

Tutorial: Creating new business logic

List of parameter-value pairs sent from the controller command

ControllerParm1= Hello from IBM!
ControllerParm2= Have a nice day!

This is an example of using the tag from JSP Standard Tag Library (JSTL)
MyNewView was called by a controller command
MyNewView was called by the controller command which is - **com.ibm.commerce.sample.commands.MyNewControllerCmdImpl**

UserName= abc
Points= 1000
Greeting= Hello ! abc

Figure 39.

Modifying MyNewTaskCmd

In this step of the tutorial, you modify MyNewTaskCmd to determine if the user name included in the URL is that of a registered user. MyNewDataBean is also modified to handle fields from the task command (for example, the user name). Correspondingly, MyNewJSPTemplate is modified to display whether or not the user is registered.

In this section of the tutorial, you will learn the following:

- How to use URL parameters and access existing WebSphere Commerce information from within your own customized code

Modifying MyNewControllerCmdImpl to create an object for the task command

In order to promote the efficient use of objects, the controller command creates a UserRegistryAccessBean object instance variable. As a result, this object is also available to the task command. By taking this approach, the task command does not need to create a separate instance of the object. This UserRegistryAccessBean object is used later (in the task command) to determine if the shopper is a registered user.

To modify MyNewControllerCmdImpl, do the following:

1. In the Java perspective, double-click the **MyNewControllerCmdImpl.java** class to view its source code.
2. In the main body of this class, uncomment Section 2, to introduce the following code into the class:
```
/// Section 2 ////////////////////////////////////////////////
/// create a user registry accessbean resource instance variable

    private UserRegistryAccessBean rrb = null;

/// End of Section 2 /////////////////////////////////////////
```
3. In the Outline view, select the **performExecute** method.
4. In the source code for the performExecute method uncomment Sections 4D and 4F to pass the instance variable to the task command and then make the returned user ID available in the response properties. This introduces the following code into the method:
```
// Section 4D ///////////////////////////////////
/// pass rrb instance variable to the task command

  cmd.setUserRegistryAccessBean(rrb);

// End of section 4D ////////////////////////////


// Section 4F ///////////////////////////////////////
///using access bean to get information from database
  if (cmd.getFoundUserId() != null) {
   rspProp.put("taskOutputUserId", cmd.getFoundUserId());
  }
// End of section 4F ////////////////////////////
```
   You will receive an error indicating that some of the methods are undefined, but this will be resolved in the next step when you modify your task command.
5. Save your work.

Modifying the new task command for user name validation

Next you must modify the new task command to validate if the user name input in the URL is that of a registered user, as follows:

1. Double-click the **MyNewTaskCmd.java** interface to view its source code.
2. Uncomment Section 3, to introduce the following code into the interface:

```
/// Section 3 ////////////////////////////////////////////////

 public void setFoundUserId(java.lang.String inputUserId);
 public java.lang.String getFoundUserId();

 public void setUserRegistryAccessBean(UserRegistryAccessBean rrb);

/// End of section 3/////////////////////////////////////////
```

3. Save your work.
4. Double-click the **MyNewTaskCmdImpl.java** class to view its source code. Uncomment Import section 1 to introduce the following two import statements into the code:

```
/// Import section 1 ///////////////////////////////////////
import com.ibm.commerce.user.objects.*;
import com.ibm.commerce.sample.databeans.*;
///  End of Import section 1 ///////////////////////////////
```

5. Uncomment Sections 2A and 2B to create the new fields and getter and setter methods that correspond to the methods added into the interface. This introduces the following code into the class:

```
//// Section 2A /////////////////////////////////////////////

 private java.lang.String foundUserId = null;

 private UserRegistryAccessBean rrb = null;

////End of Section 2A ///////////////////////////////////////

//// Section 2B /////////////////////////////////////////////

 public void setUserRegistryAccessBean(UserRegistryAccessBean newRRB) {
  rrb = newRRB;
 }

 public void setFoundUserId(java.lang.String newFoundUserId) {
  foundUserId = newFoundUserId;
 }

 public java.lang.String getFoundUserId() {
  return foundUserId;
 }


/// End of section 2B ///////////////////////////////////////////
```

6. In the Outline view, select the **validateParameters** method and examine its source code. Uncomment Section 1, to introduce the following code into the method:

```
// section 1 ///////////////////////////////////////////////////

// use UserRegistryAccessBean to check user Id

  try {

   if (rrb!=null){
    setFoundUserId(rrb.getUserId());
   } else {
     rrb =new UserRegistryAccessBean();
     rrb=rrb.findByUserLogonId(getInputUserName());
     setFoundUserId(rrb.getUserId());
   }

  } catch (javax.ejb.FinderException e) {
    return;

  } catch (java.rmi.RemoteException e) {
   throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
    this.getClass().getName(), "validateParameters");
  } catch (javax.naming.NamingException e) {
   throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
    this.getClass().getName(), "validateParameters");
  } catch (javax.ejb.CreateException e) {
   throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
    this.getClass().getName(), "validateParameters");
  }

// end of section 1 ////////////////////////////////////////////
```

7. Save your work.
8. Compile the code changes by right-clicking the **WebSphereCommerceServerExtensionsLogic** project and selecting **Build Project**.

Modify MyNewJSPTemplate for user name validation

The current JSP template must be modified in order to display the user name validation information. To modify this file, do the following:

1. Switch to the Web perspective.
2. Open both the **MyNewJSPTemplate_All.jsp** and **MyNewJSPTemplate.jsp** files.
3. Copy Section 8 from the MyNewJSPTemplate_All.jsp file into the MyNewJSPTemplate.jsp file. This introduces the following text into the JSP template:

```
<!-- SECTION 8 -->

<c:if test="${!empty taskOutputUserId}">
```

```
   <fmt:message key="UserId" bundle="${tutorial}" />
   <c:out value="${taskOutputUserId}"/> <br />
   <fmt:message key="FirstInput" bundle="${tutorial}" />
   <b><c:out value="${userName}"/></b>
   <fmt:message key="RegisteredUser" bundle="${tutorial}" /> <br />
   <fmt:message key="ReferenceNumber" bundle="${tutorial}" />
   <b><c:out value="${taskOutputUserId}"/></b> <br /> <br />
</c:if>

<c:if test="${empty taskOutputUserId}">
   <fmt:message key="FirstInput" bundle="${tutorial}" />
   <b><c:out value="${userName}"/></b>
   <fmt:message key="NotRegisteredUser" bundle="${tutorial}" /> <br />
</c:if>


<!-- END OF SECTION 8 -->
```

4. Save the changes made to the MyNewJSPTemplate.jsp file.

Testing user name validation

To test the user name validation logic, do the following:

1. Switch to the Server perspective (**Window > Open Perspective > Server**).
2. Right-click the **WebSphereCommerceServer** server and select **Start** (or **Restart**).
3. Right-click **index.jsp** under the Stores\Web Content*FashionFlow_name* directory and select **Run on Server**.
   The store home page is displayed in the Web browser.
4. Create a new registered user, by doing the following:
   a. Click **Register**.
   b. Click **Register** again to create a new customer.
   c. In the registration form, enter appropriate values into all of the mandatory fields. For example, in the e-mail field, enter `tester@mycompany`. Make note of the value for the e-mail address: ________________________.
   d. Once the values have been entered, click **Submit**.
5. Next, enter a valid user name as the value for input1. Enter the following URL:

   ```
   http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd?
      input1=new_e-mail&input2=1000
   ```

   where *new_e-mail* is the e-mail address for the user created in step 4. The MyNewJSPTemplate is displayed. It should now indicate that the value for input1 is a valid user name.

Programmer's Guide

Tutorial: Creating new business logic

List of parameter-value pairs sent from the controller command

ControllerParm1= Hello from IBM!
ControllerParm2= Have a nice day!

This is an example of using the tag from JSP Standard Tag Library (JSTL)

MyNewView was called by a controller command
MyNewView was called by the controller command which is -
com.ibm.commerce.sample.commands.MyNewControllerCmdImpl

UserName= tester@mycompany
Points= 1000
Greeting= Hello ! tester@mycompany

UserId= 1002
Your first input parameter is a registered user
The member refernce number of this user is **1002**

Figure 40.

6. Next, enter the following URL in which the value for the user name is invalid:

```
http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd?
   input1=abc&input2=1000
```

A generic exception is displayed. If you view the source of the page, you find that a _ERR_FINDER_EXCEPTION occurred. This is due to the fact that the user name provided does not correspond to a registered user.

Creating a new entity bean

This section describes how to create a new entity bean. In this example scenario, you have a business requirement to include a tally of bonus points for each user in the commerce application. The WebSphere Commerce database schema does not contain this information, so you need to create a new database table to hold this information. In accordance with the WebSphere Commerce programming model, once the database table is created, you must create an entity bean to access the data.

Creating the XBONUS table

In preparation for creating the entity bean, you must first create the new database table. The table to be created is called XBONUS.

DB2 If you are using a DB2 database, do the following to create the table:

1. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Line Tools > Command Center**) and click the **Scripting** tab.
2. In the Script window, enter the following:

```
connect to developmentDB user dbuser using dbpassword;
create table XBONUS (MEMBERID BIGINT NOT NULL,
    BONUSPOINT INTEGER NOT NULL,
           constraint p_xbonus primary key (MEMBERID),
    constraint f_xbonus foreign key (MEMBERID)
    references users (users_id) on delete cascade)
```

where
- *developmentDB* is the name of your development database
- *dbuser* is the database user
- *dbpassword* is the password for your database user

Click the Execute icon.
You should see a message indicating that the SQL statement completed successfully.

Oracle If you are using an Oracle database, do the following to create the table:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

```
create table XBONUS (MEMBERID NUMBER NOT NULL,
    BONUSPOINT INTEGER NOT NULL,
           constraint p_xbonus primary key (MEMBERID),
    constraint f_xbonus foreign key (MEMBERID)
    references users (users_id) on delete cascade);
```

and press Enter to run the SQL statement. The XBONUS table is now created.

6. Enter the following to commit your database changes:

```
commit;
```

and press Enter to run the SQL statement.

Creating the BonusBean entity bean

Once the table has been created, you are ready to begin creating the new entity bean. The next steps use WebSphere Studio Application Developer to create this bean.

Next you create the new Bonus bean, by doing the following:

1. In WebSphere Studio Application Developer, switch to the J2EE perspective.
2. Within the J2EE Hierarchy view, expand **EJB Modules**.
3. Right-click the **WebSphereCommerceServerExtensionsData** module and select **New > Enterprise Bean**.
   The Enterprise Bean Creation wizard opens.
4. From the **EJB Project** drop-down list, select **WebSphereCommerceServerExtensionsData** and click **Next**.
5. In the Create an Enterprise Bean window, do the following:
   a. Select **Entity bean with container-managed persistence (CMP) fields**
   b. In the **Bean name** field, enter `Bonus`.
   c. In the **Source folder** field, leave the default value that is specified (ejbModule).
   d. In the **Default package** field, enter `com.ibm.commerce.extension.objects`.
   e. Click **Next**.
6. In the Enterprise Bean Details window, do the following:
   a. Click **Add** to add new CMP attributes for the MEMBERID and BONUSPOINT columns in the BONUS table.
      The Create CMP Attribute window opens. In this window, do the following:
      1) In the **Name** field, enter `memberId`.
      2) In the **Type** field, enter `java.lang.Long`.

Note: You must use the *java.lang.Long* data type, not the *long* data type.

3) Select the **Key Field** check box.
4) Click **Apply**.
5) In the **Name** field, enter `bonusPoint`.
6) In the **Type** field, enter `java.lang.Integer`.

Note: You must use the *java.lang.Integer* data type, not the *integer* data type.

7) Select the **Access with getter and setter methods** check box.
8) Clear the **Promote getter and setter methods to remote interface** check box. The **Make getter read-only** check box will be made unavailable.
9) Click **Apply**.
10) Click **Close** to close this window.

b. Clear the **Use the single key attribute type for the key class** check box, then click **Next**.

7. In the EJB Java Class Details window, do the following:
   a. To select the bean's superclass, click **Browse**.
      The Type Selection window opens.
   b. In the **Select a class using: (any)** field, enter `ECEntityBean` and click **OK**. This selects the com.ibm.commerce.base.objects.ECEntityBean as the superclass.
   c. Specify the interfaces that the remote interface should extend by clicking **Add**. The Type Selection window opens.
   d. In the **Select a class using: (any)** field, enter `Protectable` and click **OK**. This selects com.ibm.commerce.security.Protectable. This interface is required in order to protect the new resource under access control.
   e. Click **Finish**.

Set the isolation level for the new bean, by doing the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**.
2. Double-click the **WebSphereCommerceServerExtensionsData** project to open it with the Deployment Descriptor Editor. As an alternative way to open this file in this editor, you could do the following:
   a. In the Business Professional J2EE Navigator view Express Project Navigator view, expand **WebSphereCommerceServerExtensionsData > ejbModule > META-INF**.
   b. Right-click **ejb-jar.xml** and select **Open With > Deployment Descriptor Editor**)
3. Click the **Access** tab.

4. Click **Add** next to the Isolation Level text box.
   The Add Isolation Level window opens.
5. DB2 Select **Repeatable Read**, then click **Next**.

   * DB2 400 Select **Read Committed**, then click **Next**.

   Oracle Select **Read Committed**, then click **Next**
6. From the **Beans found** list, select the **Bonus** bean, then click **Next**.
7. From the **Methods found** list, select **Bonus** to select all of its methods, and click **Finish**.
8. Save your work (Ctrl+S), and keep the editor open.

Next, set the security identity of the bean, by doing the following:
1. In the Deployment Descriptor editor, ensure that you have the Access tab selected.
2. Click **Add** next to the Security Identity text box.
   The Add Security Identity window opens.
3. Select **Use identity of EJB server**, then click **Next**.
4. From the **Beans found** list, select the **Bonus** bean, then click **Next**.
5. From the **Methods found** list, select **Bonus** to select all of its methods, and click **Finish**.
6. Save your work (Ctrl+S) and keep the editor open.

Next, set the security role for the methods in the bean, by doing the following:
1. In the Deployment Descriptor editor, select the Assembly Descriptor tab.
2. In the Method permissions section, click **Add**.
3. Select **WCSecurityRole** as the security role and click **Next**.
4. From the list of beans found, select **Bonus** and click **Next**.
5. In the Method elements page, click **Apply to All**, then click **Finish**.
6. Save your work (Ctrl+S) and close the deployment descriptor editor.

The next step is to remove some of the fields and methods related to the entity context that are generated by WebSphere Studio Application Developer. The reason that these fields need to be deleted is that the ECEntityBean base class provides its own implementation of these methods. To delete the generated entity context fields and methods, do the following:
1. In the J2EE Hierarchy view, expand the **WebSphereCommerceServerExtensionsData** project.
2. Expand the **Bonus** bean and then double-click the **BonusBean** class.
3. In the Outline view, do the following:
   a. Right-click the **myEntityCtx** field and select **Delete**.

b. Right-click the **getEntityContext()** method and select **Delete**.
c. Right-click the **setEntityContext(EntityContext)** method and select **Delete**.
d. Right-click the **unsetEntityContext()** method and select **Delete**.

4. Save your work (Ctrl+S). Keep the BonusBean class open.

Next, add a new getMemberId method to the enterprise bean, by doing the following:

1. View the source code of the **BonusBean** class.
2. Add the following code to the end of this class (still within the class):

```
public java.lang.Long getMemberId() {
   return memberId;
}
```

3. You must add the new method to the remote interface, by doing the following:
   a. In the Outline view, right-click the **getMemberId** method and select **Enterprise Bean > Promote to Remote Interface**. Once this is complete, a small R icon is displayed next to the method, indicating that it has been promoted to the remote interface.
4. Save your work.
5. Close the BonusBean editor.

Next, add new FinderHelper methods to the BonusHome interface, by doing the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**.
2. Double-click the **WebSphereCommerceServerExtensionsData** project to open the EJB Deployment Descriptor Editor.
3. Click the **Beans** tab.
4. In the Beans pane, select the **Bonus** bean, then in the pane on the right, scroll down and expand **WebSphere Extensions**.
5. Click **Add** next to the **Finders** text box.
   The Add Finder Descriptor window opens.
6. Select **New**, then in the **Name** field, enter `findByMemberId`.
7. Click **Add** next to the **Parameters** text box, then do the following
   a. In the **Name** field, enter `memberId`.
   b. In the **Type** field enter `java.lang.Long`.
   c. Click **OK**.
8. From the **Return Type** drop-down list, select **com.ibm.commerce.extension.objects.Bonus**, then click **Next**.
9. From the **Finder type** drop-down list, select **WhereClauseFinderDescriptor**.

10. In the **Finder statement** field, enter T1.MEMBERID = ?, then click **Finish**.
11. Save your work, then close the EJB Deployment Descriptor editor.

Next, add a new ejbCreate method to the Bonus bean, by doing the following:

1. In the J2EE Hierarchy view, double-click the **BonusBean** class to open it and view its source code.
2. Create a new ejbCreate(Long, Integer) method, by adding the following code into the class:

```
public com.ibm.commerce.extension.objects.BonusKey ejbCreate(
   java.lang.Long memberId,java.lang.Integer bonusPoint)
   throws javax.ejb.CreateException {
      _initLinks();
      this.memberId=memberId;
      this.bonusPoint=bonusPoint;
      return null;
}
```

3. Save the code changes.
4. You must add the new ejbCreate(Long, Integer) method to the home interface. This makes the method available in the generated access bean. To add the method to the home interface, do the following:
   a. In the Outline view, right-click the **ejbCreate(Long, Integer)** method and select **Enterprise Bean > Promote to Home Interface**.

Next create a new ejbPostCreate(Long, Integer) method so that it has the same parameters as the ejbCreate(Long, Integer) method, by doing the following:

1. Double-click the **BonusBean** class to open it and view its source code.
2. Create a new ejbPostCreate(Long, Integer) method, by adding the following code into the class:

```
public void ejbPostCreate(java.lang.Long memberId,
   java.lang.Integer bonusPoint)
   throws javax.ejb.CreateException
   {
   }
```

3. Save the code changes.

The next steps add new methods to the Bonus bean so that it can be protected by the WebSphere Commerce access control system. The getOwner and fulfills methods are be added to the bean, by doing the following:

1. Double-click the **BonusBean** class to open it and view its source code.
2. Add the new getOwner method to the BonusBean class, by adding the following code into the end of the class:

```
public java.lang.Long getOwner()
   throws java.lang.Exception {
      return getMemberId();
   }
```

3. Save your work.
4. Add the new fulfills method, by adding the following code to the end of the class:

```
public boolean fulfills(Long member, String relationship)
   throws java.lang.Exception {
      if (relationship.equalsIgnoreCase("creator"))
      {
         return member.equals(getMemberId());
      }
      return false;
   }
```

5. Save your work and close the bonus bean editor.

The next step is to map the XBONUS table to the BonusBean entity bean. Meet-in-the-middle mapping is used. To create the mapping, do the following:

Business Professional

1. In the J2EE Hierarchy view, right-click **WebSphereCommerceServerExtensionsData** and select **Generate > EJB to RDB Mapping**.
   The EJB to RDB Mapping Window opens.
2. Select **Meet In The Middle** and click **Next**.
3. In the Database Connection window, do the following:
   a. In the **Connection name** field, enter `WebSphereCommerceServerExtensionsData`
   b. In the **Database field**, enter the name of your development database.
   c. In the **User ID** field, enter the database user ID.
   d. In the **Password** field, enter the password for the database user.
   e. From the **Database vendor type** drop-down list, select the database vendor type for your development database.
      - DB2 DB2 Universal Database 8.1
      - Oracle Oracle 9i
   f. Oracle In the **Host** field, enter the fully-qualified host name of your database server. For example, enter `dbserver.yourcompany.com`
   g. Oracle In the Class Location field, enter the location of the classes12.zip file. For example, enter `D:\oracle\ora92\jdbc\lib\classes12.zip`
   h. Click **Next**. Once the connection is established, the list of tables in the database is displayed. You can also view the Connection Document later by looking at the Database Servers view in the Data perspective.
4. Select the **XBONUS** table and click **Next**.

5. Select **Match By Name and Type** and then click **Finish**. The Mapping Editor now opens.
6. Oracle Right-click the XBONUS table and select **Open Table Editor**. In the table editor, do the following:
   a. Select the **Column** tab.
   b. Select the BONUSPOINT column and change the column type from NUMBER to INTEGER
   c. Save your changes.
7. In the Enterprise Beans pane, expand the **Bonus** bean. In the Tables pane, expand the **XBONUS** table.
8. Map the fields in the Bonus bean to the columns in the XBONUS table, by doing the following:
   a. Right-click the Bonus bean and select **Match By Name**.
9. Save the Map.mapxmi file (Ctrl+S) and close the file.
10. Oracle You must edit the table definition using a text editor, as follows:
    a. Open the XBONUS.xmi file with a text editor.
    b. Replace all occurances of `SQLNumeric6` to `SQLNumeric3`.
    c. Save your changes.

Express

1. Open the Data perspective and switch to the Data Definition view.
2. Navigate to the following directory: **WebSphereCommerceServerExtensionsData** > **ejbModule** > **META-INF**.
3. Right-click **META-INF** and select **New database definition**. The New Database Definition wizard opens.
4. In the **Database name** field, enter the name of your development database. For example, enter `Demo_Dev`.
5. From the **Database vendor type** drop-down list, select **DB2 Universal Database Express V8.1** and click **Finish**. The new database definition is now created.
6. Navigate to the following directory: **WebSphereCommerceServerExtensionsData** > **ejbModule** > **META-INF** > **Schema >** *DevelopmentDB*.
7. Right-click **DevelopmentDB** and select **New > New schema definition**. The New Schema Definition window displays.
8. In the **Schema name** field, enter `NULLID` and click **Finish**.
9. Navigate to the following directory: **WebSphereCommerceServerExtensionsData** > **ejbModule** > **META-INF** > **Schema >** *DevelopmentDB* **> NULLID** and select **New > New database definition**. The New Table Definition wizard opens.

10. In the **Table name** field, type XBONUS and click on **Next**.
11. Add the key column to your table definition, as follows:
    a. Click **Add Another** to add the MEMBERID column.
    b. In the **Column name** field, enter MEMBERID.
    c. Select **Key column**.
    d. From the **Column type** drop-down list, select the following:
       DB2 BIGINT
       Oracle NUMBER
12. Click **Add Another** to add the BONUSPOINT column.
13. In the **Column name** field, enter BONUSPOINT.
14. From the **Column type** drop-down list, select **INTEGER** and click **Finish**.
15. Switch to J2EE Perspective and in the J2EE Hierarchy view, select **EJB Modules** > **WebSphereCommerceServerExtensionsData**.
16. Right-click **WebSphereCommerceServerExtensionsData** and select **New** > **Access Bean**. The Add an Access Bean window displays.
17. Select **Copy helper** and click **Next**.
18. Select the **Bonus** bean and click **Next**.
19. From the **Constructor Method** selection box, select **findByPrimaryKey(com.ibm.commerce.extension.objects.Bonus)** and click **Finish**.
20. In the J2EE Hierarchy view, select **EJB Modules** > **WebSphereCommerceServerExtensionsData**.
21. Right-click **WebSphereCommerceServerExtensionsData** and select **Open With** > **Deployment Descriptor Editor**.
22. Scoll down to the JNDI - Default section and ensure that the value for **DataSource JNDI name** is blank. If the value is **jdbc/Default**, delete **jdbc/Default** and press Crtl+S to save your changes.
23. In the J2EE Hierarchy view, expand **EJB Modules**, then right-click **WebSphereCommerceServerExtensionsData** and select **Generate** > **Deployment and RMIC Code**.
24. Select the **Bonus** bean and click **Finish**.

The next step is to fix the schema name so that your new bean will be portable to other databases. To make this modification, do the following:

1. In the J2EE Perspective, switch to the J2EE Hierarchy view.
2. Expand **Databases**, then expand **WebSphereCommerceServerExtensionsData**.
3. Right-click on the schema node (for example, DB2USER) and select **Rename**.
4. Set the value to NULLID.

Once the BonusBean entity has been created and the schema is correctly mapped, you must create an access bean for the entity bean. This access bean makes it simpler for applications to access information contained in the Bonus entity bean. The tools in WebSphere Studio Application Developer are used to generate this access bean, based upon the entity bean that you have already created (in particular, only methods that have been promoted to the remote interface will be used by the access bean). To create the access bean for the Bonus entity bean, do the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**, then right-click **WebSphereCommerceServerExtensionsData** and select **New > Access Bean**.
   The Add an Access Bean window opens.
2. Select **Copy Helper** and click **Next**.
3. Select the **Bonus** bean and click **Next**.
4. From the Constructor method drop-down list, select **findByPrimaryKey(com.ibm.commerce.extension.objects.BonusKey)** as the constructor method.
5. Select all attributes in the Attribute Helpers section.
6. Click **Finish**.

You can view the newly generated code by switching to the Business Professional J2EE Navigator tab Express Project Navigator tab, by expanding the following:
WebSphereCommerceServerExtensionsData > ejbModule > com.ibm.commerce.extension.objects
A new class called BonusAccessBean and a new interface called BonusAccessBeanData are created and displayed inside the package.

The next step is to generate the deployed code.

The code generation utility analyzes the beans to ensure that Sun Microsystems' EJB specifications are met and it ensures that rules specific to the EJB server are followed. In addition, for each selected enterprise bean, the code-generation tool generates the home and EJBObject (remote) implementations and implementation classes for the home and remote interfaces, as well as the JDBC persister and finder classes for CMP beans. It also generates the Java ORB, stubs, and tie classes required for RMI access over IIOP, as well as stubs for the home and remote interfaces

To generate the deployed code, do the following: Business Professional

1. In the J2EE Hierarchy view, expand **EJB Modules**, then right-click **WebSphereCommerceServerExtensionsData** and select **Generate > Deploy and RMIC Code**.
2. Select the **Bonus** bean and click **Finish**.

Express

1. In the J2EE Hierarchy view, expand **EJB Modules** > **WebSphereCommerceServerExtensionsData**.
2. Right-click **WebSphereCommerceServerExtensionsData** and select **Open With** > **Deployment Descriptor Editor**.
3. Scoll down to the JNDI - Default section and ensure that the value for **DataSource JNDI name** is blank. If the value is **jdbc/Default**, delete **jdbc/Default** and press Crtl+S to save your changes.

You can view the newly generated code by switching to the Business Professional J2EE Navigator view Express Project Navigator view. You will find the following:

Table 11.

Type of code	Class name
Container implementation generated code	EJSCMPBonusHomeBean.java
	EJSRemoteCMPBonus.java
	EJSRemoteCMPBonusHome.java
	EJSFinderBonusBean.java
JDBC access code	EJSJDBCPersisterCMPBonusBean.java
RMI tie and stub code	_EJSRemoteCMPBonus_Tie.java
	_Bonus_Stub.java
	_EJSRemoteCMPBonusHome_Tie.java
	_BonusHome_Stub.java

The next step is to use the universal test client to test the new enterprise bean, by doing the following:

1. Switch to the Servers perspective.
2. In the Server Configuration view, double-click the **WebSphereCommerceServer** server and click the **Configuration** tab.
3. Select **Enable universal test client**. Save the changes.
4. In the Servers view, right-click the **WebSphereCommerceServer** server and select **Start**.
5. In the J2EE Hierarchy, expand **EJB Modules > WebSphereCommerceServerExtensionsData**.
6. Right-click the **Bonus** bean and select **Run on Server**. The IBM Universal Test Client opens.
7. In the left pane, click **Bonus**, then click **Bonus Home**.
8. Click the **Bonus create(Long, Integer)** method.

9. In the **Long** field in the right pane, enter `-1000` and in the **Integer** field, enter `1000`.
10. Click **Invoke** and the result is shown in the bottom pane.
11. Click **Work with Object** to add the remote interface to the Reference pane and you will see the values you entered under EJB Reference. A new record has been created in the BONUS table.
12. Select the **getMemberId** method and click **Invoke** and the result of -1000 is displayed in the bottom pane.
13. Close the test client and stop the server.

Integrating the Bonus entity bean with MyNewControllerCmd

In the previous section, you tested the new Bonus entity bean using the test client that was generated within WebSphere Studio Application Developer. In doing so, you determined that you can successfully update database information. Now, you integrate the Bonus entity bean with the MyNewControllerCmd logic. Once the Java code is updated, the MyNewJSPTemplate.jsp file is updated to create an interface that allows a customer's balance of bonus points to be updated.

Integrating the Bonus entity bean involves the following high-level steps:

1. Modifying the MyNewTaskCmd task command to include fields and methods for bonus points, updating the validateParameters method, and adding logic to update a user's balance of bonus points.
2. Adding a getResources method to the MyNewControllerCmdImpl class to return a list of the resources that the command uses. This method is included for access control purposes.
3. Creating a new BonusDataBean so that bonus points can be displayed in a JSP template.
4. Creating a new access control policy for the new resources.
5. Modifying the MyNewJSPTemplate.jsp template to allow you to enter bonus points for a user and then display that user's new balance of bonus points.

Modifying the MyNewTaskCmd interface to include bonus points

In this step, you modify the MyNewTaskCmd interface to specify the required fields and methods for bonus points, by doing the following:

1. Switch to the Java perspective and expand the **WebSphereCommerceServerExtensionsLogic** project.
2. Expand the **com.ibm.commerce.sample.commands\src** directory.
3. Double-click the **MyNewTaskCmd** interface to view its source code.
4. Uncomment Import section 2 to include the following package:

```
/// Import section 2 /////////////////////////////////////
import com.ibm.commerce.extension.objects.*;
/// End of import section 2 //////////////////////////////
```

5. Uncomment Section 4 to introduce the following code into the method:

```
/// Section 4 ///////////////////////////////////////////////

 public java.lang.Integer getOldBonusPoints();
 public Integer getTotalBonusPoints();

 public void setBonusAccessBean(BonusAccessBean bb);
 public BonusAccessBean getBonusAccessBean();


/// End of section 4/////////////////////////////////////////
```

6. Save the changes.

Modifying MyNewTaskCmdImpl to calculate bonus points

The MyNewTaskCmdImpl is used as the point of integration between the Bonus entity bean and the MyNewControllerCmd (since MyNewControllerCmd invokes the MyNewTaskCmd).

To modify MyNewTaskCmdImpl to calculate bonus points, do the following:

1. Select the **MyNewTaskCmdImpl** class to view its source code.
2. Uncomment Import section 2 to introduce the following package:

```
/// Import section 2 //////////////////////////////////////
import com.ibm.commerce.extension.objects.*;
///  End of Import section 2 ///////////////////////////////
```

3. Uncomment Sections 3A and 3B to introduce the following code into the class:

```
//// Section 3A /////////////////////////////////////////////

 private java.lang.Integer oldBonusPoints;
 private java.lang.Integer totalBonusPoints;


 private BonusAccessBean bb = null;

////End of Section 3A ////////////////////////////////////////

//// Section 3B /////////////////////////////////////////////

 public void setBonusAccessBean(BonusAccessBean newBB) {
  bb = newBB;
 }

 public BonusAccessBean getBonusAccessBean(){
  return bb;
 }

    public java.lang.Integer getOldBonusPoints() {
```

```
 return oldBonusPoints;
}

public Integer getTotalBonusPoints(){
 return totalBonusPoints;
}


/// End of section 3B ///////////////////////////////////////////
```

4. In the Outline view, select the **validateParameters** method and uncomment Section 2, to introduce the following code into the method:

```
// section 2 ////////////////////////////////////////////////////

  try {
    oldBonusPoints = bb.getBonusPoint();
  } catch (javax.ejb.FinderException e) {
  try {
   // If bb is null, create a new instance
    bb = new BonusAccessBean(new Long(foundUserId), new Integer(0));
    oldBonusPoints = new Integer(0);
   } catch (javax.ejb.CreateException ec) {
    throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
     this.getClass().getName(), "validateParameters");
   } catch (javax.naming.NamingException ec) {
    throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
     this.getClass().getName(), "validateParameters");
   } catch (java.rmi.RemoteException ec) {
    throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
     this.getClass().getName(), "validateParameters");
   }
  } catch (javax.naming.NamingException e) {
   throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
    this.getClass().getName(), "validateParameters");
  } catch (java.rmi.RemoteException e) {
   throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
    this.getClass().getName(), "validateParameters");
  } catch (javax.ejb.CreateException e) {
   throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
    this.getClass().getName(), "validateParameters");
  }

// end of section 2 ///////////////////////////////////////////////
```

5. In the Outline view, select the **performExecute** method.
6. In the source code for this performExecute method, uncomment Section 2. This introduces the following code into the method:

```
/// use BonusAccessBean to update new bonus point
/// Section 2 ///////////////////////////////////////////////////

 int newBP =  oldBonusPoints.intValue() + getInputPoints().intValue();
 totalBonusPoints = new Integer (newBP);
 bb.setBonusPoint(totalBonusPoints)   ;
```

```
    try {
      bb.commitCopyHelper();
    } catch (javax.ejb.FinderException e) {
     throw new ECSystemException(ECMessage._ERR_FINDER_EXCEPTION,
             this.getClass().getName(), "performExecute");
    } catch (javax.naming.NamingException e) {
     throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
             this.getClass().getName(), "performExecute");
    } catch (java.rmi.RemoteException e) {
     throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
             this.getClass().getName(), "performExecute");
    } catch (javax.ejb.CreateException e) {
     throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
             this.getClass().getName(), "performExecute");
    }

/// End of section 2 /////////////////////////////////////////////
```

7. Save your changes.

Adding a getResources method to the MyNewControllerCmdImpl class

In this section, you add a new getResources method to the MyNewControllerCmdImpl. This method returns a list of resources that the command uses during processing. This method is required for resource level access control.

To add the getResources method, do the following:

1. Double-click the **MyNewControllerCmdImpl** class to open and view its source code.
2. In the source code, uncomment Import Section 2 to introduce the following package into the class:

```
/// Import Section 2 ///////////////////////////////
import com.ibm.commerce.extension.objects.*;
/// End of Import Section 2 ////////////////////////
```

3. Uncomment Section 3, to introduce the following code into the class:

```
/// Section 3 //////////////////////////////////////////////////
/// Create an instance variable of type AccessVector to hold
/// the resources and a BonusAccessBean instance variable for
/// access control purposes.

    private AccessVector resources = null;
    private BonusAccessBean bb = null;

/// End of Section 3 ///////////////////////////////////////////
```

4. In the source code, uncomment the Access Control Section. This section appears as shown in the following code snippet:

```
/// AccessControl Section /////////////////////////////////////

public AccessVector getResources() throws ECException{
```

```
if (resources == null ) {

   /// use UserRegistryAccessBean to check user reference number

String refNum = null;
String methodName = "getResources";

 rrb = new UserRegistryAccessBean();

 try {
  rrb = rrb.findByUserLogonId(getUserName());
  refNum = rrb.getUserId();
 } catch (javax.ejb.FinderException e) {
  throw new ECSystemException(ECMessage._ERR_FINDER_EXCEPTION,
       this.getClass().getName(),methodName,e);
 } catch (javax.naming.NamingException e) {
  throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
       this.getClass().getName(), methodName,e);
 } catch (java.rmi.RemoteException e) {
  throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
       this.getClass().getName(), methodName,e);
 } catch (javax.ejb.CreateException e) {
  throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
       this.getClass().getName(), methodName,e);
 }


/// find the bonus bean for this registered user

   bb = new com.ibm.commerce.extension.objects.BonusAccessBean();
 try {
  if (refNum != null) {
   bb.setInitKey_memberId(new Long(refNum));
   bb.refreshCopyHelper();
   resources = new AccessVector(bb);
  }
 } catch (javax.ejb.FinderException e) {

  ///doesn't have a bonus object so return the container that
    ///will hold the bonus object when it's created
 resources = new AccessVector(rrb);
 return resources;

 } catch (javax.naming.NamingException e) {
  throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
       this.getClass().getName(), methodName);
 } catch (java.rmi.RemoteException e) {
  throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
       this.getClass().getName(), methodName);
 } catch (javax.ejb.CreateException e) {
  throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
       this.getClass().getName(), methodName);
 }

}
```

```
     return resources;
    }

    /// End of AccessControl Section //////////////////////////////////
```

5. Save your changes.

Modifying the performExecute method of the MyNewControllerCmdImpl class

In this step you modify the code in the performExecute method of the MyNewControllerCmdImpl to include code related to the new bonus bean, as follows:

1. Double-click the **MyNewControllerCmdImpl** class.
2. In the Outline view, select the **performExecute** method.
3. In the source code, uncomment Sections 4E, 4G, and 4H to introduce the following code into the method:

```
// Section 4E ////////////////////////////////////
/// pass bb instance variable to the task command
    cmd.setBonusAccessBean(bb);
// End of section 4E /////////////////////////////

// Section 4G ///////////////////////////////////////
if (cmd.getOldBonusPoints() != null) {
      rspProp.put("oldBonusPoints", cmd.getOldBonusPoints());
}
// End of section 4G /////////////////////////////

// Section 4H ///////////////////////////////////////
///Instantiate the bonus data bean , then put it to response properties
   BonusDataBean bdb  =
      new com.ibm.commerce.sample.databeans.BonusDataBean(
         cmd.getBonusAccessBean());
  rspProp.put("bdbInstance", bdb );
// End of section 4H ////////////////////////////////
```

4. Save your changes.

Note: You will see errors because the BonusDataBean has not yet been defined. These will be fixed in the next section.

Creating the BonusDataBean data bean

In keeping with the programming model, you should create a new data bean that corresponds to the new Bonus entity bean. While not all entity beans are required to have a corresponding data bean, if you want to be able to display information from the entity bean in a JSP template, you should create a new data bean for this purpose.

In this scenario, you are required to create a new BonusDataBean data bean that extends the BonusAccessBean. As with other parts of the tutorial, the base code is provided and you need to uncomment various sections of code.

To introduce the BonusDataBean into your code, do the following:

1. The first step is to import the base code for the new data bean, as follows:
   a. Switch to the Business Professional J2EE Navigator view Express Project Navigator view in the Java Perspective.
   b. Expand the **WebSphereCommerceServerExtensionsLogic** project.
   c. Right-click the **src** folder and select **Import**.
      The Import wizard opens.
   d. From the **Select an import source** list, select **Zip file** and click **Next**.
   e. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located, as follows:
      yourDirectory\WC_SAMPLE_55.zip
      where *yourDirectory* is the directory into which you downloaded the package.
   f. Click **Deselect All**, then expand the directories and select the following file to import.
      - com\ibm\commerce\sample\databeans\BonusDataBean.java
   g. In the **Folder** field, the `WebSphereCommerceServerExtensionsLogic/src` folder is already specified. Keep this value.
   h. Click **Finish**.
2. Double click the **BonusDataBean** class to view its source code.
3. In the source code, uncomment Section 1, to introduce the following code into the bean:

```
/// Section 1 ///////////////////////////////////////////////

// create fields and accessors (setter/getter methods)

 private java.lang.String userId;
 private java.lang.Integer totalBonusPoints;


 public java.lang.String getUserId() {
  return userId;
 }

 public void setUserId(java.lang.String newUserId) {
  userId = newUserId;

  ///////////////////////////////////////
  /// Section A : instantiate BonusAccessbean

  if (userId != null)
   this.setInitKey_memberId(new Long(newUserId));

  ///////////////////////////////////////
 }
```

```
   public java.lang.Integer getTotalBonusPoints() {
 return totalBonusPoints;
 }
 public void setTotalBonusPoints(java.lang.Integer newTotalBonusPoints) {
  totalBonusPoints= newTotalBonusPoints;
 }


//// End of section 1 ////////////////////////////////////////////
```

4. Next, uncomment Section 2, to introduce the following section of code into the bean:

```
/// Section 2///////////////////////////////////////////////

// create a new constructor for passing access bean into databean
// so that JSP can work with the access bean

public BonusDataBean(BonusAccessBean bb)
   throws com.ibm.commerce.exception.ECException {
 try {
  super.setEJBRef(bb.getEJBRef());
 }  catch (javax.ejb.FinderException e) {
     throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
    "BonusDataBean", "BonusDataBean(bb)");
  } catch (javax.naming.NamingException e) {
   throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
           "BonusDataBean", "BonusDataBean(bb)");
  } catch (java.rmi.RemoteException e) {
   throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
           "BonusDataBean", "BonusDataBean(bb)");
  } catch (javax.ejb.CreateException e) {
   throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
           "BonusDataBean", "BonusDataBean(bb)");
  }
}

//// End of section 2 ////////////////////////////////////////////
```

5. Next, uncomment Section 3, to introduce the following code into the bean:

```
//// Section 3 ///////////////////////////////////////////////

// set additional data field that is used for instantiating BonusAccessbean

 try
 {

  setUserId(getRequestProperties().getString("taskOutputUserId"));

  try {
   super.refreshCopyHelper();
  } catch (javax.ejb.FinderException e) {
     throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
    "BonusDataBean", "populate");
  } catch (javax.naming.NamingException e) {
```

```
    throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
            "BonusDataBean", "populate");
   } catch (java.rmi.RemoteException e) {
    throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
            "BonusDataBean", "populate");
   } catch (javax.ejb.CreateException e) {
    throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
            "BonusDataBean", "populate");
   }


  }
  catch (ParameterNotFoundException e){}


 ///// End of Section 3 /////////////////////////////////////////
 }
```

6. Next, uncomment Section 4 to introduce the following code into the bean:

```
/// Section 4 ////////////////////////////////////////////

// copy input TypedProperteis to local

 requestProperties = aParam;

/// End of section 4 /////////////////////////////////////
```

7. Save your changes.
8. Compile the changed code by right-clicking the **WebSphereCommerceServerExtensionsLogic** project and selecting **Rebuild Project**.

Creating the access control policy for the new entity bean

A sample access control policy is provided. This policy creates the following access control objects:

An action
: The action that is created is `com.ibm.commerce.sample.commands.MyNewControllerCmd`

An action group
: The action group that is created is `MyNewControllerCmdActionGroup`. This action group contains only one action; `com.ibm.commerce.sample.commands.MyNewControllerCmd`

A resource category
: The resource category that is created is `com.ibm.commerce.sample.objects.BonusResourceCategory`. This resource category is for the Bonus entity bean.

A resource group
: The resource group that is created is `BonusResourceGroup`. This resource group only contains the preceding resource category.

A policy

The policy that is created is `AllUsersUpdateBonusResourceGroup`. This policy allows users to perform the MyNewControllerCmd action on the Bonus bean only if the user is the "owner" of the bonus object. For example, if the user is logged on as the `tester@mycompany` user, the user can only modify their own bonus points.

Setting up the `AllUsersUpdateBonusResourceGroup` policy involves the following steps:

1. Modifying the access control policies to reflect your environment.
2. Loading the SampleACPolicy.xml file using the `acpload` command.
3. Loading the SampleACPolicy_en_US.xml description using the `acpnlsload` command.

To customize the access control policies for your environment, do the following:

1. Determine the member ID value for your FashionFlow store, as follows:
   - DB2 If you are using a DB2 database, do the following:
     a. Connect to your development database.
     b. Issue the following SQL statement:
        ```
        select member_id from storeent where storeent_id=FF_storeent_ID
        ```
        where *FF_storeent_ID* is the store entity ID for your FashionFlow store. For example, you could enter:
        ```
        select member_id from storeent where storeent_id=10001
        ```
        Make note of the member ID value: ______________________
   - Oracle If you are using an Oracle database, do the following:
     a. Open the Oracle SQL Plus command window and connect to your development database.
     b. Enter the following:
        `column member_id format 99999999999999999999999999999999999` (thiry-five 9's)
        This sets the display of the member_id column wide enough to view the entire value.
     c. Issue the following SQL statement:
        ```
        select member_id from storeent where storeent_id=FF_storeent_ID
        ```
        where *FF_storeent_ID* is the store entity ID for your FashionFlow store. For example, you could enter:
        ```
        select member_id from storeent where storeent_id=10001
        ```

Make note of the member ID value: ______________________

d. Reset the display width of the member_id column by entering the following:

```
column member_id clear
```

2. Using a text editor, open the SampleACPolicy_template.xml file (in the *WCDE_installdir*\Commerce\xml\policies\xml directory) and replace *FashionFlowMemberId* with the member ID value for your Fashion Flow store, as determined in step 1. Save your modified file as SampleACPolicy.xml (within the same directory).
3. Using a text editor, open the SampleACPolicy_template_en_US.xml file (in the *WCDE_installdir*\Commerce\xml\policies\xml directory) and replace *FashionFlowMemberId* with the member ID value for your Fashion Flow store, as determined in step 1. Save your modified file as SampleACPolicy_en_US.xml (within the same directory).
4. At a command prompt, switch to the following directory: *WCDE_installdir*\commerce\bin
5. To load the SampleACPolicy.xml file, you must issue the acpload command, which has the following form:

```
acpload db_name db_user db_password inputXMLFile
```

where

- *db_name* is the name of your database
- *db_user* is your database user name
- *db_password* is your database password
- *inputXMLFile* is the name of the XML file containing the policy. In this case, enter `SampleACPolicy.xml`

For example, you may issue the following command:

```
acpload Demo_dev db2user db2user SampleACPolicy.xml
```

6. To load the policy description, you must issue the acpnlsload command, which has the following form:

```
acpnlsload db_name db_user db_password inputXMLFile
```

For example, you may issue the following command:

```
acpnlsload Demo_dev db2user db2user SampleACPolicy_en_US.xml
```

7. If the server for the test environment is currently running, you can use the registry refresh option in the WebSphere Commerce Administration Console to update the access control registry, as follows:

a. Open a Web browser and enter the following URL:

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

b. When prompted, log in using a site administrator ID.

c. Select to work on the **Site** and click **OK**.
d. From the **Configuration** menu, select **Registry**.
e. Click **Update All**, then after a moment, click **Refresh** to verify that the updates have been made.
f. Logout and close the Administration Console windows.

Modifying the MyNewJSPTemplate.jsp template to include bonus points

To modify the display page, do the following:

1. In WebSphere Studio Application Developer, switch to the Web perspective.
2. Open both the **MyNewJSPTemplate_All.jsp** and **MyNewJSPTemplate.jsp** files.
3. Copy Section 9 from the MyNewJSPTemplate_All.jsp file into the MyNewJSPTemplate.jsp file. This introduces the following text into the JSP template:

```
<!-- SECTION 9 -->

<h2><fmt:message key="BonusAdmin" bundle="${tutorial}" /> </h2>

<c:if test="${!empty taskOutputUserId}">
 <ul>
  <li>
   <b>
   <fmt:message key="PointBeforeUpdate" bundle="${tutorial}" />
   <c:out value="${oldBonusPoints}"/>
   </b>
  </li>
  <li>
   <b>
   <fmt:message key="PointAfterUpdate" bundle="${tutorial}" />
      <c:out value="${bdbInstance.bonusPoint}" />
      </b>
     </li>
 </ul>
</c:if>


<br />
<b><fmt:message key="EnterPoint" bundle="${tutorial}" /></b><p />


<form name="Bonus" action="MyNewControllerCmd">
<table>
 <tr>
  <td>
   <b>Logon ID </b>
  </td>
  <td>
   <input type="text" name="input1" value="<c:out
    value="${userName}"/>" />
```

```
    </td>
   </tr>
   <tr>
    <td>
     <b>Bonus Point</b>
    </td>
    <td>
     <input type="text" name="input2" />
    </td>
   </tr>
   <tr>
    <td colspan="2">
     <input type="submit" />
    </td>
   </tr>
  </table>
  </form>

  <!-- END OF SECTION 9 -->
```

4. Save the MyNewJSPTemplate.jsp file.

Testing the integrated Bonus bean

Since the new Bonus bean is protected under access control and users can only execute the MyNewControllerCmd action on a bean that they own, the user must log in. As such, you will use the login feature in your sample store to allow the user to log in.

To test the new logic, do the following:

1. Switch to the Server view.
2. Right-click the **WebSpherCommerceServer** server and select **Start** (or **Restart**).
3. Right-click the **index.jsp** file for your store and select **Run on Server**. The store home page is displayed.
4. Log on as a registered user, by doing the following:
   a. Click the **Register** link. The Register page is displayed.
   b. In the **E-mail address** field, enter the e-mail address for the user that you created in “Testing user name validation” on page 262.
   c. In the **Password** field, enter the password for this user and then click **Login**.
   d. Once the login has completed, enter the following URL in the same browser:

      ```
      http://localhost/webapp/wcs/stores/servlet/MyNewControllerCmd?
         input1=user_e-mail&input2=1000
      ```

      where *user_e-mail* is the e-mail address for the user that you created in “Testing user name validation” on page 262. You are presented with a

page that contains all of the previous output parameters as well as a new form that allows you to update the balance of bonus points for the user.

Bonus Administration

- The bonus point before update is 3000
- The bonus point after update is 4000

Please enter the points, then submit it to the controller command

Logon ID

Bonus Point

Submit Query

Figure 41.

e. Now, into the **Logon ID** field enter the user's e-mail address and in the **Bonus Point** field, enter `500`. Click **Submit**. You are presented with a page similar to the following that shows that the balance of bonus points is updated.

Bonus Administration

- The bonus point before update is 4000
- The bonus point after update is 4500

Please enter the points, then submit it to the controller command

Logon ID

Bonus Point

Submit Query

Figure 42.

Deploying the bonus points logic

This section describes how to deploy your new business logic into a store running on a remote WebSphere Commerce Server. You must have created a store (based upon the FashionFlow sample store) on the remote WebSphere Commerce Server before starting these deployment steps.

The deployment process includes steps that are performed on the development machine, as well as steps that are performed on the target WebSphere Commerce Server.

There are a number of different types of assets that must be deployed to the target WebSphere Commerce Server. These include:

- Controller command, task command and data bean logic
- Enterprise bean logic
- A JSP template and image file
- A properties file and resource bundles
- Database updates including schema updates (new table) as well as command registry updates
- Access control updates

This section describes how to deploy all of these assets, *incrementally* to the target WebSphere Commerce Server. This is done as an incremental deployment, in contrast to a deployment of the entire EAR file.

Creating the command and data bean JAR file

This section describes how to create the JAR file that contains the controller command, task command, and data bean logic.

To create this JAR file, perform the following steps on your development machine:

1. Create a directory on your local file system called *drive*:\ExportTemp.
2. In WebSphere Studio Application Developer, switch to the Business Professional J2EE Navigator view Express Project Navigator view.
3. Right-click the **WebSphereCommerceServerExtensionsLogic** project and select **Export**.
   The Export wizard opens.
4. In the Export wizard, do the following:
   a. Select **JAR file** and click **Next**.
   b. The left pane under **Select the resources to export** is prepopulated with the name of the project. Leave this value as is.
   c. In the right pane ensure that only the following resources are selected:
      - .classpath

- .project
- .serverPreference

d. Ensure that **Export generated class files and resources** is selected.

e. Do *not* select **Export Java source files and resources**.

f. In the **Select the export destination** field, enter the fully-qualified JAR file name to use. In this case, enter `*drive*:\ExportTemp\WebSphereCommerceServerExtensionsLogic.jar`. Note that the JAR file name must be WebSphereCommerceServerExtensionsLogic.jar.

g. Click **Finish**.

Creating the EJB JAR file

To create the EJB JAR file, do the following:

1. Open WebSphere Studio Application Developer and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the **WebSphereCommerceServerExtensionsData** project.
3. Double-click **EJB Deployment Descriptor**.
4. With the Overview tab selected, scroll to the bottom of the pane, to locate the **WebSphere Bindings** section.
5. In the **DataSource JNDI name** field, enter the datasource JNDI name of the target WebSphere Commerce Server. The following is an example value:

   DB2 `jdbc/WebSphere Commerce DB2 DataSource demo`
   where the target WebSphere Commerce Server is using a DB2 database, and the WebSphere Commerce instance name is "demo"

   Oracle `jdbc/WebSphere Commerce Oracle DataSource demo`
   where the target WebSphere Commerce Server is using an Oracle database, and the WebSphere Commerce instance name is "demo".

The value for the DataSource JNDI name is created by adding "`jdbc/`" to the data source name of the target WebSphere Commerce Server. You can verify the data source name by opening the *instanceName*.xml file on the target WebSphere Commerce Server and searching for `DatasourceName=` in the file.

---

6. Save your deployment descriptor changes (Ctrl+S).
7. In the Business Professional J2EE Navigator view Express Project Navigator view, right-click the **WebSphereCommerceServerExtensionsData** project and select **Export**.
   The Export wizard opens.
8. In the Export wizard, do the following:

   a. Select **EJB JAR file** and click **Next**.

b. Business Professional The value for **What resources do you want to export?** is prepopulated with the name of the EJB project.

Express The EJB project name is prepopulate.
Leave this value as is.

c. Business Professional In the **Where do you want to export resources to?** field, enter the fully-qualified JAR file name to use.

Express For the destination, enter the fully-qualified JAR file name to use.
In this case, enter
`drive:\ExportTemp\WebSphereCommerceServerExtensionsData.jar`.

d. Click **Finish**.

9. After the JAR file has been created, undo the changes to the local deployment descriptor that was made in step 5, to restore the setting that is required for your local test server.

Note: This tutorial assumes that your development database and the database used by the target WebSphere Commerce Server are the same type. If you were deploying to a different database type, you would follow the instructions contained in "Creating an EJB JAR file with conversion" on page 204.

Exporting store assets

To export the store assets, do the following:

1. Switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the **Stores** folder.
3. Right-click the **Web Content** folder and select **Export**.
   The Export Wizard opens.
4. In the Export wizard, do the following:
   a. Select **File system** and click **Next**.
   b. Click **Deselect All**.
   c. Select to export the following resources:
      - Web Content*FashionFlow_name*\MyNewJSPTemplate.jsp
      - Web Content*FashionFlow_name*\images\male_blueshirt.gif
      - Web Content\WEB-INF\classes*FashionFlow_name*\Tutorial_NLS_en_US.properties
      - Web Content\WEB-INF\lib\jstl.jar
      - Web Content\WEB-INF\lib\standard.jar
   d. Select **Create directory structure for files**.

e. In the Directory field, enter a temporary directory into which these resources will be placed. For example, enter `C:\ExportTemp`
f. Click **Finish**.

Packaging access control policies

In this section, you copy the access control policies that were created for your new resources into the *drive*:\ExportTemp directory, as follows:

1. Navigate to the *WCDE_installdir*\Commerce\xml\policies\xml directory.
2. Copy the following files into the *drive*:\ExportTemp\ACPolicies directory:
   - MyNewViewACPolicy.xml
   - MyNewControllerCmdACPolicy.xml
   - SampleACPolicy_template.xml
   - SampleACPolicy_template_en_US.xml

Transferring assets to your target WebSphere Commerce Server

In this step, you create a temporary directory on the target WebSphere Commerce Server and then copy your bonus point assets into this directory. In subsequent steps, you will place the different types of code into the appropriate place within your WebSphere Commerce application.

To copy the files from your development machine to your target WebSphere Commerce Server, do the following:

1. On the target WebSphere Commerce Server, create a temporary directory called *drive*:\ImportTemp.
2. Determine how you will copy your files from one computer to another. You can do this by mapping a drive on the target WebSphere Commerce Server to the development machine, or by using an FTP application, if you have that configured.
3. From the development machine, copy the contents of *drive*:\ExportTemp into *drive*:\ImportTemp on the target WebSphere Commerce Server.

Stopping your target WebSphere Commerce Server

Before starting the deployment steps, you should stop your target WebSphere Commerce Server.For details about stopping the WebSphere Commerce Server, refer to the Business Professional *WebSphere Commerce Studio Installation Guide* or Express *WebSphere Commerce - Express Developer Edition Installation Guide*.

Updating the database on your target WebSphere Commerce Server

Before updating the target database, verify the store entity ID for the store to which you are deploying the customized logic. The following SQL statement can be used to determine this value:

```
select STOREENT_ID from STOREENT where IDENTITY='FashionFlow_name'
```

where *FashionFlow_name* is the name of the store to which you are deploying the code.

Registering the view

DB2 If you are using a DB2 database, do the following to register MyNewView:

1. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Line Tools > Command Center**).
2. From the **Tools** menu, select **Tools Settings**.
3. Select the **Use statement termination character** check box and ensure the character specified is a semicolon (**;**)
4. Close the tools settings.
5. With the Script tab selected, create the required entry in the VIEWREG table, by entering the following information in the script window:

```
connect to targetDB user dbuser using dbpassword;
insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID, INTERFACENAME,
   CLASSNAME, PROPERTIES, DESCRIPTION, HTTPS, LASTUPDATE)
values ('MyNewView',-1, FF_storeent_ID,
   'com.ibm.commerce.command.ForwardViewCommand',
   'com.ibm.commerce.command.HttpForwardViewCommandImpl',
   'docname=MyNewJSPTemplate.jsp','This is my new view for tutorial 1',
   0, null);
```

   where
   - *targetDB* is the name of the target database
   - *dbuser* is the database user
   - *dbpassword* is the password of the database user
   - *FF_storeent_ID* is the unique identifier for your store that is based on the FashionFlow sample store

   Click the **Execute** icon. Keep the Command Center open.

Oracle If you are using an Oracle database, do the following to register your view in the database:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

```
insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID, INTERFACENAME,
   CLASSNAME, PROPERTIES, DESCRIPTION, HTTPS, LASTUPDATE)
values ('MyNewView',-1, FF_storeent_ID,
```

```
'com.ibm.commerce.command.ForwardViewCommand',
'com.ibm.commerce.command.HttpForwardViewCommandImpl',
'docname=MyNewJSPTemplate.jsp','This is my new view for tutorial 1',
0, null);
```

where

- *FF_storeent_ID* is the unique identifier for your store that is based on the FashionFlow sample store.

Press Enter to run the SQL statement.

6. Enter the following to commit your database changes:

```
commit;
```

and press Enter to run the SQL statement.

MyNewView is now registered.

Registering the new controller command

To register MyNewControllerCmd, do the following:

1. DB2 If you are using a DB2 database, do the following to register MyNewControllerCmd:
   a. In the Command Center, with the Script tab selected, create the required entry in the URLREG table, by entering the following information in the script window:

```
insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS, DESCRIPTION,
   AUTHENTICATED) values ('MyNewControllerCmd',FF_storeent_ID,
   'com.ibm.commerce.sample.commands.MyNewControllerCmd',
   0, 'This is a new controller command for tutorial one.',null);
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION,
   CLASSNAME, TARGET)
values (FF_storeent_ID,
   'com.ibm.commerce.sample.commands.MyNewControllerCmd',
   'This is a new controller command for tutorial one.',
   'com.ibm.commerce.sample.commands.MyNewControllerCmdImpl',
   'local');
```

   where

   - *targetDB* is the name of the target database
   - *dbuser* is the database user
   - *dbpassword* is the password of the database user
   - *FF_storeent_ID* is the unique identifier for your store that is based on the FashionFlow sample store.

   Click the **Execute** icon. Keep the Command Center open.

2. Oracle If you are using an Oracle database, do the following to register MyNewControllerCmd:

a. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
b. In the **User Name** field, enter your Oracle user name.
c. In the **Password** field, enter your Oracle password.
d. In the **Host String** field, enter your connect string.
e. In the SQL Plus window, enter the following SQL statement:

```
insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS, DESCRIPTION,
   AUTHENTICATED) values ('MyNewControllerCmd',FF_storeent_ID,
   'com.ibm.commerce.sample.commands.MyNewControllerCmd',
   0, 'This is a new controller command for tutorial one.',null);
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION,
   CLASSNAME, TARGET)
values (FF_storeent_ID,
   'com.ibm.commerce.sample.commands.MyNewControllerCmd',
   'This is a new controller command for tutorial one.',
   'com.ibm.commerce.sample.commands.MyNewControllerCmdImpl','local');
```

where

- *FF_storeent_ID* is the unique identifier for your store that is based on the FashionFlow sample store.

Press Enter to run the SQL statement.

f. Enter the following to commit your database changes:

```
commit;
```

and press Enter to run the SQL statement.

Creating the XBONUS table

In this step, you create the XBONUS table in the database used by your target WebSphere Commerce Server.

DB2 If you are using a DB2 database, do the following to create the table:

1. In the Script window, enter the following:

```
create table XBONUS (MEMBERID BIGINT NOT NULL,
    BONUSPOINT INTEGER NOT NULL, constraint p_xbonus
           primary key (MEMBERID),
    constraint f_xbonus foreign key (MEMBERID)
    references users (users_id) on delete cascade)
```

where

- *targetDB* is the name of the target database
- *dbuser* is the database user
- *dbpassword* is the password of the database user

Click the Execute icon.
The XBONUS table is now created.

Oracle If you are using an Oracle database, do the following to create the table:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

   ```
   create table XBONUS (MEMBERID NUMBER NOT NULL,
       BONUSPOINT INTEGER NOT NULL, constraint p_xbonus
              primary key (MEMBERID),
       constraint f_xbonus foreign key (MEMBERID)
       references users (users_id) on delete cascade);
   ```

   and press Enter to run the SQL statement. The XBONUS table is now created.
6. Enter the following to commit your database changes:

   ```
   commit;
   ```

   and press Enter to run the SQL statement.

Loading access control policies on your target WebSphere Commerce Server

In this step, you load the access control policies for your new resources onto your target WebSphere Commerce Server.

To load the new policy, do the following:

1. Update the access control policies to reflect the values specific to your target WebSphere Commerce Server, as follows:
   a. Determine the member ID value for your FashionFlow store, as follows:
      - DB2 If you are using a DB2 database, do the following:
        1) Connect to your development database.
        2) Issue the following SQL statement:

           ```
           select member_id from storeent where storeent_id=FF_storeent_ID
           ```

           where *FF_storeent_ID* is the store entity ID for your FashionFlow store. For example, you could enter:

           ```
           select member_id from storeent where storeent_id=10001
           ```

           Make note of the member ID value: ________________________
      - Oracle If you are using an Oracle database, do the following:

1) Open the Oracle SQL Plus command window and connect to your development database.
2) Enter the following:
`column member_id format 99999999999999999999999999999999999` (thiry-five 9's)
This sets the display of the member_id column wide enough to view the entire value.
3) Issue the following SQL statement:

```
select member_id from storeent where storeent_id=FF_storeent_ID
```

where *FF_storeent_ID* is the store entity ID for your FashionFlow store. For example, you could enter:

```
select member_id from storeent where storeent_id=10001
```

Make note of the member ID value: ______________________
4) Reset the display width of the member_id column by entering the following:

```
column member_id clear
```

b. Navigate to the following directory: *drive*:\ImportTemp\ACPolicies
c. Using a text editor, open the SampleACPolicy_template.xml file and replace *FashionFlowMemberId* with the member ID value for your Fashion Flow store, as determined in step 1a. Save your modified file as SampleACPolicy.xml.
d. Using a text editor, open the SampleACPolicy_template_en_US.xml file and replace *FashionFlowMemberId* with the member ID value for your Fashion Flow store, as determined in step 1a. Save your modified file as SampleACPolicy_en_US.xml.

2. Copy the following access control policies into the *WC_installdir*\xml\policies\xml directory:
   - MyNewViewACPolicy.xml
   - MyNewControllerCmdACPolicy.xml
   - SampleACPolicy.xml
   - SampleACPolicy_en_US.xml
3. At a command prompt, navigate to the following directory: *WC_installdir*\bin
4. You must issue the acpload command, which has the following form:

```
acpload targetDB dbuser dbpassword inputXMLFile
```

where
   - *targetDB* is the name of your development database.
   - *dbuser* is the name of the database user.

- *dbpassword* is the password for your database user.
- *inputXMLFile* is the XML file containing the access control policy specification. In this case, specify `MyNewViewACPolicy.xml`.

The following is an example of the command, with variables specified:

```
acpload Demo_Dev db2admin db2admin MyNewViewACPolicy.xml
```

5. Repeat step 4 for each of the following access control policies:
   - MyNewControllerCmdACPolicy.xml
   - SampleACPolicy.xml
6. To load the policy description (which is contained in the SampleACPolicy_en_US.xml file), you must issue the acpnlsload command, which has the following form:

   ```
   acpnlsload db_name db_user db_password inputXMLFile
   ```

   For example, you may issue the following command:

   ```
   acpnlsload Demo_dev user password SampleACPolicy_en_US.xml
   ```
7. Check the *WC_installdir*\xml\policies\xml directory for any error logs that might have been generated while the access control policies were loaded.

Updating store assets on your target WebSphere Commerce Server

In this step, you update the store with your modified store assets, as follows:

1. Backup your *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear\Stores.war directory (where *cellName* is often the host name of your machine and *instanceName* is the name of your WebSphere Commerce instance).
2. Navigate to the *drive*:\ImportTemp\Stores\Web Content directory.
3. Copy the *FashionFlow_name* and WEB-INF folders into the following directory: *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear\Stores.war

Updating the command and data bean JAR file on your target WebSphere Commerce Server

In this step you update the target WebSphere Commerce Server to use the new command and data bean JAR file, as follows:

1. You should make a backup copy of the existing JAR file, as follows:
   a. Navigate to the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory.
   b. Make a copy of the WebSphereCommerceServerExtensionsLogic.jar file and save it in a backup location.
2. Copy the new WebSphereCommerceServerExtensionsLogic.jar file from the *drive*:\ImportTemp directory into the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory.

where *instanceName* is the name of your WebSphere Commerce instance.

Updating the EJB JAR file on your target WebSphere Commerce Server

In this step you update the target WebSphere Commerce Server to use the new EJB JAR file, as follows:

1. You should make a backup copy of the existing JAR file, as follows:
   a. Navigate to the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory.
   b. Make a copy of the WebSphereCommerceServerExtensionsData.jar file and save it in a backup location.
2. Copy the new WebSphereCommerceServerExtensionsData.jar file from the *drive*:\ImportTemp directory into the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory
3. Next, you must modify the EJB deployment descriptor information, as follows:
   a. Locate the deployment repository (META-INF directory) for this WebSphere Application Server cell. This typically takes the following form:
      WAS_installdir\config\cells*cellName*
      \applications\WC_*instance_name*.ear\deployments\
      WC_*instance_name**EJBModuleName*.jar\META-INF.
      The following is a specific example of this:
      D:\WebSphere\AppServer\config\cells\myCell\applications\
      WC_demo.ear\deployments\WC_demo\
      WebSphereCommerceServerExtensionsData.jar\META-INF
   b. The directory contains the following files:
      - ejb-jar.xml
      - ibm-ejb-access-bean.xmi
      - ibm-ejb-jar-bnd.xmi
      - ibm-ejb-jar-ext.xmi
      - MANIFEST.MF

      Backup all of these files.
   c. Use a tool to open the new WebSphereCommerceServerExtensionsData.jar file and view its contents.
   d. Extract the contents of the meta-inf directory (the preceding listed files) from this WebSphereCommerceServerExtensionsData.jar file into the directory from step 3a. Ensure that the directory structure remains correct after you have extracted the files.
4. Using the WebSphere Application Server startServer command at the command line, restart your WebSphere Commerce instance. Refer to the *WebSphere Commerce Installation Guide* for your platform and database for more information about starting and stopping this instance.

Verifying bonus points logic on the target WebSphere Commerce Server

In this step, you verify that the bonus points logic has been successfully deployed to the target WebSphere Commerce Server, by doing the following:

1. Open a Web browser and enter the URL to launch your store that is based on the FashionFlow sample store.
2. Create a new registered user, by doing the following:
   a. Click **Register**.
   b. Click **Register** again to create a new customer.
   c. In the registration form, enter appropriate values into all of the mandatory fields. For example, in the e-mail field, enter `tester@mycompany`. Make note of the value for the e-mail address: ________________________.
   d. Once the values have been entered, click **Submit**.
3. Once the login has completed, enter the following URL in the same browser:

   ```
   http://hostname/webapp/wcs/stores/servlet/MyNewControllerCmd?
      input1=user_e-mail&input2=1000
   ```

   where *hostname* is the host name and *user_e-mail* is the e-mail address for the user that you created in step 2. You are presented with a page that contains all of the previous output parameters as well as a new form that allows you to update the balance of bonus points for the user.

Figure 43.

4. Now, into the **Logon ID** field enter the user's e-mail address and in the **Bonus Point** field, enter `500`. Click **Submit**. You are presented with a page

similar to the following that shows that the balance of bonus points points is updated.

Bonus Administration

- The bonus point before update is 4000
- The bonus point after update is 4500

Please enter the points, then submit it to the controller command

Logon ID [ ]

Bonus Point [ ]

Submit Query

Figure 44.

Chapter 11. Tutorial: Modifying an existing controller command

The purpose of this tutorial is to demonstrate the process used to modify an existing controller command.

In this tutorial, you restrict the number of items in a customer's shopping cart to be five or less. To implement this solution, you override the OrderItemAddCmdImpl with your own implementation that includes logic to check for the number of items in the cart. If a customer attempts to add a sixth item to the shopping cart, an exception is thrown. This exception uses a new error message.

Note that the purpose of this tutorial is to demonstrate the development process used for modifying existing command logic. It is not meant to be an all encompassing example of restricting items in the shopping cart. The logic used in this tutorial is simplified for the sake of the tutorial.

In this tutorial, you will learn the following:

- How to create a new implementation for an existing controller command
- How to update the command registry so that the new implementation gets used in your application
- How to deploy a modified controller command to an existing WebSphere Commerce application

Prerequisites

This tutorial does not require that you have completed Chapter 10, "Tutorial: Creating new business logic," on page 217. If you have completed that tutorial, there is no harm in leaving the code in your workspace, as it will not conflict with this tutorial.

Before starting this tutorial, you must have published a store based upon the FashionFlow sample store. Within this store, you must be able to complete a purchase (for example, browse the catalog, add items to the shopping cart, checkout and see the order confirmation).

In order to complete the deployment steps, the store must also exist on the target WebSphere Commerce Server.

© Copyright IBM Corp. 2000, 2003

Creating the new MyOrderItemAddCmdImpl class

In this step in the tutorial, you create a new MyOrderItemAddCmdImpl class. To create this class, do the following:

1. In WebSphere Studio Application Developer, open the Java perspective (**Window > Open Perspective > Java**).
2. Navigate to the **WebSphereCommerceServerExtensionsLogic** project.
3. Navigate to the **src** directory.
4. If you have not completed Chapter 10, "Tutorial: Creating new business logic," on page 217, right-click the **src** directory and select **New > Package**. The New Java Package wizard opens. In this wizard, do the following:
   a. In the **Name** field, enter `com.ibm.commerce.sample.commands`.
   b. Click **Finish**.
5. Right-click the **com.ibm.commerce.sample.commands** package. Select **New > Class**.
   The New Java Class wizard opens.
6. In the New Java Class wizard, do the following:
   a. In the **Name** field, enter `MyOrderItemAddCmdImpl`.
   b. To specify the superclass, click the **Browse** button beside the Superclass field and then enter `OrderItemAddCmdImpl`. Click **OK**.
   c. To specify which interface to implement, click **Add**, then enter `OrderItemAddCmd` and click **OK**.
   d. Click **Finish**.

The source code for the MyOrderItemAddCmdImpl class is displayed.

The next step is to update the import statements in this class, as follows:

1. In the Outline view, select and open **import declarations**. You will find that there are two import statements already created.
2. Into the source code for the import statements, add the following import statements:

```
import com.ibm.commerce.exception.ECApplicationException;
import com.ibm.commerce.exception.ECException;
import com.ibm.commerce.exception.ECSystemException;
import com.ibm.commerce.order.objects.OrderAccessBean;
import com.ibm.commerce.ras.ECMessage;
import com.ibm.commerce.sample.messages.MyNewMessages;
```

3. Save your changes.

The next step is to add the business logic and exception handling to determine if there are more than five items in the customer's shopping cart before adding any more order items. Update the code, as follows:

1. In the Outline view, select the MyOrderItemAddCmdImpl class and view its source code. It currently only has the following:

```
public class MyOrderItemAddCmdImpl
 extends OrderItemAddCmdImpl
 implements OrderItemAddCmd {

}
```

2. You must add a new performExecute method to this class. This method contains the logic to check the number of items in the shopping cart, and if the number is less than five, the regular performExecute method of the superclass (OrderItemAddCmdImpl) will be called, as normal. If there are five or more items, an exception is thrown and the user cannot add more items to the cart. To add this method, copy the following source code into the class (ensure that it is included before the last closing brace "}" denoting the end of the class):

```
public void performExecute() throws ECException {
  // Get a list of order ids
  String[] orderIds = getOrderId();

  // Check to make sure that an id exists at all
  // if order id exists then get number of items in the order
  // else if no order id exists then execute normal code
  if (orderIds != null && orderIds.length > 0) {
   // An exception should be thrown when trying to add a sixth item
   // to the cart.  Since this code is run before any items are added
   // throw and exception if there are 5 or more items in the cart
   if (itemsInOrder(orderIds[0]) >= 5) {
    throw new ECApplicationException(
     MyNewMessages._ERR_TOO_MANY_ITEMS,
     this.getClass().getName(),
     "performExecute");
   }
   //else perform normal flow
  }
  super.performExecute();
 }
 //get number of items in the order
 protected int itemsInOrder(String orderId) throws ECException {
  try {
   OrderAccessBean order = new OrderAccessBean();
   order.setInitKey_orderId(orderId);
   order.refreshCopyHelper();
   return order.getOrderItems().length;
  } catch (javax.ejb.FinderException e) {
   throw new ECSystemException(
    ECMessage._ERR_FINDER_EXCEPTION,
    this.getClass().getName(),
    "itemsInOrder");
  } catch (javax.naming.NamingException e) {
   throw new ECSystemException(
    ECMessage._ERR_NAMING_EXCEPTION,
    this.getClass().getName(),
```

```
      "itemsInOrder");
    } catch (java.rmi.RemoteException e) {
     throw new ECSystemException(
      ECMessage._ERR_REMOTE_EXCEPTION,
      this.getClass().getName(),
      "itemsInOrder");
    } catch (javax.ejb.CreateException e) {
     throw new ECSystemException(
      ECMessage._ERR_CREATE_EXCEPTION,
      this.getClass().getName(),
      "itemsInOrder");
    }
   }
```

After you have pasted the new code into the class, right-click in the source code and select **Format** to format the code.

3. Save your work.

   Note: There are warnings indicating that message information is missing. This will be corrected in subsequent steps.

Creating message information

The new command implementation uses a new error message: `_ERR_TOO_MANY_ITEMS`. In this section, you create the code for that new message and its associated properties file. To import this code, do the following:

1. Expand the **WebSphereCommerceServerExtensionsLogic** project.
2. Right-click the **src** directory and select **New > Package**.
   The New Java Package wizard opens. In this wizard, do the following:
   a. In the **Name** field, enter `com.ibm.commerce.sample.messages`.
   b. Click **Finish**.
3. Right-click the **com.ibm.commerce.sample.messages** package. Select **New > Class**.
   The New Java Class wizard opens.
4. In the New Java Class wizard, do the following:
   a. In the **Name** field, enter `MyNewMessages`.
   b. Click **Finish**.
5. Double-click the **MyNewMessages** class to view its source code.
6. Immediately preceding the `public class MyNewMessages` line of code, add the following import statements:

```
import com.ibm.commerce.ras.ECMessage;
import com.ibm.commerce.ras.ECMessageSeverity;
import com.ibm.commerce.ras.ECMessageType;
```

7. Within the class, add the following code:

```
// Resouce bundle used to extract the text for an exception
 static final String errorBundle = "MyNewErrorMessages";

 // An ECMessage is used to describe an ECException and is passed
 // into the ECException when thrown
 public static final ECMessage _ERR_TOO_MANY_ITEMS =
      new ECMessage(ECMessageSeverity.ERROR, ECMessageType.USER,
      MyNewMessageKeys._ERR_TOO_MANY_ITEMS, errorBundle);
```

8. Save your changes.
9. Right-click the **com.ibm.commerce.sample.messages** package. Select **New > Class**.
   The New Java Class wizard opens.
10. In the New Java Class wizard, do the following:
    a. In the **Name** field, enter `MyNewMessageKeys`.
    b. Click **Finish**.
11. Define the code in the class to be the following:

```
public class MyNewMessageKeys {
// This class defines the keys used to create new exceptions that are
// thrown by customized code.
 public static final String _ERR_TOO_MANY_ITEMS = "_ERR_TOO_MANY_ITEMS";
}
```

12. Save your changes.
13. Compile your code, by right-clicking the **WebSphereCommerceServerExtensionsLogic** project and selecting **Build Project**.
14. Create the new properties file that will contain the message information, as follows:
    a. Open the Web perspective (**Window > Open Perspective > Web**).
    b. Within the **Stores** Web project, expand the **Web Content > WEB-INF > classes** folders.
    c. Right-click the **classes** folder and select **New > Other > Simple > File > Next** to create a new properties file.
       The New File window opens.
    d. In the **File name** field, enter `MyNewErrorMessages.properties`, then click **Finish**.
       The new empty file opens.
    e. Into the new file, copy the following text:

```
_ERR_TOO_MANY_ITEMS=You are trying to place too many different items
   in one shopping cart.
```

       Note: The line break in the preceding code is for presentation purposes only. Enter your text on a single line.
    f. Save your changes.

Modifying the command registry

In this step, you modify the command registry so that the new MyOrderItemAddCmdImpl implementation class gets used instead of the original OrderItemAddCmdImpl implementation class. The only table in the command registry that needs to be modified is the CMDREG table. In this case, the new implementation class is used for all stores.

To modify the command registry, do the following:

1. DB2 If you are using a DB2 database, do the following to register MyOrderItemAddCmdImpl:
   a. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Center**).
   b. From the **Tools** menu, select **Tools Settings**.
   c. Select the **Use statement termination character** check box and ensure the character specified is a semicolon (**;**)
   d. With the Script tab selected, create the required entry in the URLREG table, by entering the following information in the script window:

   ```
   connect to developmentDB user dbuser using dbpassword;
   update CMDREG
   set CLASSNAME='com.ibm.commerce.sample.commands.MyOrderItemAddCmdImpl'
   WHERE INTERFACENAME=
   'com.ibm.commerce.orderitems.commands.OrderItemAddCmd'
   and storeent_Id=0;
   ```

   where
   - *developmentDB* is the name of your development database
   - *dbuser* is the database user
   - *dbpassword* is the password for your database user

   Click the **Execute** icon.
2. Oracle If you are using an Oracle database, do the following to register MyOrderItemAddCmdImpl:
   a. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
   b. In the **User Name** field, enter your Oracle user name.
   c. In the **Password** field, enter your Oracle password.
   d. In the **Host String** field, enter your connect string.
   e. In the SQL Plus window, enter the following SQL statement:

   ```
   update CMDREG
   set CLASSNAME='com.ibm.commerce.sample.commands.MyOrderItemAddCmdImpl'
   WHERE INTERFACENAME=
   'com.ibm.commerce.orderitems.commands.OrderItemAddCmd'
   and storeent_Id=0;
   ```

Press Enter to run the SQL statement.

f. Enter the following to commit your database changes:

```
commit;
```

and press Enter to run the SQL statement.

Testing the MyOrderItemAddCmdImpl command

The next step is to test to ensure the new logic is functioning well. To test this, you should be able to successfully add five items to the shopping cart, but expect an error to be thrown when you attempt to add a sixth item to the cart.

To test your new business logic, do the following:

1. Switch to the Server perspective (**Window > Open Perspective > Server**).
2. Right-click the **WebSphereCommerceServer** server and select **Start** (or **Restart**).
3. Right-click **index.jsp** under the Stores\Web Content*FashionFlow_name* directory and select **Run on Server**.
The store home page is displayed in the Web browser.
4. Shop through the store and add five items to the shopping cart. After adding the fifth item, you will have a shopping cart that is similar to the following:

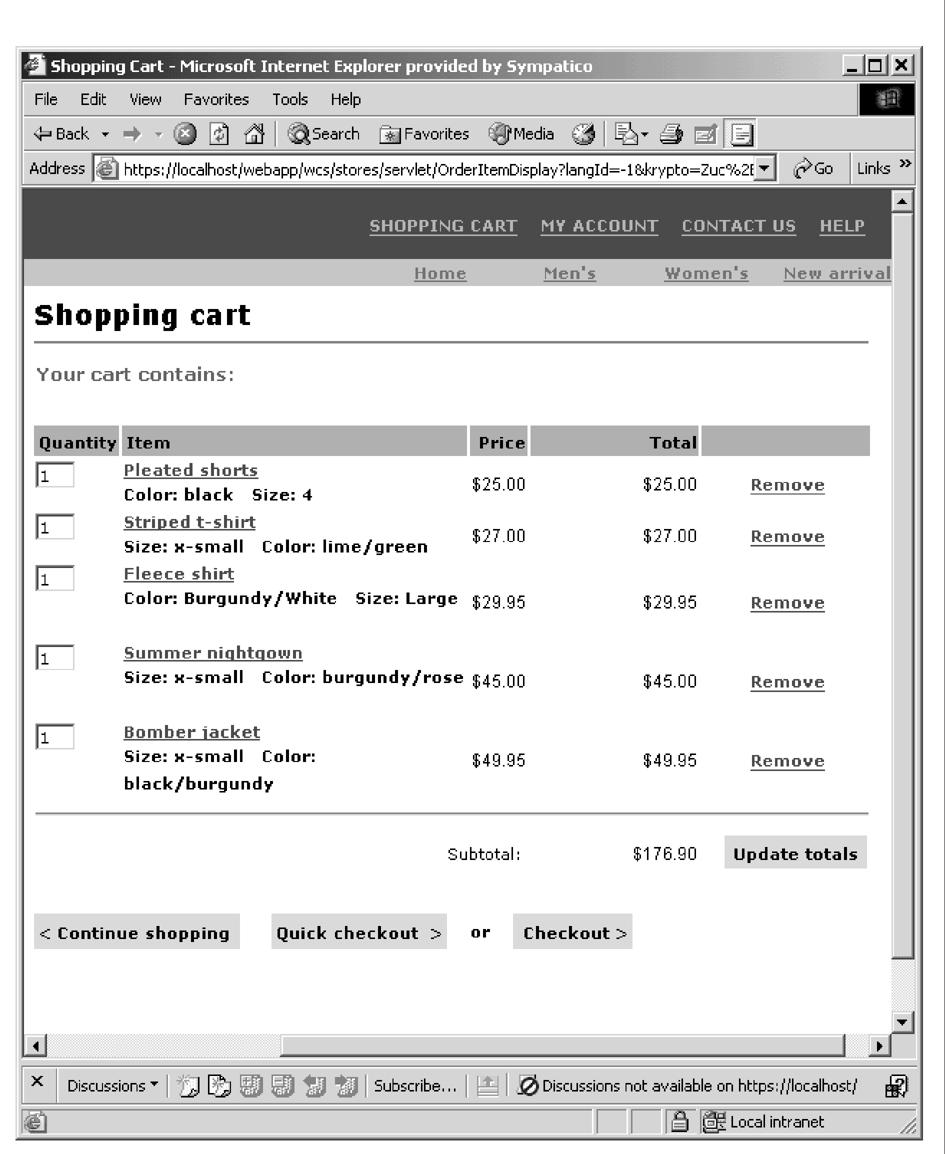


Figure 45.

5. Click **Continue Shopping** and select another item. For this selected item, click **Add to shopping cart**. You are presented with the following error page:

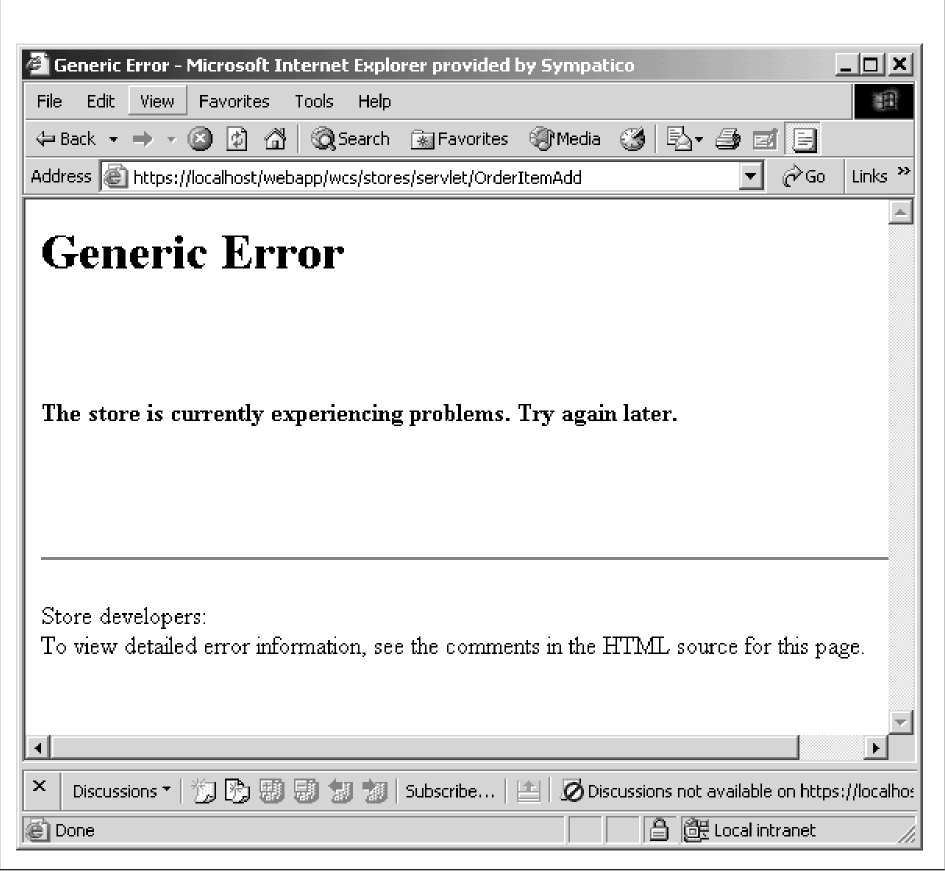


Figure 46.

View the source for the error page. In the source, scroll down to find the following error information:

```
Message Key: _ERR_TOO_MANY_ITEMS
Message: You are trying to place too many different items in one shopping
         cart
```

Deploying MyOrderItemAddCmdImpl

In this step, you deploy the modified business logic to a target WebSphere Commerce Server. In this case, deployment consists of the following high-level steps:

1. Creating the JAR file that contains the command logic and error classes.
2. Exporting the error message properties file.
3. Transferring the assets to the target WebSphere Commerce Server.

4. Updating the command registry on the target WebSphere Commerce Server.
5. Validating the new logic on the target WebSphere Commerce Server.

Creating the command JAR file

Following the WebSphere Commerce code customization strategy, customized commands and data beans are placed in the WebSphereCommerceServerExtensionsLogic project. As a result, when it comes time to deploy customized code, you will notice that you include previously customized code in your JAR file. For example, if you have completed the Chapter 10, "Tutorial: Creating new business logic," on page 217, you will notice that when you create the JAR file to deploy the MyOrderItemAddCmdImpl class, you include the previously created MyNewControllerCmd and other classes.

To create the JAR file that contains the MyOrderItemAddCmdImpl class, perform the following steps on your development machine:

1. Create a directory on your local file system called *drive*:\ExportTemp2.
2. Open WebSphere Studio Application Developer and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
3. Right-click the **WebSphereCommerceServerExtensionsLogic** project and select **Export**.
   The Export wizard opens.
4. In the Export wizard, do the following:
   a. Select **JAR file** and click **Next**.
   b. The left pane under **Select the resources to export** is prepopulated with the name of the project. Leave this value as is.
   c. In the right pane ensure that only the following resources are selected:
      - .classpath
      - .project
      - .serverPreference
   d. Ensure that **Export generated class files and resources** is selected.
   e. Do *not* select **Export Java source files and resources**.
   f. In the **Select the export destination** field, enter the fully-qualified JAR file name to use. In this case, enter `drive:\ExportTemp2\WebSphereCommerceServerExtensionsLogic.jar`. Note that the JAR file name must be WebSphereCommerceServerExtensionsLogic.jar.
   g. Click **Finish**.

Exporting the message properties file

In this section, you export the properties file that contains the text for the new message, as follows:

1. Switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the **Stores** folder.
3. Right-click the **Web Content** folder and select **Export**.
   The Export Wizard opens.
4. In the Export wizard, do the following:
   a. Select **File system** and click **Next**.
   b. Click **Deselect All**.
   c. Select to export the following resource:
      Web Content\WEB-INF\classes\MyNewErrorMessages.properties
   d. Select **Create directory structure for files**.
   e. In the Directory field, enter a temporary directory into which these resources will be placed. For example, enter `C:\ExportTemp2`
   f. Click **Finish**.

Transferring assets to the target WebSphere Commerce Server

In this step, you create a temporary directory on the target WebSphere Commerce Server and then copy your MyOrderItemAddCmdImpl assets into this directory.

To copy the files from your development machine to your target WebSphere Commerce Server, do the following:

1. On the target WebSphere Commerce Server, create a temporary directory called *drive*:\ImportTemp2.
2. Determine how you will copy your files from one computer to another. You can do this by mapping a drive on the target WebSphere Commerce Server to the development machine, or by using an FTP application, if you have that configured.
3. From the development machine, copy the contents of *drive*:\ExportTemp2 into *drive*:\ImportTemp2 on the target WebSphere Commerce Server.

Stopping your target WebSphere Commerce Server

Before starting the deployment steps, you should stop your target WebSphere Commerce Server. For details about stopping your target WebSphere Commerce Server, refer to the Business Professional *WebSphere Commerce Studio Installation Guide* or Express *WebSphere Commerce - Express Developer Edition Installation Guide*.

Updating the database on the target WebSphere Commerce Server

In this step, you modify the command registry so that it will use your new MyOrderItemAddCmdImpl implementation class.

DB2 If you are using a DB2 database, do the following to register MyOrderItemAddCmdImpl:

1. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Center**).
2. From the **Tools** menu, select **Tools Settings**.
3. Select the **Use statement termination character** check box and ensure the character specified is a semicolon (**;**)
4. Close the tools settings.
5. With the Script tab selected, create the required entry in the CMDREG table, by entering the following information in the script window:

```
connect to targetDB user dbuser using dbpassword;
update CMDREG
set CLASSNAME='com.ibm.commerce.sample.commands.MyOrderItemAddCmdImpl'
WHERE INTERFACENAME=
'com.ibm.commerce.orderitems.commands.OrderItemAddCmd'
and storeent_Id=0;
```

   where
   - *targetDB* is the name of the database used by your target WebSphere Commerce Server
   - *dbuser* is the database user
   - *dbpassword* is the database password

   Click the **Execute** icon.

Oracle If you are using an Oracle database, do the following to register MyOrderItemAddCmdImpl:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

```
update CMDREG
set CLASSNAME='com.ibm.commerce.sample.commands.MyOrderItemAddCmdImpl'
WHERE INTERFACENAME=
'com.ibm.commerce.orderitems.commands.OrderItemAddCmd'
and storeent_Id=0;
```

   Press Enter to run the SQL statement.

6. Enter the following to commit your database changes:
```
commit;
```
and press Enter to run the SQL statement.

Updating the command JAR file on the target WebSphere Commerce Server

In this step you update the target WebSphere Commerce Server to use the JAR file that contains the new MyOrderItemAddCmdImpl, as follows:

1. Using the WebSphere Application Server stopServer command at a command line, stop your WebSphere Commerce instance. Refer to the *WebSphere Commerce Installation Guide* for your platform and database for information about starting and stopping this instance, if required.
2. You should make a backup copy of the existing JAR file, as follows:
   a. Navigate to the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory, where *cellName* is usually the host name of the machine
   b. Make a copy of the WebSphereCommerceServerExtensionsLogic.jar file and save it in a backup location.
3. Copy the new WebSphereCommerceServerExtensionsLogic.jar file from the *drive*:\ImportTemp2 directory into the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory.

where *instanceName* is the name of your WebSphere Commerce instance.

Updating the message properties on the target WebSphere Commerce Server

In this step, you add the new message properties file to the application, as follows:

1. Backup your *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear\Stores.war directory (where *cellName* is often the host name of your machine and *instanceName* is the name of your WebSphere Commerce instance).
2. Navigate to the *drive*:\ImportTemp\Stores\Web Content directory.
3. Copy the WEB-INF folder into the following directory: *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear\Stores.war
4. Using the WebSphere Application Server startServer command at a command line, restart your WebSphere Commerce instance.

where *instanceName* is the name of your WebSphere Commerce instance.

Verifying the MyOrderItemAddCmdImpl logic on the target WebSphere Commerce Server

In this step, you perform a quick check to verify that the code is working well, once it has been deployed.

To verify the code, do the following:

1. Open a Web browser and launch your FashionFlow store. For example, enter the following URL to launch the store:

   ```
   http://hostname/webapp/wcs/stores/servlet/FashionFlow/index.jsp
   ```

   where *hostname* is the host name for your instance.
2. Shop through the store and add five items to the shopping cart. After adding the fifth item, you will have a shopping cart that is similar to the following:

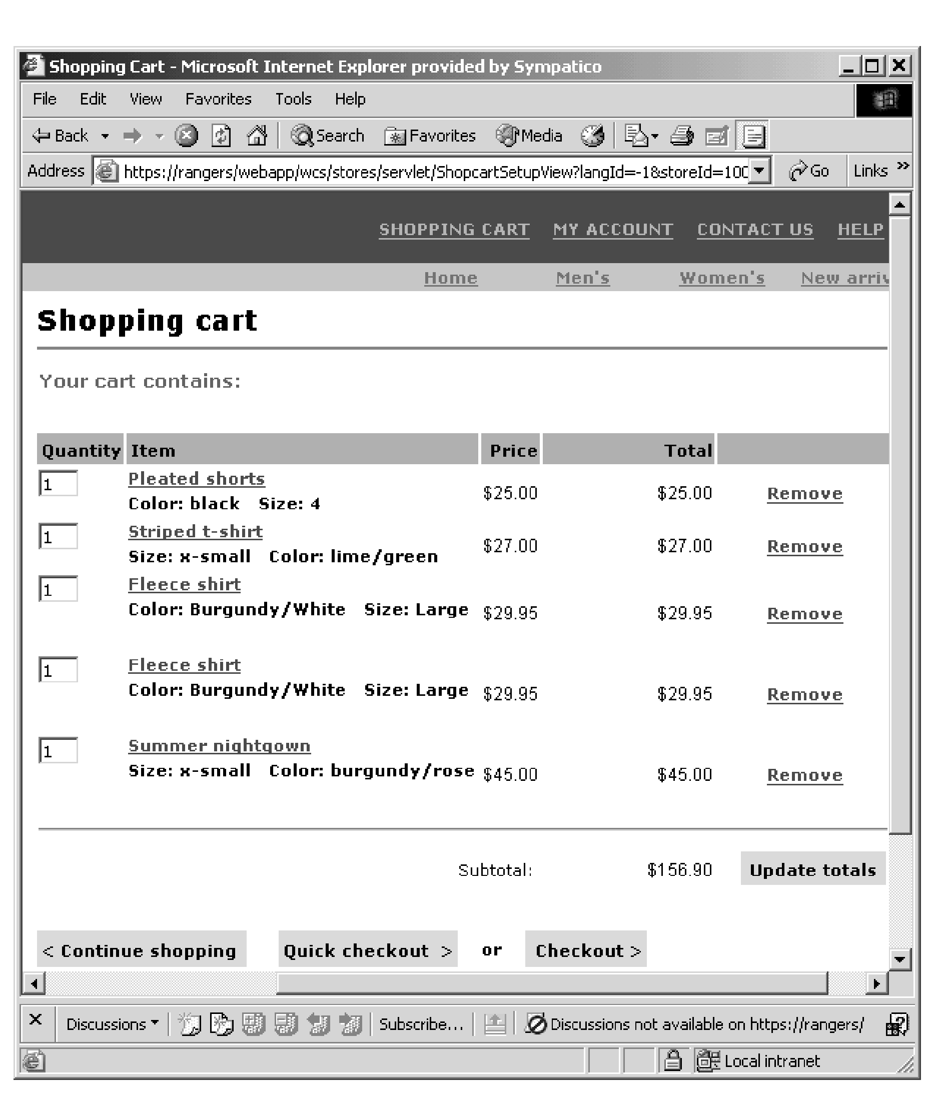


Figure 47.

3. Click **Continue Shopping** and select another item. For this selected item, click **Add to shopping cart**. You are presented with the following error page:

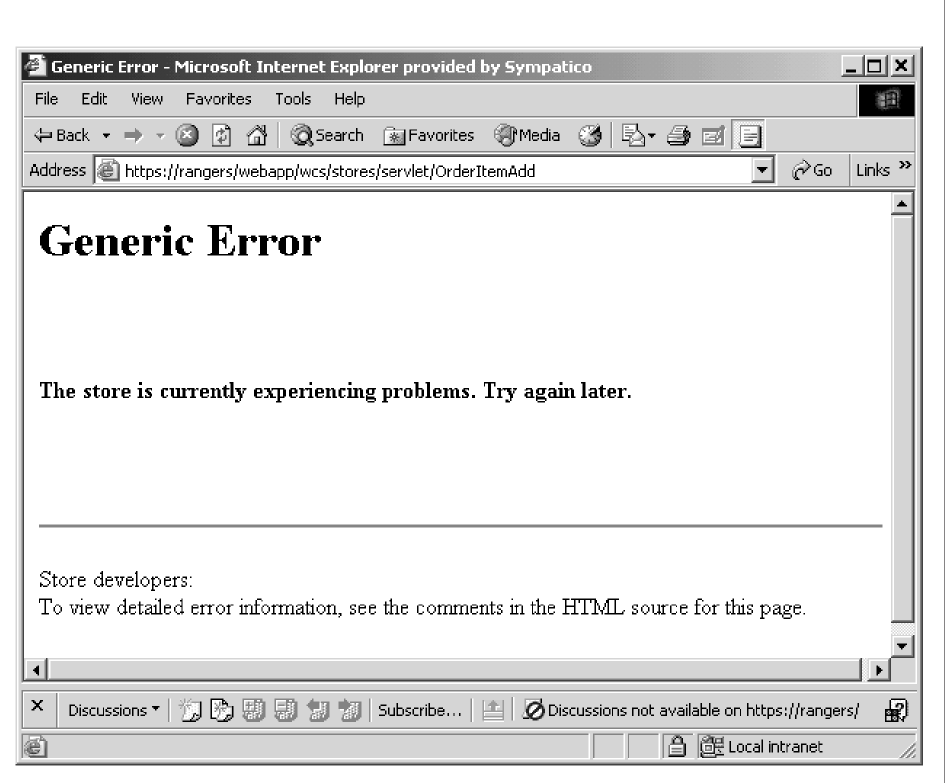


Figure 48.

View the source for the error page. Scroll down in the source and you will find that the message key for the error is _ERR_TOO_MANY_ITEMS.

Chapter 12. Tutorial: Extending the object model and modifying an existing task command

In this tutorial, you address a requirement to collect gift card information for orders. The information that must be collected includes the name of the recipient, the name of the sender and two messages. The information is to be collected when the customer submits the order.

As the information for the gift card must be stored in the database, it is clear that a new database table is required. Since this is an extension to the order process, there are two possible ways that the object model can be modified. Typically, you could either modify an existing WebSphere Commerce public entity bean and map new fields in the modified bean to the new table, or you can create a new entity bean that maps directly to the new table. As this approach would require you to extend the Order entity bean and this bean uses a ″find for update″ type of SQL query, you cannot extend the bean in this manner. As such, this is a case in which you must create a new entity bean that maps to the new table.

In addition to the new entity bean, the existing ExtOrderProcessCmdImpl task command is extended. The extension is used to instantiate a new data bean that corresponds to the new table and it is used to update the gift information in the database.

This tutorial includes the following high-level tasks:

1. Creating and populating the new XORDGIFT table
2. Creating the new OrderGift entity bean
3. Creating the XORDGIFT schema
4. Creating the table definition and mapping the fields in the OrderGift entity bean to the columns in the XORDGIFT table
5. Generating the deployed code and access bean for the OrderGift bean
6. Creating the new OrderGiftDataBean
7. Creating the new MyExtOrderProcessCmdImpl task command implementation
8. Modifying the OrderSubmitForm.jsp to collect the message information and modifying the OrderDetailDisplayForm.jsp to display the message information.
9. Testing the modified code

© Copyright IBM Corp. 2000, 2003

Prerequisites

This tutorial does not require that you have completed tutorials. If you have completed those tutorials, there is no harm in leaving the code in your workspace, as it will not conflict with this tutorial.

Before starting this tutorial, you must have published a store based upon the FashionFlow sample store. Within this store, you must be able to complete a purchase (for example, browse the catalog, add items to the shopping cart, checkout and see the order confirmation).

Creating and populating the XORDGIFT table

In this step, you create the XORDGIFT table.

DB2 If you are using a DB2 database, do the following to create the table:

1. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Line Tools > Command Center**) and click the **Script** tab.
2. In the Script window, enter the following:

```
connect to developmentDB user dbuser using dbpassword;
create table XORDGIFT (ORDERSID bigint not null, RECEIPTNAME varchar(50),
SENDERNAME varchar(50), MSGFIELD1 varchar(50), MSGFIELD2 varchar(50),
constraint p_xordgift primary key (ORDERSID),
constraint f_xordgift foreign key (ORDERSID) references ORDERS(ORDERS_ID)
on delete cascade);
insert into XORDGIFT (ORDERSID) (select ORDERS_ID from ORDERS);
```

   where
   - *developmentDB* is the name of your development database
   - *dbuser* is the database user
   - *dbpassword* is the password for your database user

   Click the Execute icon.
   The XORDGIFT table is now created.

Oracle If you are using an Oracle database, do the following to create the table:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statements:

```
create table XORDGIFT (ORDERSID NUMBER NOT NULL, RECEIPTNAME VARCHAR(50),
SENDERNAME VARCHAR(50), MSGFIELD1 VARCHAR(50), MSGFIELD2 VARCHAR(50),
constraint p_xordgift primary key (ORDERSID),
constraint f_xordgift foreign key (ORDERSID) references ORDERS(ORDERS_ID)
on delete cascade);
insert into XORDGIFT (ORDERSID) (select ORDERS_ID from ORDERS);
```

and press Enter to run the SQL statement. The XORDGIFT table is now created.

Note: You must issue the following command before creating the XORDGIFT table, if anyone has previously run this example using this database:

```
drop table XORDGIFT;
```

6. Enter the following to commit your database changes:

```
commit;
```

and press Enter to run the SQL statement.

Creating the OrderGift entity bean

Once the database table has been created, you are ready to begin creating the new entity bean. The next steps use WebSphere Studio Application Developer to create this bean.

Next you create the new OrderGift bean, by doing the following:

1. Start WebSphere Commerce development environment as follows:
   - Business Professional **Start > Programs > IBM WebSphere Commerce Studio > WebSphere Commerce development environment**
   - Express Express **Start > Programs > IBM WebSphere - Express Developer Edition > WebSphere Commerce development environment**
2. Open the J2EE Perspective.
3. Within the J2EE Hierarchy view, expand **EJB Modules**.
4. Right-click the **WebSphereCommerceServerExtensionsData** module and select **New > Enterprise Bean**.
   The Enterprise Bean Creation wizard opens.
5. From the **EJB Project** drop-down list, select **WebSphereCommerceServerExtensionsData** and click **Next**.
6. In the Create an Enterprise Bean window, do the following:
   a. Select **Entity bean with container-managed persistence (CMP) fields**
   b. In the **Bean name** field, enter `OrderGift`.
   c. In the **Source folder** field, leave the default value that is specified (ejbModule).

d. In the **Default package** field, enter `com.ibm.commerce.extension.objects`.
e. Click **Next**.

7. In the CMP Attributes window, do the following:
   a. Click **Add** to add new fields for the following columns in the table:
      - ORDERSID
      - RECEIPTNAME
      - SENDERNAME
      - MSGFIELD1
      - MSGFIELD2

      The Create CMP Attribute window opens. In this window, do the following:
      1) Create the ordersId field, as follows:

Table 12.

Parameter Name	Parameter Value
Name	`ordersId`
Type	`java.lang.Long` **Note:** You must use the *java.lang.Long* data type, not the *long* data type.
Key Field	Select

Click **Apply**.

2) Create the receiptName field, as follows:

Table 13.

Parameter Name	Parameter Value
Name	`receiptName`
Type	`java.lang.String`
Access with getter and setter methods	Select
Promote getter and setter methods to remote interface	Clear

Click **Apply**.

3) Create the senderName field, as follows:

Table 14.

Parameter Name	Parameter Value
Name	`senderName`
Type	`java.lang.String`

Table 14. (continued)

Parameter Name	Parameter Value
Access with getter and setter methods	Select
Promote getter and setter methods to remote interface	Clear

Click **Apply**.

4) Create the msgField1 field, as follows:

Table 15.

Parameter Name	Parameter Value
Name	`msgField1`
Type	`java.lang.String`
Access with getter and setter methods	Select
Promote getter and setter methods to remote interface	Clear

Click **Apply**.

5) Create the msgField2 field, as follows:

Table 16.

Parameter Name	Parameter Value
Name	`msgField2`
Type	`java.lang.String`
Access with getter and setter methods	Select
Promote getter and setter methods to remote interface	Clear

Click **Apply**.

6) Click **Close** to close this window.

b. Clear the **Use the single key attribute type for the key class** check box, then click **Next**.

8. In the EJB Java Class Details window, do the following:
   a. To select the bean's superclass, click **Browse**.
      The Type Selection window opens.
   b. In the **Select a class using: (any)** field, enter `ECEntityBean` and click **OK**. This selects the com.ibm.commerce.base.objects.ECEntityBean as the superclass.
   c. Click **Finish**.

Set the isolation level for the new bean, by doing the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**.
2. Double-click the **WebSphereCommerceServerExtensionsData** project to open it with the Deployment Descriptor Editor.
3. Click the **Access** tab.
4. Click **Add** next to the Isolation Level text box.
   The Add Isolation Level window opens.
5. DB2 Select **Repeatable Read**, then click **Next**.

   * DB2 400 Select **Read Committed**, then click **Next**.

   Oracle Select **Read Committed**, then click **Next**
6. Select the **OrderGift** bean, then click **Next**.
7. Select **OrderGift** to select all of its methods, and click **Finish**.
8. Save your work (Ctrl+S).

Next set the security identity of the bean, by doing the following:
1. In the Deployment Descriptor editor, select the Access tab.
2. Click **Add** next to the Security Identity (Method level) text box.
   The Add Security Identity window opens.
3. Select **Use identity of EJB server**, then click **Next**.
4. Select the **OrderGift** bean, then click **Next**.
5. Select **OrderGift** to select all of its methods, and click **Finish**.
6. Save your work (Ctrl+S) and keep the editor open.

Next, set the security role for the methods in the bean, by doing the following:
1. In the Deployment Descriptor editor, select the Assembly Descriptor tab.
2. In the Method permissions section, click **Add**.
3. Select **WCSecurityRole** as the security role and click **Next**.
4. From the list of beans found, select **OrderGift** and click **Next**.
5. In the Method elements page, click **Apply to All**, then click **Finish**.
6. Save your work (Ctrl+S) and close the deployment descriptor editor.

The next step is to remove the fields and methods related to the entity context that are generated by WebSphere Studio Application Developer. The reason that these fields need to be deleted is that the ECEntityBean base class provides its own implementation of these methods. To delete the generated entity context fields and methods, do the following:
1. In the J2EE Hierarchy view, expand the **WebSphereCommerceServerExtensionsData** project.
2. Expand the **OrderGift** EJB module, then double-click **OrderGiftBean**.
3. In the Outline view, do the following:

Note: Express WebSphere Studio Application Developer 5.1 prompts you to delete getEntityContext() and setEntityContext(EntityContext), so you do not have to delete them manually as mentioned below. Ensure that you select you delete these.

a. Right-click the **myEntityCtx** field and select **Delete**.
b. Right-click the **getEntityContext()** method and select **Delete**.
c. Right-click the **setEntityContext(EntityContext)** method and select **Delete**.
d. Right-click the **unsetEntityContext()** method and select **Delete**.

4. Save your work (Ctrl+S).

You should modify the generated ebjCreate method so that all of the parameters are explicitly set when a new OrderGift bean is created, as follows:

1. In the J2EE Hierarchy view, double-click the **OrderGiftBean** class to open it and view its source code.
2. In the Outline view, select the ejbCreate(Long) method. Its source code appears as follows:

```
public com.ibm.commerce.extension.objects.OrderGiftKey
   ejbCreate(java.lang.Long ordersId)
   throws javax.ejb.CreateException {
      _initLinks();
      this.ordersId = ordersId;
      return null;
}
```

3. Modify the code so that it appears as follows:

```
public com.ibm.commerce.extension.objects.OrderGiftKey
   ejbCreate(java.lang.Long ordersId)
   throws javax.ejb.CreateException {
      _initLinks();
      this.ordersId = ordersId;
      this.senderName = null;
      this.receiptName = null;
      this.msgField1 = null;
      this.msgField2 = null;
      return null;
}
```

4. Save the code changes.

Next, add a new ejbCreate method to the OrderGift bean, by doing the following:

1. In the J2EE Hierarchy view, double-click the **OrderGiftBean** class to open it and view its source code.
2. Create a new ejbCreate(Long, String, String, String, String) method, by adding the following code into the class:

```
public com.ibm.commerce.extension.objects.OrderGiftKey
   ejbCreate(java.lang.Long ordersId,
            java.lang.String receiptName,
            java.lang.String senderName,
            java.lang.String msgField1,
            java.lang.String msgField2)
   throws javax.ejb.CreateException {
      _initLinks();
      this.ordersId = ordersId;
      this.senderName = senderName;
      this.receiptName = receiptName;
      this.msgField1 = msgField1;
      this.msgField2 = msgField2;
      return null;
  }
```

3. Save the code changes.
4. You must add this new ejbCreate method to the home interface. This makes the method available in the generated access bean. To add the method to the home interface, do the following:
   a. In the Outline view, right-click the **ejbCreate(Long, String, String, String, String)** method and select **Enterprise Bean > Promote to Home Interface**.

Next create a new ejbPostCreate(Long, String, String, String, String) method so that it has the same parameters as the ejbCreate(Long, String, String, String, String) method, by doing the following:

1. Double-click the **OrderGiftBean** class to open it and view its source code.
2. Create a new ejbPostCreate(Long ordersId, String receiptName, String senderName, String msgField1, String msgField2) method, by adding the following code into the class:

```
public void ejbPostCreate(
   java.lang.Long ordersId,
   java.lang.String receiptName,
   java.lang.String senderName,
   java.lang.String msgField1,
   java.lang.String msgField2)
      throws javax.ejb.CreateException {
   }
```

3. Save the code changes.

Next you must create the definition for the XORDGIFT table.If you have *not* completed Chapter 10, “Tutorial: Creating new business logic,” on page 217, do the following to create the XORDGIFT table definition

1. Open the Data perspective and switch to the Data Definition view.
2. Navigate to the following directory: **WebSphereCommerceServerExtensionsData > ejbModule > META-INF**

3. Right-click **META-INF** and select **New database definition**.
   The New Database Definition wizard opens.
4. In the **Database name** field, enter the name of your development database. For example, enter `Demo_Dev`.
5. From the Database vendor type drop-down list, select the appropriate database type for your development environment:

   DB2

   - Business Professional **DB2 Universal Database V8.1**
   - Express **DB2 Universal Database V8.1 Express**

   Oracle **Oracle 9i**
6. Click **Finish**. The new database definition is now created.
7. Navigate to the following directory: **WebSphereCommerceServerExtensionsData > ejbModule > META-INF > Schema >** *developmentDB*.
8. Right-click *developmentDB* and select **New > New schema definition**.
   The New Schema Definition wizard opens.
9. In the **Schema name** field, enter `NULLID` click **Finish**.
10. Navigate to the **WebSphereCommerceServerExtensionsData > ejbModule > META-INF > Schema >** *developmentDB* **> NULLID >Tables**.
11. Right-click **Tables** and select **New table definition**.
    The New Table Definition wizard opens.
12. In the **Table name** field, enter `XORDGIFT` and click **Next**.
13. Add the key column to your table definition, as follows:

    Business Professional

    a. Click **Add Another**.
    b. In the **Column name** field, enter `ORDERSID`.
    c. From the **Column type** drop-down list, select the following:

       DB2 `BIGINT`

       Oracle `NUMBER`
    d. Select **Key column**.
    e. Oracle In the **Numeric precision** field, enter 38.
    f. Oracle Leave the value for **Numeric Scale** at `0`.

    Express

    a. Click **Add Another**.
    b. In the **Column name** field, enter `ORDERSID`.
    c. Select **Key column**.

d. From the **Column type** drop-down list, select the following:

DB2 BIGINT

Oracle NUMBER

e. Oracle In the **Numeric precision** field, enter 38.

f. Oracle Leave the value for **Numeric Scale** at 0.

14. Add additional columns to the table definition, as follows:

a. Click **Add Another** and create a column with the following properties:

Table 17.

Property	Value
Column name	RECEIPTNAME
Column type	DB2 VARCHAR Oracle VARCHAR2
Nullable	Select
String length	50
For bit data	Clear

b. Click **Add Another** and create a column with the following properties:

Table 18.

Property	Value
Column name	SENDERNAME
Column type	DB2 VARCHAR Oracle VARCHAR2
Nullable	Select
String length	50
For bit data	Clear

c. Click **Add Another** and create a column with the following properties:

Table 19.

Property	Value
Column name	MSGFIELD1
Column type	DB2 VARCHAR Oracle VARCHAR2

Table 19. (continued)

Property	Value
Nullable	Select
String length	50
For bit data	Clear

d. Click **Add Another** and create a column with the following properties:

Table 20.

Property	Value
Column name	MSGFIELD2
Column type	DB2 VARCHAR Oracle VARCHAR2
Nullable	Select
String length	50
For bit data	Clear

e. Click **Finish**.

f. Oracle You must edit the table definition using a text editor, as follows:

1) Switch to the Business Professional J2EE Navigator view Express Project Navigator view.

2) Expand the **WebSphereCommerceServerExtensionsData** project.

3) Expand the following: **ejbModule > META-INF >Schema**.

4) Right-click the **WebSphereCommerceServerExtensionsData_NULL_XORDGIFT. xmi** file and select **Open With > Text Editor**.

5) Replace all occurances of `SQLNumeric_6` to `SQLNumeric_3`.

6) Save your changes and close the text editor.

If you have completed Chapter 10, "Tutorial: Creating new business logic," on page 217, do the following to create the XORDGIFT table definition:

1. Open the Data perspective and switch to the Data Definition view.
2. Navigate to the following directory: **WebSphereCommerceServerExtensionsData > ejbModule > META-INF > Schema >** *developmentDB* **> NULLID > Tables**
3. Right-click the **Tables** directory and select **New > New Table Definition**. The New Table Definition wizard opens.

4. In the **Table name** field, enter XORDGIFT and click **Next**.
5. Add the key column to your table definition, as follows:

   Business Professional

   a. Click **Add Another**.
   b. In the **Column name** field, enter ORDERSID.
   c. From the **Column type** drop-down list, select the following:

      DB2 BIGINT

      Oracle NUMBER
   d. Select **Key column**.
   e. Oracle In the **Numeric precision** field, enter 38.
   f. Oracle Leave the value for **Numeric Scale** at 0.

   Express

   a. Click **Add Another**.
   b. In the **Column name** field, enter ORDERSID.
   c. Select **Key column**.
   d. From the **Column type** drop-down list, select the following:

      DB2 BIGINT

      Oracle NUMBER
   e. Oracle In the **Numeric precision** field, enter 38.
   f. Oracle Leave the value for **Numeric Scale** at 0.
6. Add additional columns to the table definition, as follows:
   a. Click **Add Another** and create a column with the following properties:

Table 21.

Property	**Value**
Column name	RECEIPTNAME
Column type	DB2 VARCHAR Oracle VARCHAR2
Nullable	Select
String length	50
For bit data	Clear

b. Click **Add Another** and create a column with the following properties:

Table 22.

Property	Value
Column name	SENDERNAME
Column type	DB2 VARCHAR Oracle VARCHAR2
Nullable	Select
String length	50
For bit data	Clear

c. Click **Add Another** and create a column with the following properties:

Table 23.

Property	Value
Column name	MSGFIELD1
Column type	DB2 VARCHAR Oracle VARCHAR2
Nullable	Select
String length	50
For bit data	Clear

d. Click **Add Another** and create a column with the following properties:

Table 24.

Property	Value
Column name	MSGFIELD2
Column type	DB2 VARCHAR Oracle VARCHAR2
Nullable	Select
String length	50
For bit data	Clear

e. Click **Finish**.

7. Oracle You must edit the table definition using a text editor, as follows:

a. Switch to the Business Professional J2EE Navigator view Express Project Navigator view.

b. Expand the **WebSphereCommerceServerExtensionsData** project.

c. Expand the following: **ejbModule > META-INF >Schema**.

d. Right-click the **WebSphereCommerceServerExtensionsData_NULL_XORDGIFT.xmi** file and select **Open With > Text Editor**.

e. Replace all occurances of `SQLNumeric_6` to `SQLNumeric_3`.

f. Save your changes and close the text editor.

The next step is to map the XORDGIFT table to the OrderGiftBean entity bean. Due to the fact that your development database and the XORDGIFT table already exist, Meet-in-the-middle mapping is used.

If you have not completed Chapter 10, "Tutorial: Creating new business logic," on page 217, do the following to create the mapping:

1. In the J2EE Hierarchy view, right-click the **WebSphereCommerceServerExtensionsData** project and select **Generate > EJB to RDB Mapping**.
   Create new EJB/RDB Mapping wizard opens.
2. Select **Meet In The Middle** and click **Next**.
3. Select **Match By Name, and Type** and click **Finish**.
   The Map.mapxmi editor opens.
4. In the Enterprise Beans pane, expand the **OrderGift** bean. In the Tables pane, expand the **XORDGIFT** table.
5. Map the fields in the Bonus bean to the columns in the XORDGIFT table, by doing the following:
   a. Right-click the **OrderGift** bean and select **Match By Name**.
6. Save the Map.mapxmi file by hitting Ctrl+S. Close the file.

If you have completed Chapter 10, "Tutorial: Creating new business logic," on page 217, do the following to create the mapping:

1. In the J2EE Hierarchy view, right-click the **WebSphereCommerceServerExtensionsData** project and select **Generate > EJB to RDB Mapping**.
   The Map.mapxmi editor opens.
2. In the Enterprise Beans pane, expand the **OrderGift** bean. In the Tables pane, expand the **XORDGIFT** table.
3. Map the fields in the Bonus bean to the columns in the XORDGIFT table, by doing the following:
   a. Right-click the **OrderGift** bean and select **Match By Name**.
4. Save the Map.mapxmi file by hitting Ctrl+S. Close the file.

Once the OrderGiftBean entity has been created and the schema is correctly mapped, you can create an access bean for the entity bean. This access bean

makes it simpler for applications to access information contained in the OrderGift entity bean. The tools in WebSphere Studio Application Developer are used to generate this access bean, based upon the entity that you have already created (in particular, only methods that have been promoted to the remote interface will be used by the access bean). To create the access bean for your OrderGift entity bean, do the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**, then right-click **WebSphereCommerceServerExtensionsData** and select **New > Access Bean**.
   The Add an Access Bean window opens.
2. Select **Copy Helper** and click **Next**.
3. Select the **OrderGift** bean and click **Next**.
4. From the **Constructor method** drop-down list, select **findByPrimaryKey(com.ibm.commerce.extension.objects.OrderGiftKey)**.
5. Select all attributes in the Attribute Helpers section.
6. Click **Finish**.

You can view the newly generated code by switching to the Business Professional J2EE Navigator view Express Project Navigator view, expand the **WebSphereCommerceServerExtensionsData** project, and expand **ejbModule** sub-folder, then expand **com.ibm.commerce.extension.objects**. A new class called OrderGiftAccessBean and a new interface called OrderGiftAccessBeanData are created and displayed inside the package.

The next step is to generate the deployed code.

The code generation utility analyzes the beans to ensure that Sun Microsystems' EJB specifications are met and it ensures that rules specific to the EJB server are followed. In addition, for each selected enterprise bean, the code-generation tool generates the home and EJBObject (remote) implementations and implementation classes for the home and remote interfaces, as well as the JDBC persister and finder classes for CMP beans. It also generates the Java ORB, stubs, and tie classes required for RMI access over IIOP, as well as stubs for the home and remote interfaces

To generate the deployed code, do the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**, then right-click **WebSphereCommerceServerExtensionsData** and select **Generate > Deploy and RMIC Code**.
   The Deploy and RMIC Code window opens.
2. Select **OrderGift** and click **Finish**.

You can view the newly generated code by switching to the Business Professional J2EE Navigator view Express Project Navigator view. . You will

find the following:

Table 25.

Type of code	Class name
Container implementation generated code	EJSCMPOrderGiftHomeBean.java
	EJSRemoteCMPOrderGift.java
	EJSRemoteCMPOrderGiftHome.java
	EJSFinderOrderGiftBean.java
JDBC access code	EJSJDBCPersisterCMPOrderGiftBean.java
RMI tie and stub code	_EJSRemoteCMPOrderGift_Tie.java
	_OrderGift_Stub.java
	_EJSRemoteCMPOrderGiftHome_Tie.java
	_OrderGiftHome_Stub.java

Integrating the OrderGift entity bean into the shopping flow

In this section, you integrate the OrderGift entity bean into the regular shopping flow of your sample store, by doing the following:

1. Creating a new OrderGiftDataBean data bean
2. Creating a new MyExtOrderProcessCmdImpl task command
3. Modifying the OrderSubmitForm.jsp display page

Each of the preceding steps are described in detail in subsequent sections.

Creating the OrderGiftDataBean

You must create the OrderGiftDataBean so that attributes from the OrderGift entity bean can be displayed on a JSP display page. As with other parts of the tutorial, the base code is provided and you need to uncomment various sections of code.

To create the OrderGiftDataBean, do the following:

1. The first step is to import the base code for the new data bean, as follows:
   a. Ensure that you have completed the steps in “Locating the sample code” on page 218.
   b. Switch to the J2EE Perspective and then select the Business Professional J2EE Navigator view Express Project Navigator view. .
   c. Expand the **WebSphereCommerceServerExtensionsLogic** project.
   d. Right-click the **src** folder and select **Import**.
   The Import wizard opens.
   e. From the **Select an import source** list, select **Zip file** and click **Next**.

f. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located, as follows:
yourDirectory\WC_SAMPLE_55.zip
where *yourDirectory* is the directory into which you downloaded the package.
g. Click **Deselect All**, then expand the directories and select the following file to import.
   - com\ibm\commerce\sample\databeans\OrderGiftDataBean.java
h. In the **Folder** field, the `WebSphereCommerceServerExtensionsLogic/src` folder is already specified. Keep this value.
i. Click **Finish**.

Once you have imported the code for the bean, examine the code.

Creating the MyExtOrderProcessCmdImpl class

In this section, you add new logic at the end of the OrderProcess business process so that information related to the gift order will be updated in the XORDGIFT database table. The ExtOrderProcessCmdImpl command is provided as the extension point to this business process. Following the programming model, to extend this logic, you create a new implementation class of the task command and include the new logic in this class. You must then update the command registry to associate the new implementation class with the ExtOrderProcessCmd interface.

To create the MyExtOrderProcessCmdImpl class, do the following:

1. The first step is to import the base code for the new command, as follows:
   a. Expand the **WebSphereCommerceServerExtensionsLogic** project.
   b. Right-click the **src** folder and select **Import**.
   The Import wizard opens.
   c. As the import source, select **Zip file** and click **Next**.
   d. From the **Select an import source** list, select **Zip file** and click **Next**.
   e. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located, as follows:
   yourDirectory\WC_SAMPLE_55.zip
   where *yourDirectory* is the directory into which you downloaded the package.
   f. Click **Deselect All**, then expand the directories and select the following file to import.
      - com\ibm\commerce\sample\commands\MyExtOrderProcessCmdImpl.java
   g. In the **Folder** field, the `WebSphereCommerceServerExtensionsLogic/src` folder is already specified. Keep this value.
   h. Click **Finish**.

Once you have imported the code for this new command, you can examine the source to see what the command does. Notice that the command calls the performExecute() method of its superclass to ensure that any processing from that command is executed. It then includes the logic to set the gift order information into the new data bean.

In this step, you modify the command registry so that the new MyExtOrderProcessCmdImpl implementation class gets used instead of the original ExtOrderProcessCmdImpl implementation class. The only table in the command registry that needs to be modified is the CMDREG table. In this case, the new implementation class is used for all stores.

To modify the command registry, do the following:

DB2 If you are using a DB2 database, do the following to register MyExtOrderProcessCmdImpl:

1. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Line Tools > Command Center**).
2. From the **Tools** menu, select **Tools Settings**.
3. Select the **Use statement termination character** check box and ensure the character specified is a semicolon (**;**)
4. Close the tools settings.
5. With the Script tab selected, create the required entry in the URLREG table, by entering the following information in the script window:

```
connect to developmentDB user dbuser using dbpassword;
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION,
   CLASSNAME, TARGET)
values (FashionFlow_storeent_ID,
   'com.ibm.commerce.order.commands.ExtOrderProcessCmd',
   'This is a new task command for tutorial two.',
   'com.ibm.commerce.sample.commands.MyExtOrderProcessCmdImpl',
   'local');
```

   where
   - *developmentDB* is the name of your development database
   - *dbuser* is the database user
   - *dbpassword* is the password of your database user
   - *FashionFlow_storeent_ID* is the store identifier for your sample store

   Click the **Execute** icon.

Oracle If you are using an Oracle database, do the following to register MyOrderItemAddCmdImpl:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

```
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION,
   CLASSNAME, TARGET)
values (FashionFlow_storeent_ID,
   'com.ibm.commerce.order.commands.ExtOrderProcessCmd',
   'This is a new task command for tutorial two.',
   'com.ibm.commerce.sample.commands.MyExtOrderProcessCmdImpl',
   'local');
```

   Press Enter to run the SQL statement.
6. Enter the following to commit your database changes:

```
commit;
```

   and press Enter to run the SQL statement.

Compiling changes

Important: Ensure that you do not rebuild the Stores web project.

In this section, you compile the changes you have made to your code, as follows:

1. In the Business Professional J2EE Navigator view Express Project Navigator view, select the following projects:
   - WebSphereCommerceServerExtensionsData
   - WebSphereCommerceServerExtensionsLogic
2. With the preceding projects highlighted, right-click and select **Build project**.

Modifying display pages for gift messages

In this step, you modify the OrderSubmitForm and OrderDetailDisplay templates so that the customer can input message information for the gift order as well as view the information in a summary page. The strategy for modifying these pages is to include additional JSP templates that specify the new information for the page. These new pages (OrderSubmitFormInclude.jsp and OrderDetailDisplayInclude.jsp) use the JSTL to display the new information.

To modify the display pages, do the following:

1. Switch to the Web perspective and use the Business Professional J2EE Navigator view Express Project Navigator view.
2. If you have not completed Chapter 10, "Tutorial: Creating new business logic," on page 217, you must modify the properties of the Stores web project, as follows:
   a. Right-click the **Stores** Web project and select **Properties**.
   b. Business Professional Select **Web** in the left pane and then from the list of Available Web Project Features, select **Include the JSP Standard Tag Library**.

      Express Select **Web Project Features** in the left pane and then from the list of Available Web Project Features, select **JSP Standard Tag Library**.

      Click **Apply**. When the update is complete, click **OK** to close the properties editor.
3. Expand to the following directory: **Stores\Web Content***FashionFlow_name*.
4. Create a backup copy of the OrderSubmitForm.jsp file, by doing the following:
   a. Expand the **ShoppingArea>CheckoutSection>StandardCheckoutSubsection** directories.
   b. Right-click the **OrderSubmitForm.jsp** file and select **Rename**.
   c. In the Rename window, enter `OrderSubmitForm_bak.jsp` and click **OK**.
   d. When prompted if you would like to update links to this file, click **No**.
5. Create a backup copy of the OrderDetailDisplay.jsp file, by doing the following:
   a. Expand the **UserArea>ServiceSection>TrackOrderStatusSubsection** directories.
   b. Right-click the **OrderDetailDisplay.jsp** file and select **Rename**.
   c. In the Rename window, enter `OrderDetailDisplay_bak.jsp` and click **OK**.
   d. When prompted if you would like to update links to this file, click **No**.
6. Right-click the *FashionFlow_name* directory and select **Import**. The Import wizard opens.
7. From the **Select an import source** list, select **Zip file** and click **Next**.
8. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located, as follows:

yourDirectory\WC_SAMPLE_55.zip
where *yourDirectory* is the directory into which you downloaded the package.

9. Click **Deselect All**, then expand the directories and select the following files to import.
   - ShoppingArea\CheckoutSection\StandardCheckoutSubsection\OrderSubmitForm.jsp
   - ShoppingArea\CheckoutSection\StandardCheckoutSubsection\OrderSubmitFormInclude.jsp
   - UserArea\ServiceSection\TrackOrderStatusSubsection\OrderDetailDisplay.jsp
   - UserArea\ServiceSection\TrackOrderStatusSubsection\OrderDetailDisplayInclude.jsp
10. In the **Folder** field, the `Stores/Web Content/`*`FashionFlow_name`* folder is already specified. Keep this value.
11. Click **Finish**.

If you examine the OrderSubmitForm.jsp file, you will find that includes the following:

```
<!-- tutorial start-->
   <jsp:include page="OrderSubmitFormInclude.jsp" flush="true" />
<!-- tutorial done -->
```

This is the way that the new OrderSubmitFormInclude.jsp file is incorporated into the existing JSP template. Examine the OrderSubmitFormInclude.jsp for an example of how to use the JSTL in your templates. Similarly, examine the OrderDetailDisplay.jsp and OrderDetailDisplayInclude.jsp files.

You must also import the properties file that contains the string values used in the modified JSP templates. This file is called ordergift.properties. To import this file, do the following:

1. In the Business Professional J2EE Navigator view Express Project Navigator view, expand the following directories:
   Stores > Web Content> WEB-INF >classes > *FashionFlow_name* directory.
2. Right-click the *FashionFlow_name* directory and select **Import**.
   The Import wizard opens.
3. From the **Select an import source** list, select **Zip file** and click **Next**.
4. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located, as follows:
   yourDirectory\WC_SAMPLE_55.zip
   where *yourDirectory* is the directory into which you downloaded the package.

5. Click **Deselect All**, then expand the directories and select the following file to import.
   - ordergift.properties
6. In the **Folder** field, the `Stores/Web Content/WEB-INF/classes/`*`FashionFlow_name`* folder is already specified. Keep this value.
7. Click **Finish**.

Testing the new gift message functionality

In this section, you test the new gift message functionality, as follows:

1. Switch to the Server perspective (**Window > Open Perspective > Server**).
2. Start your payments server.
3. Right-click the **WebSphereCommerceServer** server and select **Start** (or **Restart**).
4. Open a Web browser and enter the following URL:

   `http://localhost/webapp/wcs/stores/servlet/FashionFlow/index.jsp`
5. Logon as a new user. For example, click **Register** and then create a new user, or logon as an existing user.
6. As the new registered user, browse through the store, add an item to the shopping cart, and then complete the purchase. You will be able to add a gift message to your order, as shown in the following screen shot:

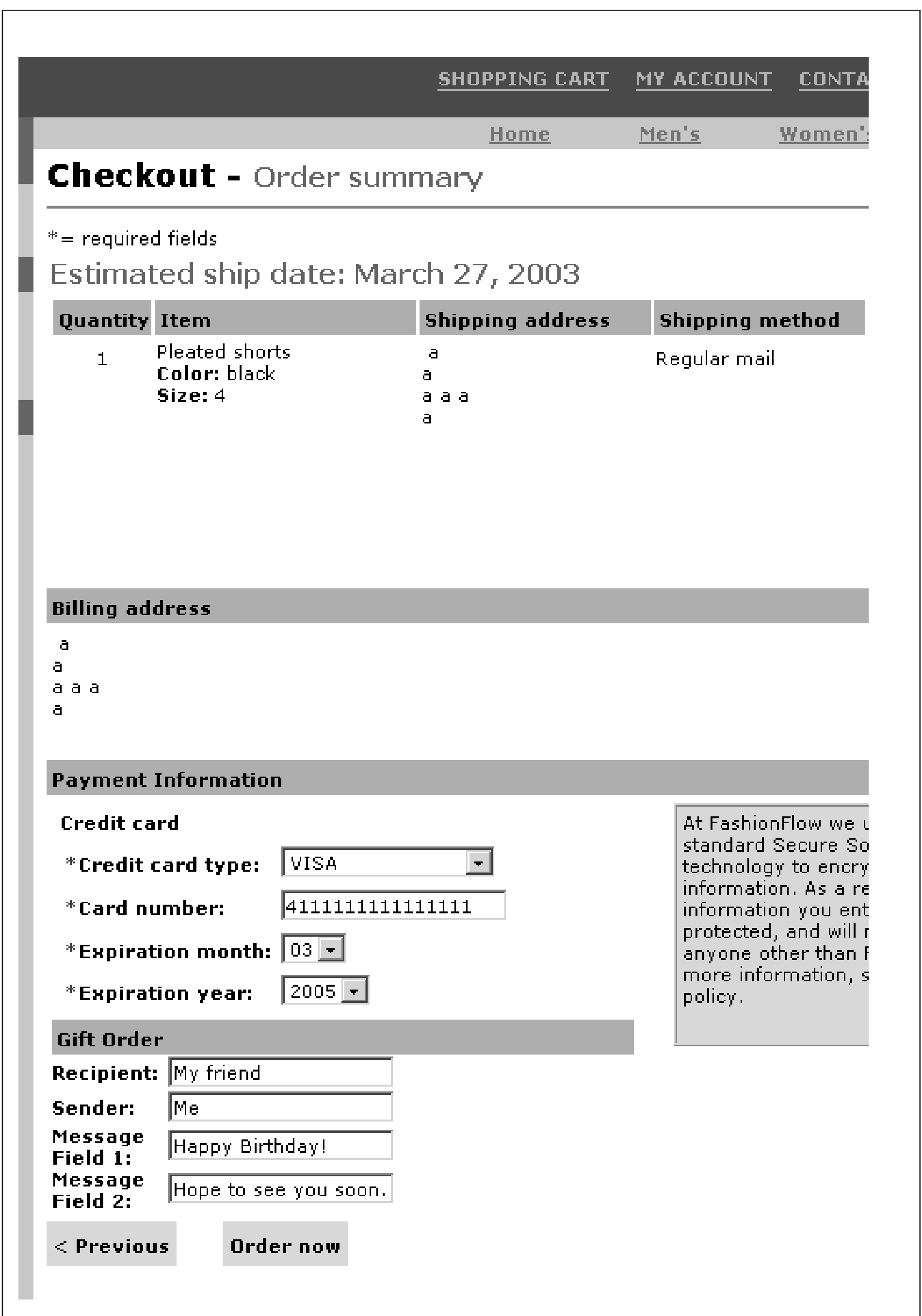


Figure 49.

Note the order number associated with this order.

7. Click **My Account**.
8. Click **View orders**.
9. Select the order that you created in step 6. You will see a screen similar to the following:

Your order details:

Estimated ship date: June 2

Quantity	Tracking Number	Item
1		Team s

Gift Order

Receiver=My Friend

Sender=Me

Message field 1=Happy Birthday!

Message field 2=Hope to see you soon.

Home | Shopping cart | My acc

Figure 50.

Deploying the gift message functionality

This section describes how to deploy your new business logic into a store running on a remote WebSphere Commerce Server. You must have created a store (based upon the FashionFlow sample store) on the remote WebSphere Commerce Server before starting these deployment steps. Within that store, you must be able to complete a purchase.

The deployment process includes steps that are performed on the development machine, as well as steps that are performed on the target WebSphere Commerce Server.

There are a number of different types of assets that must be deployed to the target WebSphere Commerce Server. These include:

- Task command and data bean logic
- Enterprise bean logic
- Updated JSP templates
- Database updates including schema updates (new table) as well as command registry updates

This section describes how to deploy all of these assets, *incrementally* to the target WebSphere Commerce Server. This is done as an incremental deployment, in contrast to a deployment of the entire EAR file.

Creating the command and data bean JAR file

This section describes how to create the JAR file that contains the controller command, task command, and data bean logic.

To create this JAR file, perform the following steps on your development machine:

1. Create a directory on your local file system called \ExportTemp3.
2. Open WebSphere Studio Application Developer and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
3. Right-click the **WebSphereCommerceServerExtensionsLogic** project and select **Export**.
   The Export wizard opens.
4. In the Export wizard, do the following:
   a. Select **JAR file** and click **Next**.
   b. The left pane under **Select the resources to export** is prepopulated with the name of the project. Leave this value as is.
   c. In the right pane, ensure that only the following resources are selected:
      - .classpath
      - .project

- .serverPreference

d. Ensure that **Export generated class files and resources** is selected.
e. Do *not* select **Export Java source files and resources**.
f. In the **Select the export destination** field, enter the fully-qualified JAR file name to use. In this case, enter `drive:\ExportTemp3\WebSphereCommerceServerExtensionsLogic.jar`. Note that the JAR file name must be WebSphereCommerceServerExtensionsLogic.jar.
g. Click **Finish**.

Creating the EJB JAR file

To create the EJB JAR file, do the following:

1. Open WebSphere Studio Application Developer and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the **WebSphereCommerceServerExtensionsData** project.
3. Double-click **EJB Deployment Descriptor**.
4. With the Overview tab selected, scroll to the bottom of the pane, to locate the **WebSphere Bindings** section.
5. In the **DataSource JNDI name** field, enter the datasource JNDI name of the target WebSphere Commerce Server. The following is an example value:

   DB2 `jdbc/WebSphere Commerce DB2 DataSource demo`
   where the target WebSphere Commerce Server is using a DB2 database, and the WebSphere Commerce instance name is "demo"

   Oracle `jdbc/WebSphere Commerce Oracle DataSource demo`
   where the target WebSphere Commerce Server is using an Oracle database, and the WebSphere Commerce instance name is "demo".

The value for the DataSource JNDI name is created by adding "`jdbc/`" to the data source name of the target WebSphere Commerce Server. You can verify the data source name by opening the *instanceName*.xml file on the target WebSphere Commerce Server and searching for `DatasourceName=` in the file.

6. Save your deployment descriptor changes (Ctrl+S).
7. In the Business Professional J2EE Navigator view Express Project Navigator view, right-click the **WebSphereCommerceServerExtensionsData** project and select **Export**.
   The Export wizard opens.
8. In the Export wizard, do the following:
   a. Select **EJB JAR file** and click **Next**.

b. Business Professional The value for **What resources do you want to export?** is prepopulated with the name of the EJB project.

   Express The EJB project name is prepopulate.
   Leave this value as is.

c. Business Professional In the **Where do you want to export resources to?** field, enter the fully-qualified JAR file name to use.

   Express For the destination, enter the fully-qualified JAR file name to use.
   In this case, enter
   `*drive*:\ExportTemp3\WebSphereCommerceServerExtensionsData.jar`.

d. Click **Finish**.

9. After the JAR file has been created, undo the changes to the local deployment descriptor that was made in step 5, to restore the setting that is required for your local test server.

Exporting store assets

To export the OrderSubmitForm and OrderDetailDisplay display templates from WebSphere Studio Application Developer, do the following:

1. Open WebSphere Studio Application Developer and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the **Stores** folder.
3. Right-click the **Web Content** folder and select **Export**.
   The Export Wizard opens.
4. In the Export wizard, do the following:
   a. Select **File system** and click **Next**.
   b. Select the following resources to deploy:
      - Web Content*FashionFlow_name*\ShoppingArea\CheckoutSection\StandardCheckoutSubsection\OrderSubmitForm.jsp
      - Web Content*FashionFlow_name*\ShoppingArea\CheckoutSection\StandardCheckoutSubsection\OrderSubmitFormInclude.jsp
      - Web Content*FashionFlow_name*\UserArea\ServiceSection\TrackOrderStatusSubsection\OrderDetailDisplay.jsp
      - Web Content*FashionFlow_name*\UserArea\ServiceSection\TrackOrderStatusSubsection\OrderDetailDisplayInclude.jsp
      - Web Content\WEB-INF\lib\jstl.jar
      - Web Content\WEB-INF\lib\standard.jar

- Web Content\WEB-INF\classes*FashionFlow_name*\ordergift.properties

c. Select **Create directory structure for files**.

d. In the Directory field, enter a temporary directory into which these resources will be placed. For example, enter `C:\ExportTemp3`

e. Click **Finish**.

Transferring assets to your target WebSphere Commerce Server

In this step, you create a temporary directory on the target WebSphere Commerce Server and then copy your gift order assets into this directory. In subsequent steps, you will place the different types of code into the appropriate place within your WebSphere Commerce application.

To copy the files from your development machine to your target WebSphere Commerce Server, do the following:

1. On the target WebSphere Commerce Server, create a temporary directory called *drive*:\ImportTemp3.
2. Determine how you will copy your files from one computer to another. You can do this by mapping a drive on the target WebSphere Commerce Server to the development machine, or by using an FTP application, if you have that configured.
3. From the development machine, copy the contents of \ExportTemp3 into \ImportTemp3 on the target WebSphere Commerce Server.

Stopping your target WebSphere Commerce Server

Before starting the deployment steps, you should stop your target WebSphere Commerce Server by issuing the stopServer command at the command line. For details about this command, refer to the Business Professional *WebSphere Commerce Studio Installation Guide* or Express *WebSphere Commerce - Express Developer Edition Installation Guide*.

Updating the database on your target WebSphere Commerce Server

Registering the task command implementation

To modify the command registry, do the following:

1. DB2 If you are using a DB2 database, do the following to register MyExtOrderProcessCmdImpl:
   a. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Line Tools > Command Center**).
   b. From the **Tools** menu, select **Tools Settings**.
   c. Select the **Use statement termination character** check box and ensure the character specified is a semicolon (**;**)
   d. Close the tools settings.

e. With the Script tab selected, create the required entry in the URLREG table, by entering the following information in the script window:

```
connect to targetDB user dbuser using dbpassword;
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION,
   CLASSNAME, TARGET)
values (FashionFlow_storeent_ID,
   'com.ibm.commerce.order.commands.ExtOrderProcessCmd',
   'This is a new task command for tutorial two.',
   'com.ibm.commerce.sample.commands.MyExtOrderProcessCmdImpl',
   'local');
```

where

- *targetDB* is the name of your target database
- *dbuser* is database user
- *dbpassword* is the password of the database user
- *FashionFlow_storeent_ID* is the store identifier for your sample store

Click the **Execute** icon. Keep the Command Center open.

2. Oracle If you are using an Oracle database, do the following to register MyOrderItemAddCmdImpl:
   a. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
   b. In the **User Name** field, enter your Oracle user name.
   c. In the **Password** field, enter your Oracle password.
   d. In the **Host String** field, enter your connect string.
   e. In the SQL Plus window, enter the following SQL statement:

```
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION,
   CLASSNAME, TARGET)
values (FashionFlow_storeent_ID,
   'com.ibm.commerce.order.commands.ExtOrderProcessCmd',
   'This is a new task command for tutorial two.',
   'com.ibm.commerce.sample.commands.MyExtOrderProcessCmdImpl',
   'local');
```

   Press Enter to run the SQL statement.
   f. Enter the following to commit your database changes:

```
commit;
```

   and press Enter to run the SQL statement.

Creating the XORDGIFT table

In this step, you create the XORDGIFT table.

DB2 If you are using a DB2 database, do the following to create the table:

1. In the Script window, enter the following:

```
create table XORDGIFT (ORDERSID bigint not null, RECEIPTNAME varchar(50),
SENDERNAME varchar(50), MSGFIELD1 varchar(50), MSGFIELD2 varchar(50),
constraint p_xordgift primary key (ORDERSID),
constraint f_xordgift foreign key (ORDERSID) references ORDERS(ORDERS_ID)
on delete cascade);
insert into XORDGIFT (ORDERSID) (select ORDERS_ID from ORDERS);
```

where

- *targetDB* is the name of your target database
- *dbuser* is database user
- *dbpassword* is the password of the database user

Click the Execute icon.
The XORDGIFT table is now created.

Oracle If you are using an Oracle database, do the following to create the table:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statements:

   ```
   create table XORDGIFT (ORDERSID NUMBER NOT NULL, RECEIPTNAME VARCHAR(50),
   SENDERNAME VARCHAR(50), MSGFIELD1 VARCHAR(50), MSGFIELD2 VARCHAR(50),
   constraint p_xordgift primary key (ORDERSID),
   constraint f_xordgift foreign key (ORDERSID) references ORDERS(ORDERS_ID)
   on delete cascade);
   insert into XORDGIFT (ORDERSID) (select ORDERS_ID from ORDERS);
   ```

   and press Enter to run the SQL statement. The XORDGIFT table is now created.
6. Enter the following to commit your database changes:

   ```
   commit;
   ```

   and press Enter to run the SQL statement.

Updating store assets on your target WebSphere Commerce Server

In this step, you update the store with your modified store assets, as follows:

1. Backup your *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear\ Stores.war directory.
2. Navigate to the \ImportTemp3\Stores\Web Content directory.
3. Copy the *FashionFlow_name* folder into the following directory: *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear\ Stores.war

where *instanceName* is the name of your WebSphere Commerce instance.

Updating the command and data bean JAR file on your target WebSphere Commerce Server

In this step you update the target WebSphere Commerce Server to use the new command and data bean JAR file, as follows:

1. You should make a backup copy of the existing JAR file, as follows:
   a. Navigate to the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory.
   b. Make a copy of the WebSphereCommerceServerExtensionsLogic.jar file and save it in a backup location.
2. Copy the new WebSphereCommerceServerExtensionsLogic.jar file from the \ImportTemp3 directory into the *WAS_installdir*\nstalledApps*cellName*\WC_*instanceName*.ear directory

where *instanceName* is the name of your WebSphere Commerce instance.

Updating the EJB JAR file on your target WebSphere Commerce Server

In this step you update the target WebSphere Commerce Server to use the new EJB JAR file, as follows:

1. You should make a backup copy of the existing JAR file, as follows:
   a. Navigate to the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory.
   b. Make a copy of the WebSphereCommerceServerExtensionsData.jar file and save it in a backup location.

   where *instanceName* is the name of your WebSphere Commerce instance.
2. Copy the new WebSphereCommerceServerExtensionsData.jar file from the \ImportTemp3 directory into the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory
3. Next, you must modify the EJB deployment descriptor information, as follows:
   a. Locate the deployment repository (META-INF directory) for this WebSphere Application Server cell. This typically takes the following form:
      WAS_installdir\config\cells*cellName*
      \applications\WC_*instance_name*.ear\deployments\
      WC_*instance_name**EJBModuleName*.jar\META-INF
      The following is a specific example of this:
      D:\WebSphere\AppServer\config\cells\myCell\applications\
      WC_demo.ear\deployments\WC_demo\
      WebSphereCommerceServerExtensionsData.jar\META-INF
      where
      - mycell is the name of the WebSphere Application Server cell

- demo is the name of the WebSphere Commerce instance

b. The directory contains the following files:
   - ejb-jar.xml
   - ibm-ejb-access-bean.xmi
   - ibm-ejb-jar-bnd.xmi
   - ibm-ejb-jar-ext.xmi
   - MANIFEST.MF

   Backup all of these files.

c. Use a tool to open the new WebSphereCommerceServerExtensionsData.jar file and view its contents.

d. Extract the contents of the meta-inf directory from this WebSphereCommerceServerExtensionsData.jar file into the directory from step 3a.

4. Using the WebSphere Application Server startServer command at the command line, restart your WebSphere Commerce instance.

Verifying the gift message functionality on the target WebSphere Commerce Server

In this section, you verify that the gift message logic is functioning correctly on the target WebSphere Commerce Server by doing the following:

1. Open a browser and enter the URL for your store that is based on the FashionFlow sample store.
2. Logon as a new user. For example, click ″Register″ and then create the user ″shopper″.
3. As the new registered user, browse through the store, add an item to the shopping cart, and then complete the purchase. You will be able to add a gift message to your order, as shown in the following screen shot:

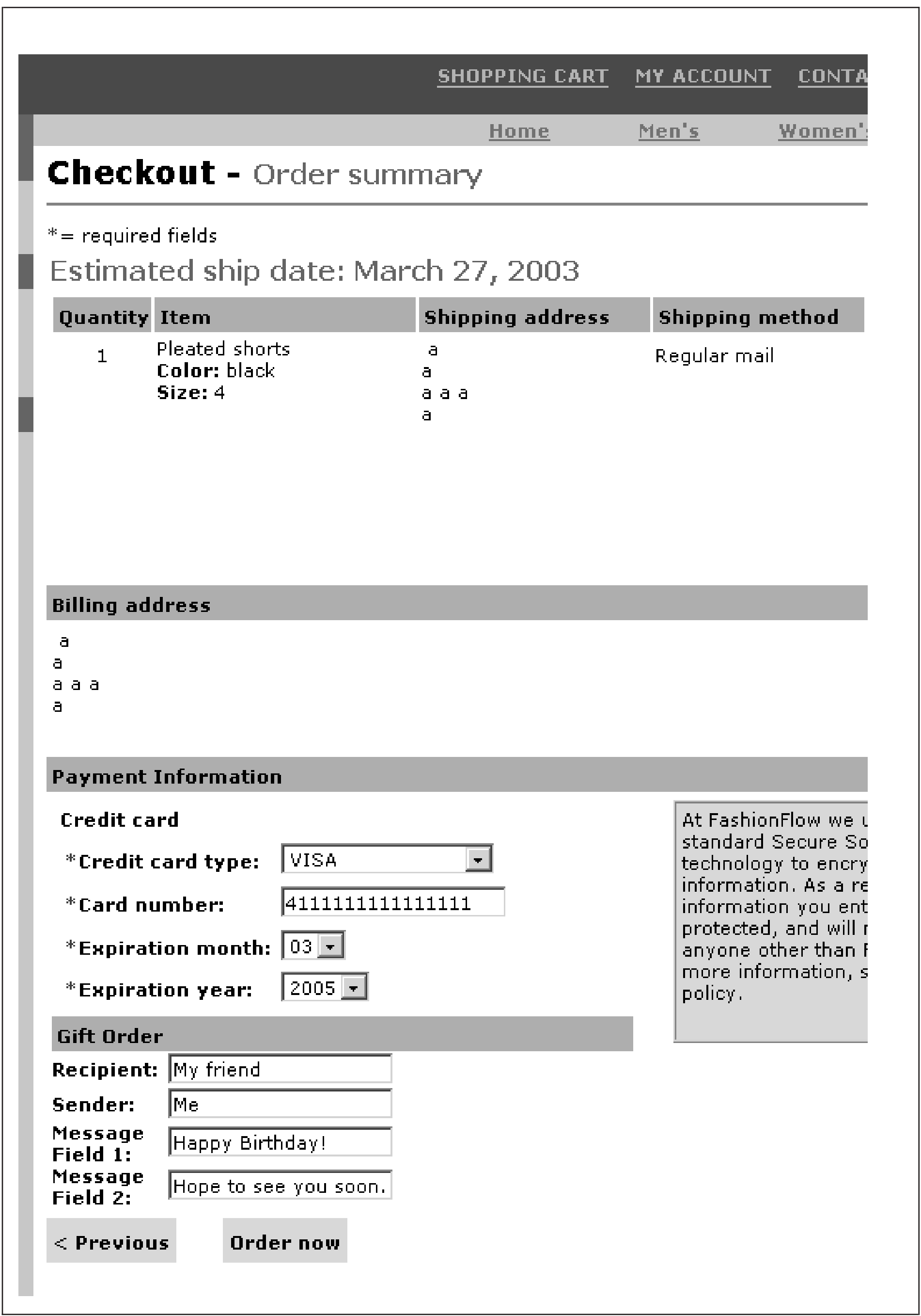


Figure 51.

Note the order number associated with this order.

4. Click **My Account**.
5. Click **View orders**.
6. Select the order that you created in step 3. You will see a screen similar to the following:

Your order details:

Estimated ship date: June 2

Quantity	Tracking Number	Item
1		Team s

Gift Order

Receiver=My Friend

Sender=Me

Message field 1=Happy Birthday!

Message field 2=Hope to see you soon.

Home | Shopping cart | My acc

Figure 52.

Chapter 13. Tutorial: Extending an existing WebSphere Commerce entity bean

In this tutorial, you learn the process used to modify an existing WebSphere Commerce entity bean. It uses the scenario of adding an additional housing information survey to the user registration process. In this case, the User entity bean is modified to include additional fields that store a user's housing type value and a location value. A new database table is created (called XHOUSING) and the existing mapping information for the User bean is modified to include this new table.

In addition to the new table and entity bean changes, a new MyPostUserRegistrationAddCmdImpl implementation class is created. This contains the logic that processes the housing survey information and updates the database with the new information.

In order to be able to collect the housing information and then later, display the results, the UserRegistrationAddForm.jsp and UserRegistrationUpdateForm.jsp files are modified.

Prerequisites

This tutorial does not require that you have completed tutorials. If you have completed those tutorials, there is no harm in leaving the code in your workspace, as it will not conflict with this tutorial.

Before starting this tutorial, you must have published a store based upon the FashionFlow sample store. Within this store, you must be able to complete a purchase (for example, browse the catalog, add items to the shopping cart, checkout and see the order confirmation).

Creating and populating the XHOUSING table

In preparation for creating the entity bean, you must first create and populate a new database table. The table to be created is called XHOUSING.

DB2 If you are using a DB2 database, do the following to create the table:

1. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Line Tools > Command Center**) and click the **Scripting** tab.
2. In the Script window, enter the following:

© Copyright IBM Corp. 2000, 2003

```
connect to developmentDB user dbuser using dbpassword;
create table XHOUSING (MEMBERID bigint not null,
   HOUSINGTYPE integer, LOCATION integer,
   constraint p_xhousing primary key (MEMBERID),
   constraint f_xhousing foreign key (MEMBERID)
   references USERS (USERS_ID) on delete cascade);
insert into XHOUSING (MEMBERID) (select USERS_ID from USERS);
```

where

- *developmentDB* is the name of your development database
- *dbuser* is the database user
- *dbpassword* is the password for your database user

Click the Execute icon.
You should see a message indicating that the SQL statement completed successfully.

Oracle If you are using an Oracle database, do the following to create the table:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

   ```
   create table XHOUSING (MEMBERID number not null,
      HOUSINGTYPE integer, LOCATION integer,
      constraint p_xhousing primary key (MEMBERID),
      constraint f_xhousing foreign key (MEMBERID)
      references USERS (USERS_ID) on delete cascade);
   insert into XHOUSING (MEMBERID) (select USERS_ID from USERS);
   ```

   and press Enter to run the SQL statement. The XHOUSING table is now created.
6. Enter the following to commit your database changes:

   ```
   commit;
   ```

   and press Enter to run the SQL statement.

Adding new fields to the User entity bean

In this section, you add two new fields to the User entity bean that are used to capture housing information. The new fields are called housingType and location. These fields are eventually mapped to the HOUSINGTYPE and LOCATION columns of the XHOUSING table.

To add these new fields, do the following:

1. Start WebSphere Commerce development environment as follows:
   - Business Professional **Start > Programs > IBM WebSphere Commerce Studio > WebSphere Commerce development environment**
   - Express Express **Start > Programs > IBM WebSphere - Express Developer Edition > WebSphere Commerce development environment**
2. Switch to the Server perspective and ensure that the WebSphereCommerceServer test server is stopped.
3. Switch to the J2EE perspective and select the J2EE Hierarchy view.
4. Expand **EJB Modules** and then double-click the **Member-MemberManagementData** EJB project. This opens the EJB Deployment Descriptor editor.
5. Click the **Beans** tab and select the **User** bean from the list of beans displayed.
6. Click **Add** beside the CMP fields text box.
   The Create CMP attribute window opens.
7. Create a CMP field with the following properties:
   a. In the **Name** field, enter `housingType`.
   b. In the Type field, enter `java.lang.Integer`.
   c. Enable **Access with getter and setter methods**.
   d. Clear the **Promote getter and setter to remote interface** check box. (This automatically clears the option to make the getter a read-only option.)
   e. Click **OK**.
8. Click **Add** again to create another CMP field with the following properties:
   a. In the **Name** field, enter `location`.
   b. In the Type field, enter `java.lang.Integer`.
   c. Enable **Access with getter and setter methods**.
   d. Clear the **Promote getter and setter to remote interface** check box. (This automatically clears the option to make the getter a read-only option.)
   e. Click **OK**.

   The new fields are displayed in the list of CMP fields.
9. Save your changes and then close the EJB Deployment Descriptor editor.

Updating the schema and table mapping information

In the following sections, you update the User schema with the new XHOUSING table, create the foreign key relationship for the new table, and create a table map between the fields of the User entity bean and the columns of the XHOUSING table. By taking this approach to extending the object model, it appears to code as if new columns have been added directly to the USERS table.

Creating the table definition for the XHOUSING table

To create the XHOUSING table definition and create the foreign key relationship between the USERS and XHOUSING tables, do the following:

1. In the J2EE Hierarchy view, expand **Databases**.
2. Expand the **Member-MemberManagementData** database, then expand the **NULLID** schema.
3. Right-click **Tables** and select **New > New table definition**. The Table Definition window opens.
4. In the **Table name** field, enter `XHOUSING` and click **Next**.
5. Add the key column to your table definition, as follows:

   a. Click **Add Another**.
   b. In the **Column name** field, enter `ORDERSID`.
   c. From the **Column type** drop-down list, select the following:
      DB2 `BIGINT`
      Oracle `NUMBER`
   d. Select **Key column**.
   e. Oracle In the **Numeric precision** field, enter 38.
   f. Oracle Leave the value for **Numeric Scale** at `0`.

   Express

   a. Click **Add Another**.
   b. In the **Column name** field, enter `ORDERSID`.
   c. Select **Key column**.
   d. From the **Column type** drop-down list, select the following:
      DB2 `BIGINT`
      Oracle `NUMBER`
   e. Oracle In the **Numeric precision** field, enter 38.
   f. Oracle Leave the value for **Numeric Scale** at `0`.
6. Add additional columns to the table definition, as follows:

a. Click **Add Another** and create a column with the following properties:

Table 26.

Property	**Value**
Column name	HOUSINGTYPE
Column type	INTEGER
Nullable	Select

b. Click **Add Another** and create a column with the following properties:

Table 27.

Property	**Value**
Column name	LOCATION
Column type	INTEGER
Nullable	Select

c. Click **Next**.

7. In the **Primary key name** field enter `p_xhousing` and click **Next**.
8. Click **Add another** to add the foreign key. Specify the following values:
   a. In the **Foreign key name** field, enter `f_xhousing`.
   b. From the **On Delete** drop-down list, select **CASCADE**
   c. From the **Target Table** drop-down list, select **NULLID.USERS**
   d. In the Source Columns pane, click **MEMBERID** and then click **>** to add the foreign key.
9. Click **Finish**.
10. Oracle You must edit the table definition using a text editor, as follows:
    a. Switch to the Business Professional J2EE Navigator view Express Project Navigator view.
    b. Expand the **Member-MemberManagementData** project.
    c. Expand the following: **ejbModule > META-INF >Schema**.
    d. Right-click the **Member-MemberManagementData_NULL_XHOUSING.xmi** file and select **Open With > Text Editor**.
    e. Replace all occurrences of `SQLNumeric_6` to `SQLNumeric_3`.
    f. Save your changes and close the text editor.

The new Member-MemberManagementData_NULL_XHOUSING table definition can be seen by expanding the Tables folder under Member-MemberManagementData/NULLID.

Creating the XHOUSING table map

In this section, you create the mapping between the two columns (HOUSINGTYPE and LOCATION) in the XHOUSING table and the two fields (housingType and location) in the User entity bean.

To create this mapping, do the following:

1. Switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the following folders: **Member-MemberManagementData > ejbModule >META-INF**.
3. Double-click the **Map.mapxmi** file.
4. Express In the Enterprise Beans pane, expand the **Member** group, then right-click the **User** entity bean.
5. In the Tables pane, highlight the following tables:
   - **MEMBER**
   - **USERS**
   - **XHOUSING**

   (Hold the Ctrl key to select multiple tables at once.)
6. Business Professional In the Enterprise Beans pane (at the top of editor), expand the **Member** group, then right-click the **User** entity bean and select **Create Mapping**.

   Express In the Enterprise Beans pane (at the top of editor), right-click the **User** entity bean and select **Create Mapping**.
7. Expand the **User** entity bean.
8. In the Tables pane, expand the **XHOUSING** table, so that its columns can be viewed.
9. Highlight the **housingType** bean attribute and drag it onto the **HOUSINGTYPE** column to create the mapping.
10. Highlight the **location** bean attribute and drag it onto the **LOCATION** column to create the mapping.
11. Save your changes, then close the Map.mapxmi file.

Updating the mapping file

1. Expand the following folders: **Member-MemberManagementData > ejbModule >META-INF**.
2. Right-click the **Map.mapxmi** file and select **Open With > Text Editor**.
3. Locate the following string:

   ```
   User_EJB
   ```

   Two lines below this, you may find:

```
<discriminatorValues>User</discriminatorValues>
```

If you find the preceding line, then you must change it to

```
<discriminatorValues>'U'</discriminatorValues>
```

4. Save your change.

Generating the access beans and deployed code

Since you have modified the User entity bean, you must regenerate its access bean and its deployed code.

To regenerate the access bean, do the following:

1. In the J2EE Hierarchy view, expand **EJB Modules**.
2. Right-click the **Member-MemberManagementData** EJB module and select **Access Beans> Edit Access Beans**.
3. Select **CopyHelper** and click **Next**.
4. In the Select EJB Project window, click **Next**.
5. In the Copy Helper Access Bean window, do the following:
   a. From the Enterprise beans drop-down list, select **User**.
   b. From the **Constructor method** drop-down list, select **findByPrimaryKey(com.ibm.commerce.user.objects.MemberKey)**.
   c. From the Attribute Helpers list, ensure that the **housingType** and **location** attributes are selected along with all of the other attributes.
6. Click **Finish**.
7. Right-click the **Member-MemberManagementData** EJB module and select **Access Beans> Regenerate Access Beans**.
8. Click **Select All** and then click **Finish**.

To regenerate the deployed code, do the following:

1. Right-click the **Member-MemberManagementData** EJB module and select **Generate > Deploy and RMIC Code**.
2. Click **Select All** and then click **Finish**.

Creating the MyPostUserRegistrationAddCmdImpl implementation

In this step, you extend the UserRegistrationAddCmd controller command to include new logic that is used to parse the new housing information that gets collected on the registration form. The extension is made by creating a new MyPostUserRegistrationAddCmdImpl implementation class that implements the PostUserRegistrationCmd interfaces. After creating this new implementation class, you must update the command registry to reflect this change. As with other tutorials, the code for this new command is provided.

To create the new MyPostUserRegistrationAddCmdImpl implementation class, do the following:

1. Ensure that you have completed the steps in “Locating the sample code” on page 218.
2. In WebSphere Studio Application Developer, open the Java perspective (**Window > Open Perspective > Java**).
3. Expand the **WebSphereCommerceServerExtensionsLogic** project.
4. Right-click the **src** folder and select **Import**.
   The Import wizard opens.
5. From the **Select an import source** list, select **Zip file** and click **Next**.
6. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located, as follows:
   yourDirectory\WC_SAMPLE_55.zip
   where *yourDirectory* is the directory into which you downloaded the package.
7. Click **Deselect All**, then expand the directories and select to import the following file:
   - com\ibm\commerce\sample\commands\MyPostUserRegistrationAddCmdImpl.java
8. In the **Folder** field, the `WebSphereCommerceServerExtensionsLogic/src` folder is already specified. Keep this value.
9. Click **Finish**.
10. Expand the **com.ibm.commerce.sample.commands** package.
11. Double-click the new **MyPostUserRegistrationAddCmdImpl** class.
12. Uncomment Section 1. This code sets up variables and creates getters and setters for these variables:

```
/// Section 1 ////////////////
 Integer housingType = null;
 Integer location = null;

 public Integer getHousingType() {
  return housingType;
 }

 public Integer getLocation() {
  return location;
 }

 public void setHousingType(Integer newHousingType) {
  housingType = newHousingType;
 }

 public void setLocation(Integer newLocation) {
  location = newLocation;
 }
/// End of Section 1 ////////////////
```

13. Uncomment Section 2. This introduces the following code into the class:

```
/// Section 2 ////////////////
 public void performExecute() throws ECException {
  super.performExecute();

  // Set the needed fields before processing the survey
  setHousingType(requestProperties.getInteger("housingType", 0));
  setLocation(requestProperties.getInteger("location", 0));

  processSurvey();
 }

/// End of Section 2 ////////////////
```

The preceding code calls the performExecute method of the superclass so that the regular logic of the PostUserRegistrationAddCmdImpl is executed. Once that has completed, the new processSurvey method is called.

14. Uncomment Section 3 to introduce the following code into the class:

```
/// Section 3 ////////////////

 private void processSurvey() throws ECException {

  try {
   // load up the user data
   UserAccessBean abUser = new UserAccessBean();
   abUser.setInitKey_MemberId(commandContext.getUserId().toString());
   abUser.refreshCopyHelper();

   // store the new attributes
   abUser.setHousingType(getHousingType());
   abUser.setLocation(getLocation());

   abUser.commitCopyHelper();
  } catch (javax.ejb.FinderException e) {
   throw new ECSystemException(
    ECMessage._ERR_FINDER_EXCEPTION,
    this.getClass().getName(),
    "processSurvey");
  } catch (javax.naming.NamingException e) {
   throw new ECSystemException(
    ECMessage._ERR_NAMING_EXCEPTION,
    this.getClass().getName(),
    "processSurvey");
  } catch (java.rmi.RemoteException e) {
   throw new ECSystemException(
    ECMessage._ERR_REMOTE_EXCEPTION,
    this.getClass().getName(),
    "processSurvey");
  } catch (javax.ejb.CreateException e) {
   throw new ECSystemException(
    ECMessage._ERR_CREATE_EXCEPTION,
    this.getClass().getName(),
```

```
      "processSurvey");
    }

  }
/// End of Section 3 ///////////////
```

15. Save your changes.
16. Right-click the **WebSphereCommerceServerExtensionsLogic** project and select **Build Project**.

Modifying the command registry

You must modify the command registry so that the new implementation class is used in the shopping flow.

To modify the command registry, do the following:

1. DB2 If you are using a DB2 database, do the following to register MyPostUserRegistrationAddCmdImpl:
   a. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Center**).
   b. From the **Tools** menu, select **Tools Settings**.
   c. Select the **Use statement termination character** check box and ensure the character specified is a semicolon (**;**)
   d. With the Script tab selected, create the required entry in the URLREG table, by entering the following information in the script window:

   ```
   connect to developmentDB user dbuser using dbpassword;
   insert into CMDREG values (FashionFlow_storeent_Id,
   'com.ibm.commerce.usermanagement.commands.PostUserRegistrationAddCmd'
   'Command for modified user bean tutorial',
   'com.ibm.commerce.sample.commands.MyPostUserRegistrationAddCmdImpl',
   null, null, 'Local');
   ```

   where
   - *developmentDB* is the name of your development database
   - *dbuser* is the database user
   - *dbpassword* is the password for your database user
   - *FashionFlow_storeent_Id* is the unique store entity identifier for your store.

   Click the **Execute** icon.
2. Oracle If you are using an Oracle database, do the following to register MyPostUserRegistrationAddCmdImpl:
   a. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
   b. In the **User Name** field, enter your Oracle user name.
   c. In the **Password** field, enter your Oracle password.

d. In the **Host String** field, enter your connect string.

e. In the SQL Plus window, enter the following SQL statement:

```
insert into CMDREG values (FashionFlow_storeent_Id,
'com.ibm.commerce.usermanagement.commands.PostUserRegistrationAddCmd'
'Command for modified user bean tutorial',
'com.ibm.commerce.sample.commands.MyPostUserRegistrationAddCmdImpl',
null, null, 'Local');
```

Press Enter to run the SQL statement.

f. Enter the following to commit your database changes:

```
commit;
```

and press Enter to run the SQL statement.

Modifying JSP templates to collect and display housing information

In this step, you modify the UserRegistrationAddForm and UserRegistrationUpdateForm templates so that the customer can input housing information when logging in, as well as view the housing information in a summary page. The strategy for modifying these pages is to include additional JSP templates that specify the new information for the page. These new pages (UserRegistrationAddFormInclude.jsp and UserRegistrationUpdateFormInclude.jsp) use the JSTL to display the new information.

To modify these pages, do the following:

1. Switch to the Web perspective.
2. If you have not completed Chapter 10, “Tutorial: Creating new business logic,” on page 217, you must modify the properties of the Stores web project, as follows:
   a. Right-click the **Stores** Web project and select **Properties**.
   b. Business Professional Select **Web** in the left pane and then from the list of Available Web Project Features, select **Include the JSP Standard Tag Library**.

      Express Select **Web Project Features** in the left pane and then from the list of Available Web Project Features, select **JSP Standard Tag Library**.

      Click **Apply**. When the update is complete, click **OK** to close the properties editor.
3. Expand to the following directory: **Stores\Web Content***FashionFlow_name*.
4. Create a backup copy of the UserRegistrationAddForm.jsp file, by doing the following:

a. Expand the **UserArea>AccountSection>RegistrationSubsection** directories.
b. Right-click the **UserRegistrationAddForm.jsp** file and select **Rename**.
c. In the Rename window, enter `UserRegistrationAddForm_bak.jsp` and click **OK**.
d. When prompted if you would like to update links to this file, click **No**.

5. Create a backup copy of the UserRegistrationUpdateForm.jsp file, by doing the following:
   a. Expand the **UserArea>AccountSection>RegistrationSubsection** directories.
   b. Right-click the **UserRegistrationUpdateForm.jsp** file and select **Rename**.
   c. In the Rename window, enter `UserRegistrationUpdateForm_bak.jsp` and click **OK**.
   d. When prompted if you would like to update links to this file, click **No**.
6. Right-click the *FashionFlow_name* directory and select **Import**.
   The Import wizard opens.
7. From the **Select an import source** list, select **Zip file** and click **Next**.
8. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located, as follows:
   yourDirectory\WC_SAMPLE_55.zip
   where *yourDirectory* is the directory into which you downloaded the package.
9. Click **Deselect All**, then expand the directories and select the following files to import.
   - UserArea\AccountSection\RegistrationSubsection\UserRegistrationAddForm.jsp
   - UserArea\AccountSection\RegistrationSubsection\UserRegistrationAddFormInclude.jsp
   - UserArea\AccountSection\RegistrationSubsection\UserRegistrationUpdateForm.jsp
   - UserArea\AccountSection\RegistrationSubsection\UserRegistrationUpdateFormInclude.jsp
10. In the **Folder** field, the `Stores/Web Content/`*`FashionFlow_name`* folder is already specified. Keep this value.
11. Click **Finish**.

If you examine the UserRegistrationAddForm.jsp file, you will find that the following section has been added for this tutorial:

```
<%-- Add for tutorial --%>
   <tr>
      <td colspan="3">
         <jsp:include page="UserRegistrationAddFormInclude.jsp" flush="true" />
      </td>
   </tr>
<%-- End of tutorial --%>
```

You can also examine the UserRegistrationAddFormInclude.jsp file to see how the new information is collected, using JSTL. Similarly, examine the UserRegistrationAddForm.jsp and UserRegistrationAddFormInclude.jsp files.

You must also import the properties file that contains the string values used in the modified JSP templates. This file is called Housing.properties. To import this file, do the following:

1. In the Business Professional J2EE Navigator view Express Project Navigator view, expand the following directories:
   Stores > Web Content> WEB-INF >classes > *FashionFlow_name* directory.
2. Right-click the *FashionFlow_name* directory and select **Import**.
   The Import wizard opens.
3. From the **Select an import source** list, select **Zip file** and click **Next**.
4. Click **Browse** (beside the **Zip file** field) and navigate to the sample code. This file is located, as follows:
   yourDirectory\WC_SAMPLE_55.zip
   where *yourDirectory* is the directory into which you downloaded the package.
5. Click **Deselect All**, then expand the directories and select the following file to import.
   - Housing.properties
6. In the **Folder** field, the `Stores/Web Content/WEB-INF/classes/`*`FashionFlow_name`* folder is already specified. Keep this value.
7. Click **Finish**.

Testing the modified code

Next you must test the modified registration information, by doing the following:

1. Switch to the Servers perspective.
2. Right-click the **WebSphereCommerceServer** test server and select **Start**.
3. Right-click **index.jsp** under the Stores\Web Content*FashionFlow_name* directory and select **Run on Server**.
4. When the home page for the store opens, click **Register**.
5. Click **Register** again to create a new user.

6. You are presented with a modified registration page, that now has the housing survey, as shown in the following screen shot:

Figure 53.

7. Enter information for the new user, as appropriate and click **Submit**.
8. After the information has been submitted, click **Change Personal Information** to verify that the housing information has been captured. You are presented with a screen that shows a summary of your survey responses, as follows:

Figure 54.

Deploying the housing survey logic

This section describes how to deploy your new business logic into a store running on a remote WebSphere Commerce Server. You must have created a store (based upon the FashionFlow sample store) on the remote WebSphere Commerce Server before starting these deployment steps.

The deployment process includes steps that are performed on the development machine, as well as steps that are performed on the target WebSphere Commerce Server.

There are a number of different types of assets that must be deployed to the target WebSphere Commerce Server. These include:

- Command logic
- Modified enterprise bean logic
- JSP templates
- A properties file
- Database updates including schema updates (new table) as well as command registry updates

This section describes how to deploy all of these assets, *incrementally* to the target WebSphere Commerce Server.

Creating the command JAR file

This section describes how to create the JAR file that contains the new MyPostUserRegistrationAddCmdImpl logic.

To create this JAR file, perform the following steps on your development machine:

1. Create a directory on your local file system called \ExportTemp4.
2. Open WebSphere Studio Application Developer and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
3. Right-click the **WebSphereCommerceServerExtensionsLogic** project and select **Export**.
   The Export wizard opens.
4. In the Export wizard, do the following:
   a. Select **JAR file** and click **Next**.
   b. The left pane under **Select the resources to export** is prepopulated with the name of the project. Leave this value as is.
   c. In the right pane ensure that only the following resources are selected:
      - .classpath
      - .project
      - .serverPreference

d. Ensure that **Export generated class files and resources** is selected.
e. Do *not* select **Export Java source files and resources**.
f. In the **Select the export destination** field, enter the fully-qualified JAR file name to use. In this case, enter `*drive*:\ExportTemp4\WebSphereCommerceServerExtensionsLogic.jar`. Note that the JAR file name must be WebSphereCommerceServerExtensionsLogic.jar.
g. Click **Finish**.

Creating the EJB JAR file

To create the EJB JAR file, do the following:

1. Open WebSphere Studio Application Developer and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the **Member-MemberManagementData** project.
3. Double-click **EJB Deployment Descriptor**.
4. With the Overview tab selected, scroll to the bottom of the pane, to locate the **WebSphere Bindings** section.
5. In the **DataSource JNDI name** field, enter the datasource JNDI name of the target WebSphere Commerce Server. The following is an example value:

   DB2 `jdbc/WebSphere Commerce DB2 DataSource demo`
   where the target WebSphere Commerce Server is using a DB2 database, and the WebSphere Commerce instance name is "demo"

   Oracle `jdbc/WebSphere Commerce Oracle DataSource demo`
   where the target WebSphere Commerce Server is using an Oracle database, and the WebSphere Commerce instance name is "demo".
6. Save your deployment descriptor changes (Ctrl+S).
7. In the Business Professional J2EE Navigator view Express Project Navigator view, right-click the **Member-MemberManagementData** project and select **Export**.
   The Export wizard opens.
8. In the Export wizard, do the following:
   a. Select **EJB JAR file** and click **Next**.
   b. Business Professional The value for **What resources do you want to export?** is prepopulated with the name of the EJB project.

      Express The EJB project name is prepopulate.
      Leave this value as is.
   c. Business Professional In the **Where do you want to export resources to?** field, enter the fully-qualified JAR file name to use.

      Express For the destination, enter the fully-qualified JAR file name to

use.
In this case, enter `drive:\ExportTemp4\Member-MemberManagementData.jar`.

d. Click **Finish**.

9. After the JAR file has been created, undo the changes to the local deployment descriptor that was made in step 5, to restore the setting that is required for your local test server.

Exporting store assets

To export the modified JSP templates and the new properties file, do the following:

1. Open WebSphere Studio Application Developer and switch to the Business Professional J2EE Navigator view Express Project Navigator view.
2. Expand the **Stores** folder.
3. Right-click the **Web Content** folder and select **Export**.
The Export Wizard opens.
4. In the Export wizard, do the following:
   a. Select **File system** and click **Next**.
   b. Select the following resources to deploy:
      - Web Content*FashionFlow_name*\UserArea\AccountSection\RegistrationSubsection\UserRegistrationAddForm.jsp
      - Web Content*FashionFlow_name*\UserArea\AccountSection\RegistrationSubsection\UserRegistrationAddFormInclude.jsp
      - Web Content*FashionFlow_name*\UserArea\AccountSection\RegistrationSubsection\UserRegistrationUpdateForm.jsp
      - Web Content*FashionFlow_name*\UserArea\AccountSection\RegistrationSubsection\UserRegistrationUpdateFormInclude.jsp
      - Web Content\WEB-INF\lib\jstl.jar
      - Web Content\WEB-INF\lib\standard.jar
      - Web Content\WEB-INF\classes*FashionFlow_name*\Housing.properties
   c. Select **Create directory structure for selected files**.
   d. In the Directory field, enter a temporary directory into which these resources will be placed. For example, enter `C:\ExportTemp4`
   e. Click **Finish**.

Transferring assets to your target WebSphere Commerce Server

In this step, you create a temporary directory on the target WebSphere Commerce Server and then copy your housing survey assets into this directory. In subsequent steps, you will place the different types of code into the appropriate place within your WebSphere Commerce application.

To copy the files from your development machine to your target WebSphere Commerce Server, do the following:

1. On the target WebSphere Commerce Server, create a temporary directory called \ImportTemp4.
2. Determine how you will copy your files from one computer to another. You can do this by mapping a drive on the target WebSphere Commerce Server to the development machine, or by using an FTP application, if you have that configured.
3. From the development machine, copy the contents of \ExportTemp4 into \ImportTemp4 on the target WebSphere Commerce Server.

Stopping your target WebSphere Commerce Server

Before starting the deployment steps, you should stop your target WebSphere Commerce Server by issuing the stopServer command at the command line.

For details about this command, refer to the Business Professional *WebSphere Commerce Studio Installation Guide* or Express *WebSphere Commerce - Express Developer Edition Installation Guide*.

Updating the database on your target WebSphere Commerce Server

Creating the XHOUSING table

DB2 If you are using a DB2 database, do the following to create the table:

1. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Line Tools > Command Center**) and click the **Scripting** tab.
2. In the Script window, enter the following:

```
connect to developmentDB user dbuser using dbpassword;
create table XHOUSING (MEMBERID bigint not null,
   HOUSINGTYPE integer, LOCATION integer,
   constraint p_xhousing primary key (MEMBERID),
   constraint f_xhousing foreign key (MEMBERID)
   references USERS (USERS_ID) on delete cascade);
insert into XHOUSING (MEMBERID) (select USERS_ID from USERS);
```

   where

   - *developmentDB* is the name of your development database
   - *dbuser* is the database user
   - *dbpassword* is the password for your database user

   Click the Execute icon.
   You should see a message indicating that the SQL statement completed successfully.

Oracle If you are using an Oracle database, do the following to create the table:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

```
create table XHOUSING (MEMBERID number not null,
   HOUSINGTYPE integer, LOCATION integer,
   constraint p_xhousing primary key (MEMBERID),
   constraint f_xhousing foreign key (MEMBERID)
   references USERS (USERS_ID) on delete cascade);
insert into XHOUSING (MEMBERID) (select USERS_ID from USERS);
```

   and press Enter to run the SQL statement. The XHOUSING table is now created.
6. Enter the following to commit your database changes:

```
commit;
```

   and press Enter to run the SQL statement.

Registering the task command

To modify the command registry, do the following:

1. DB2 If you are using a DB2 database, do the following to register MyPostUserRegistrationAddCmdImpl:
   a. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Center**).
   b. From the **Tools** menu, select **Tools Settings**.
   c. Select the **Use statement termination character** check box and ensure the character specified is a semicolon (**;**)
   d. With the Script tab selected, create the required entry in the URLREG table, by entering the following information in the script window:

```
connect to developmentDB user dbuser using dbpassword;
insert into CMDREG values (FashionFlow_storeent_Id,
'com.ibm.commerce.usermanagement.commands.PostUserRegistrationAddCmd'
'Description',
'com.ibm.commerce.sample.commands.MyPostUserRegistrationAddCmdImpl',
null, null, 'Local');
```

   where

   - *developmentDB* is the name of your development database
   - *dbuser* is the database user
   - *dbpassword* is the password for your database user
   - *FashionFlow_storeent_Id* is the unique store entity identifier for your store.

Click the **Execute** icon.

2. Oracle If you are using an Oracle database, do the following to register MyPostUserRegistrationAddCmdImpl:
   a. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
   b. In the **User Name** field, enter your Oracle user name.
   c. In the **Password** field, enter your Oracle password.
   d. In the **Host String** field, enter your connect string.
   e. In the SQL Plus window, enter the following SQL statement:

      ```
      insert into CMDREG values (FashionFlow_storeent_Id,
      'com.ibm.commerce.usermanagement.commands.PostUserRegistrationAddCmd'
      'Description',
      'com.ibm.commerce.sample.commands.MyPostUserRegistrationAddCmdImpl',
      null, null, 'Local');
      ```

      Press Enter to run the SQL statement.
   f. Enter the following to commit your database changes:

      ```
      commit;
      ```

      and press Enter to run the SQL statement.

Updating store assets on your target WebSphere Commerce Server

In this step, you update the store with your modified store assets, as follows:

1. Backup your *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear\ Stores.war directory (where *cellName* is often the host name of your machine).
2. Navigate to the :\ImportTemp4\Stores\Web Content directory.
3. Copy the *FashionFlow_name* and WEB-INF folders into the following directory: *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear\Stores.war

where *instanceName* is the name of your WebSphere Commerce instance.

Updating the command JAR file on your target WebSphere Commerce Server

In this step you update the target WebSphere Commerce Server to use the new command JAR file, as follows:

1. You should make a backup copy of the existing JAR file, as follows:
   a. Navigate to the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory.
   b. Make a copy of the WebSphereCommerceServerExtensionsLogic.jar file and save it in a backup location.

2. Copy the new WebSphereCommerceServerExtensionsLogic.jar file from the \ImportTemp4 directory into the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory

where *instanceName* is the name of your WebSphere Commerce instance.

Updating the EJB JAR file on your target WebSphere Commerce Server

In this step you update the target WebSphere Commerce Server to use the new EJB JAR file, as follows:

1. You should make a backup copy of the existing JAR file, as follows:
   a. Navigate to the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory.
   b. Make a copy of the Member-MemberManagementData.jar file and save it in a backup location.
2. Copy the new Member-MemberManagementData.jar file from the \ImportTemp4 directory into the *WAS_installdir*\installedApps*cellName*\WC_*instanceName*.ear directory
3. Next, you must modify the EJB deployment descriptor information, as follows:
   a. Locate the deployment repository (META-INF directory) for this WebSphere Application Server cell. This typically takes the following form:
      WAS_installdir\config\cells*cellName*
      \applications\WC_*instance_name*.ear\deployments\
      WC_*instance_name**EJBModuleName*.jar\META-INF.
      The following is a specific example of this:
      D:\WebSphere\AppServer\config\cells\myCell\applications\
      WC_demo.ear\deployments\WC_demo\ Member-
      MemberManagementData.jar\META-INF
      where
      - myCell is the name of the WebSphere Application Server cell
      - demo is the name of the WebSphere Commerce instance
      - Member-MemberManagementData is the name of the customized EJB module
   b. The directory contains the following files:
      - ejb-jar.xml
      - ibm-ejb-access-bean.xmi
      - ibm-ejb-jar-bnd.xmi
      - ibm-ejb-jar-ext.xmi
      - MANIFEST.MF

      Backup all of these files.

c. Use a tool to open the new Member-MemberManagementData.jar file and view its contents.
d. Extract the contents of the meta-inf directory from this Member-MemberManagementData.jar file into the directory from step 3a.

4. Using the WebSphere Application Server startServer command at the command line, restart your WebSphere Commerce instance.

Verifying the housing survey logic on the target WebSphere Commerce Server

In this step, you verify that the housing survey logic has been successfully deployed to the target WebSphere Commerce Server by doing the following:

1. Open a Web browser and enter the URL to launch your store that is based on the FashionFlow sample store.
2. When the home page for the store opens, click **Register**.
3. Click **Register** again to create a new user.
4. You are presented with a modified registration page, that now has the housing survey, as shown in the following screen shot:

Send me e-mails about specials and featured clothing.
Would you like to receive e-mails about:
Our men's fashions
Our women's fashions
Our specials
Housing Survey Information
Housing Type Townhouse
Location Downtown
Submit

Figure 55.

5. Enter information for the new user, as appropriate and click **Submit**.

6. After the information has been submitted, click **Change Personal Information** to verify that the housing information has been captured. You are presented with a screen that shows a summary of your survey responses, as follows:

Housing Survey Information

Housing Type: Townhouse

Location: Downtown

Figure 56.

Part 5. Appendixes

© Copyright IBM Corp. 2000, 2003

Appendix A. Configuring WebSphere Commerce component tracing in the WebSphere Commerce development environment

This appendix describes how to enable tracing for the various WebSphere Commerce components, when running in the WebSphere Commerce development environment. To enable component tracing, do the following:

1. If required, open the WebSphere Commerce development environment (**Start > Programs > IBM WebSphere Commerce development environment > WebSphere Commerce development environment**).
2. Switch to the Server perspective.
3. In the Servers view, right-click **WebSphereCommerceServer** and select **Stop** (if the server is currently running).
4. In the Server Configuration view, expand the **Server Configurations** folder.
5. Double-click **WebSphereCommerceServer**.
   The WebSphereCommerceServer editor opens.
6. Select the Trace tab.
7. Select **Enable trace**.
8. In the **Trace string** text box, specify the components for which tracing should be enabled. Use the WebSphere JRas extensions trace logger identifier value, followed by `"=all=enabled`". For a complete list of these values, refer to the "Configuration" topic of the *WebSphere Commerce Administration Guide*. Multiple components should be separated by a colon (:). As an example, to enable tracing for both the SERVER and RAS components, specify the trace string as follows:
   ```
   com.ibm.websphere.commerce.WC_SERVER=all=enabled:
     com.ibm.websphere.commerce.WC_RAS=all=enabled
   ```
   Note the line break is for presentation purposes only.
9. Save your changes (Ctrl+S).

Output file

The output log file is called activity.log by default. This file is located in the following directory:
workspace_dir\.metadata\.plugin\com.ibm.etools.server.core\tmp0\logs

© Copyright IBM Corp. 2000, 2003

Due to the fact that the activity.log file is a binary file, Log Analyzer is used to read this file. Once you have enabled component tracing, WebSphere JRas will also write the log entries in plain text format into the trace output file along with the trace entries.

For information about configuring the Log Analyzer tool in the WebSphere Commerce development environment, refer to the ▶ Business ▶ Professional *WebSphere Commerce Studio Installation Guide* or ▶ Express *WebSphere Commerce - Express Developer Edition Installation Guide*.

Additionally, messages are displayed in the Console view of WebSphere Studio Application Developer.

Appendix B. Where to find more information

More information about the WebSphere Commerce development environment system and its components is available from a variety of sources in different formats. The following sections indicate what information is available and how to access it.

WebSphere Commerce development environment information

The following are the sources of the WebSphere Commerce development environment information:

- "WebSphere Commerce development environment online help"
- "WebSphere Commerce Web site" on page 382
- "WebSphere Developer Domain" on page 382
- "IBM Redbooks™" on page 382

WebSphere Commerce development environment online help

The WebSphere Commerce development environment online information is your primary source of information for creating and publishing stores in the WebSphere Commerce development environment.

To view the WebSphere Commerce development environment online help, do the following:

1. Start the WebSphere Commerce development environment by selecting **Start → Programs → IBM WebSphere Commerce Studio → WebSphere Commerce development environment**.
2. From the **Help** menu, select **Help Contents**.

Note: If you refer to instructions for multiple platforms in the "WebSphere Commerce development environment online help," ensure you follow the instructions for the WebSphere Commerce development environment. When a help page contains information for multiple platforms, information specific to the WebSphere Commerce development environment is indicated with the following icon:

If information specific to the WebSphere Commerce development environment is not available, ensure you follow instructions specific to Windows, When a help page contains instructions for multiple platforms, instructions for Windows are indicated with the following

icon:

Windows

WebSphere Commerce Web site

WebSphere Commerce development environment product information is available at the WebSphere Commerce Web site. Refer to the following URLs for more product information:

`http://www.ibm.com/software/webservers/commerce/library/`

WebSphere Developer Domain

Additional information on the WebSphere Commerce development environment and WebSphere Commerce is also available in the WebSphere Commerce Zone at WebSphere Developer Domain:

`http://www.ibm.com/websphere/developer/zones/commerce/`

IBM Redbooks™

WebSphere Commerce development environment and WebSphere Commerce information is available at the IBM Redbooks Web site:

`http://www.ibm.com/redbooks`

WebSphere Studio Application Developer information

The following are sources of information for WebSphere Studio Application Developer:

- "WebSphere Studio Application Developer online help"
- "WebSphere Studio Application Developer Web site"
- "WebSphere Developer Domain" on page 383
- "IBM Redbooks" on page 383

WebSphere Studio Application Developer online help

The WebSphere Studio Application Developer online help is your primary source of information on how to perform tasks within WebSphere Studio Application Developer.

To view the WebSphere Studio Application Developer online help, do the following:

1. Start the WebSphere Commerce development environment by selecting **Start → Programs → IBM WebSphere Studio → Application Developer 5.0**.
2. From the **Help** menu, select **Help Contents**.

WebSphere Studio Application Developer Web site

WebSphere Studio Application Developer product information is available on the WebSphere Studio Application Developer Web site:

`http://www.ibm.com/software/ad/studioappdev/library/`

WebSphere Developer Domain

Additional information on WebSphere Studio Application Developer is available on the WebSphere Studio Application Developer page of the WebSphere Studio Zone of WebSphere Developer Domain:

http://www.ibm.com/websphere/developer/zones/studio/appdev/

IBM Redbooks

WebSphere Studio Application Developer information is available at the IBM Redbooks Web site:

http://www.ibm.com/redbooks

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

© Copyright IBM Corp. 2000, 2003

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Ltd.
Office of the Lab Director
8200 Warden Avenue, Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM

products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

©Copyright International Business Machines Corporation 2000, 2003. Portions of this code are derived from IBM Corp. Sample Programs. ©Copyright IBM Corp. 2000, 2003. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The IBM logo and the following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries or both:

400	IBM
AIX	iSeries
AS/400	OS/390
DB2	WebSphere
DB2 Universal Database	z/OS

Windows is a trademark or registered trademark of Microsoft® Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, and service names may be the trademarks or service marks of others.

Index

A

access control 89
- command-leval 99
- Groupable interface 104
- policies 92
- Protectable interface 104
- protecting resources 105
- resource-leval 99

adapters 10
application architecture 4

C

CMDREG 30
command design pattern 20
command flow 26
command registry 28
commands
- command context 137
- customize existing 149
- factory 23
- framework 21
- implementation 133
- interfaces 22
- registration 28
- types 12
- writing new business policy commands 163
- writing new controller commands 139
- writing new task commands 148

controller command invoker data bean 42
controller commands
- customize existing 149
- long-running 142
- writing new 139

customized code
- packaging 136

D

data beans
- activating 41
- BeanInfo 41
- customize existing 156
- description 14
- interfaces 39
  - command data bean 40
  - input data bean 41

data beans *(continued)*
- interfaces *(continued)*
  - smart data bean 39
- types 38

database commits 145
database considerations
- datatype 85
- naming 83

database locks 78
deployment descriptors 48
design patterns 19
- command 20
- display 37
- model-view-controller 19

display design pattern 37

E

entity beans
- cache 79
- deployment descriptors 48
- description 13
- extending 50
- overview 47
- transactions 77
- using 82

error handling 123
- command 123
- exception types 123
- flow 124
- in custom code 126
- JSP 130
- trace 130

F

flushRemote method 79

J

JSP templates 14
- setting attributes 43

M

messages
- creating messages 128
- properties files 124

model-view-controller design pattern 19

O

object life cycles 77
object model extension methodologies 50

P

packaging customized code 136
persistence 47
protocol listeners 9

R

relationship groups 97
run-time architecture 7

S

servlet engine 9
session beans
- recommended use 54
- writing new 75

software components 3

T

task commands
- customize existing 154
- writing new 148

terms and conditions 169
tracing execution flow 130
trading agreements 157
transaction scope 145

U

URLREG 28

V

view commands
- format input properties to 143
- required properties 45

VIEWREG 33

W

Web controller 11

© Copyright IBM Corp. 2000, 2003

IBM®

Printed in USA