BEA WebLogic Platform™

Tour of the WebLogic Platform Sample Application

Release 7.0 Service Pack 1
Document Date: September 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Tour of the BEA WebLogic Platform Sample Application

Part Number	Date	Software Version
N/A	September 2002	7.0 Service Pack 1

Contents

4. Web Services Tour

About This Document

This document presents a tour of the BEA WebLogic Platform sample application.

This document is organized as follows:

- Chapter 1, "Introduction," contains an overview of the sample's features and explains how to start the sample.
- Chapter 2, "Business-to-Consumer (B2C) Portal Tour," describes a fictitious portal Web site for Avitek Digital Imaging. The site demonstrates a combination of product features, including the portal framework in WebLogic Portal, the use of two Web services built in WebLogic Workshop, plus inventory checks and order management provided by WebLogic Integration components.
- Chapter 3, "Avitek Purchasing Agents Connect with Suppliers," describes a fictitious intranet site that Avitek purchasing agents use to get quotes from suppliers, and to submit purchase orders and get acknowledgements from suppliers. The business processes are managed by WebLogic Integration.
- Chapter 4, "Web Services Tour," describes how two Web services were built in WebLogic Workshop, and the steps in the Portlet Wizard to generate Web service interface code that can be used in a portlet.

Audience

This document is written for product evaluators, application developers, and system administrators who are evaluating or using WebLogic Platform software. It is assumed that readers know Web technologies and have a general understanding of Windows and UNIX systems.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at http://e-docs.bea.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, using the File—>Print option on your Web browser.

A PDF version of this document is available on the WebLogic Platform documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Platform documentation Home page, click Download Docs, and select the document you want to print.

If you do not have Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Platform. Other documents that you may find helpful when using the WebLogic Platform sample application include:

- *Introducing BEA WebLogic Platform*
- *Introduction to BEA WebLogic Server*
- *BEA WebLogic Workshop online Help*
- *Introducing BEA WebLogic Integration*

- *BEA WebLogic Portal Administration Guide*
- *BEA WebLogic Portal Development Guide*

Contact Us!

Your feedback on BEA WebLogic Platform documentation is important to us. Send us e-mail at `docsupport@bea.com` if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate which release of the WebLogic Platform documentation you are using.

If you have any questions about this version of BEA WebLogic Platform, or if you have problems installing and running BEA WebLogic Platform, contact BEA Customer Support through BEA WebSupport at `http://www.bea.com`. You can also contact Customer Support using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis and book titles.

`monospace text`	Indicates code samples, commands and their options, Java classes, data types, directories, and filenames and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`CustomerName;`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 BEA_HOME OR

Convention	Usage
{ }	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
[ ]	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed. *Example*: `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
. . .	Indicates one of the following in a command line: ■ That an argument can be repeated several times in the command line. ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Introduction

BEA WebLogic Platform provides a standards-based, *build-to-integrate* approach that enables companies to develop and deploy applications, rapidly integrate with existing systems, automate business processes, and connect with business partners.

To support its build-to-integrate approach and enable WebLogic Platform feature interoperation, specific integration entry points are available to facilitate process-level communication and data flow between front-end Web applications and heterogeneous back-end systems. An *integration entry point* defines a place at which interoperation can take place.

The WebLogic Platform sample application is designed to highlight a number of these key integration points. It is also designed to be fun!

The sample includes a business-to-consumer (B2C) portal Web site for a fictitious company named Avitek Digital Imaging. Customers who visit the Avitek portal can purchase cameras and accessories. This Web site includes a product catalog, the use of two Web services, inventory checks, a full shopping cart and underlying processing, payment authorization, and order management.

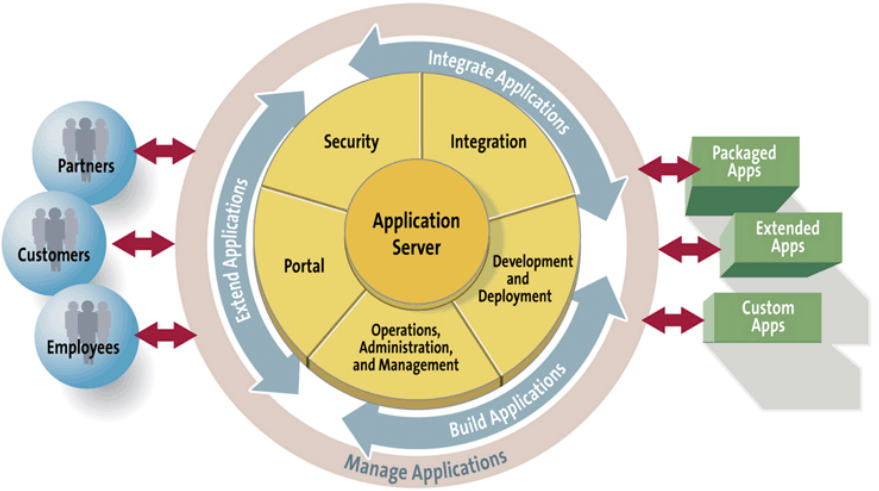
The sample also shows a business-to-business (B2B) portal site. The Avitek intranet allows its purchasing agents to get quotes for product parts from external suppliers, select a quote, generate a purchase order for the part, and exchange acknowledgements with the selected supplier.

To help you understand the technology behind each page and event, the B2C and B2B samples also provide a built-in tour guide. The dynamic documentation content that is shown in the tour guide is based on the last event that occurred in the running Web application. This online book version of the tour presents a static view of the application's pages, but contains much of the same information. Read this online book if you prefer to read the explanations apart from the running application.

This section presents the following topics:

WebLogic Platform Architecture

WebLogic Platform provides an integrated set of features that include the J2EE-compliant application server, plus the development, portal, and integration frameworks that are built on top of it. The following figure shows how WebLogic Platform provides a single, highly integrated solution:



- The application server provides the foundation, via BEA WebLogic Server, for rapidly developing, deploying, and managing e-business applications.

- The development framework delivers an integrated development and run-time environment, including support for enterprise-class Web services. This includes Web services you can program with a WebLogic Server API, or Web services you can create using BEA WebLogic Workshop.
- The portal framework delivers, via BEA WebLogic Portal, services that enable you to efficiently build, launch, and maintain high-performance e-business sites. This framework facilitates the creation, customization, and administration of multiple portals. WebLogic Portal includes many related features, including services that let you deliver personalized content, Web-based marketing campaigns, product catalogs, and order fulfillment systems that includes templates for a shopping cart, payment services, and tax services.
- The integration framework enables, via BEA WebLogic Integration, the integration of enterprise information systems via standards-based integration technology, including enterprise resource planning (ERP), supply chain management (SCM), human resource (HR), and customer relationship management (CRM), as well as custom and legacy applications. This framework also enables collaboration between suppliers and partners, and the automation of business process workflows.

For a detailed overview, see *Introducing WebLogic Platform*.

Starting the Sample

The easiest way to start the WebLogic Platform sample is from the QuickStart application. QuickStart is designed to assist first-time users in evaluating, learning, and using WebLogic Platform. In addition to providing quick access to this and other samples, QuickStart provides pointers to useful tools for accomplishing specific development tasks, and to the online documentation.

QuickStart runs automatically after you complete a Typical or full Custom installation of WebLogic Platform. Because the sample includes live Web applications, you cannot run the sample if you performed a partial installation of the WebLogic Platform software. For more information about the installation process, see Installing WebLogic Platform.

The following screen shows the QuickStart options.

On the QuickStart page, select the "Take the Live Platform Tour" link. It invokes the sample's startup script, `E2Estart.bat` (Windows) or `E2Estart.sh`. The script file is located in the following directory under your `BEA_HOME` location:

```
weblogic700/samples/platform/e2eApp/config
```

After installation, you can launch QuickStart as follows:

- On Window systems, choose Start→Programs→BEA WebLogic Platform 7.0→QuickStart
- On UNIX systems, perform the following steps:

  a. Log in to the target UNIX system.

b. Open a command-line shell.

c. Go to the `/common/bin` subdirectory of the WebLogic Platform installation. For example:

```
cd /home/bea/weblogic700/common/bin
```

d. Enter the following command:

```
sh quickstart.sh
```

After the server starts, your default browser should open automatically if you used QuickStart. If you ran the `E2Estart.bat` or `E2Estart.sh` script directly, after the server has started and you see the "WebLogic Server started on..." confirmation message in the command window, open a supported browser and point to:

```
http://<host>:<port>
```

For example, if you are running the server on your local machine and used the default port number 7501, specify in the URL:

```
http://localhost:7501
```

If the server is running on a remote machine named (for example) `blues` and you used the default port, specify in the URL:

```
http://blues:7501
```

For information about the browsers that are supported by WebLogic Platform, see the Supported Platforms documentation.

The Sample's Introduction Page

After the server has started, as explained in the previous section, the sample application displays its Introduction page. It consists of five main sections. Each section is summarized here.

Section 01 Provides Introductory Text

Section 01 on the Introduction page is informational and does not contain links to the running sample. It simply presents an annotated screen shot of the tour guide portlet that you can use in the 02 (B2C) and 03 (B2B) Web applications. For example:

As you move from one page to another in the B2C or B2B portal, the tour guide can help you understand what's new and the next step to follow.

Each tour guide page also contains links to:

- View the code of a portlet on the portal page
- Read a technical explanation about the processing behind the current page or most recently used portlet
- Get a list of links to related documentation on the BEA e-docs Web site

The Introduction page's section 01 also shows samples of two buttons used in the B2C and B2B portals. For example:

You can click either of these buttons to exit the B2C or B2B portal and return to the sample's Introduction page.

Section 02 Provides the Link to the B2C Portal

Section 02 on the Introduction page allows you to log into the B2C portal as "Rachel Adams." Rachel is an already registered customer of the fictitious Avitek Digital Image Web site. This part of the sample application includes a product catalog, the use of two Web services, inventory checks, a full shopping cart and underlying processing, payment authorization, and order management that includes passing XML order data to an integrated system. For example:

Section 03 Provides the Link to the B2B Portal

Section 03 on the Introduction page allows you to log into the B2B portal as “Jason Tang,” an Avitek purchasing agent. Jason uses the Avitek intranet to get quotes for product parts from external suppliers, select a quote, generate a purchase order for the part, and exchange acknowledgements with the selected supplier.

Section 04 Provides the Link to a Web Services Tech Tour

Section 04 on the Introduction page links to a documentation-only Web services technical tour. For example:

It explains how we used WebLogic Workshop to define the separate Web services for Product Evaluator and Payment. This part of the tour also shows the steps in Portlet Wizard to generate the Web service interfaces code for the Product Evaluator portlet that you will see in the 02 B2C portal.

Survey Section

Finally, the Introduction page also contains a link to a brief, anonymous survey that you can take to rate your satisfaction with installing and evaluating WebLogic Platform. For example:

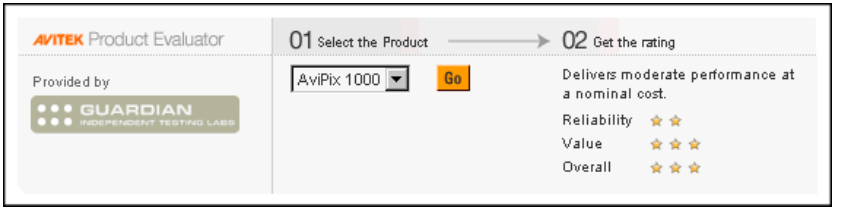
Integration Points

This sample application demonstrates several areas of technology integration with the components that comprise WebLogic Platform. The integration points include:

- Product Evaluator Web Service and Portlet
- Payment Web Service
- Orders Generated in WebLogic Portal Are Processed by WebLogic Integration Via JMS Queue and BPM
- Real-time Inventory Checks Via WebLogic Integration AI

Product Evaluator Web Service and Portlet

The business-to-consumer (B2C) sample portal includes a “Product Evaluator” portlet that returns a product rating for a selected product item. You will see this portlet near the bottom of pages with the Products tab when you browse through the catalog's categories. For example:



The portlet includes code that points to a Web service, `productEvalWSC`, which we created using WebLogic Workshop.

A **Web service** is a language-independent, platform-independent, self-describing code module that applications can access via a network or the Internet. The application can have the service's location hard-coded or can locate it using UDDI (Universal Description, Discovery, and Integration). Because the service is self-describing, the application can determine which functions are available and how to call them.

In development, we used a browser-based test form provided by WebLogic Workshop to check whether the expected results were being returned by the Web service. For information about the test form's features and the portlet, see the section "The Product Evaluator Portlet and Web Service" on page 2-33. Also see Chapter 4, "Web Services Tour," for information about the development steps we followed in WebLogic Workshop and, separately, in the Portlet Wizard that comes with WebLogic Portal.

Payment Web Service

We also used WebLogic Workshop to design a Web service that performs payment authorization, capture, and settlement. In this portal application, after you click the Submit Order button in the shopping cart, the credit card information on the page is authorized via the Web service. In this sample, although the predefined credit card data and the Web service itself are stored locally, the code is a working example of the processing required.

The Payment Web service is "conversational" — the `authorize()` call starts the conversation; the `capture()` call continues the conversation; and the `settle()` call finishes or ends the conversation.

This Web service is called from a pipeline component named `CajunBasedPaymentPC.java`. The pipeline component uses a proxy to interact with this Web service. Error codes are returned in case there is a problem with the Payment authorization, capture, and settle methods. For details, see the `Payment.jws` file. The proxy used is generated via a WebLogic Server clientgen task.

Information about the Payment Web service and the Portlet Wizard is provided in the Web Services Technical Tour, which is available from this sample's Introduction page. For details, see Chapter 4, "Web Services Tour."

Orders Generated in WebLogic Portal Are Processed by WebLogic Integration Via JMS Queue and BPM

In the b2cPortal tour, after the logged-in user clicked a Submit Order button, the order generated in WebLogic Portal can be used in business processes. The order is converted to an XML representation and then placed on a Java Message Service (JMS) queue in WebLogic Integration. The business process management (BPM) component of WebLogic Integration then processes the order. For more information, see Chapter 2, "Business-to-Consumer (B2C) Portal Tour."

In the b2bPortal tour, Avitek purchasing agents get quotes from external suppliers for parts and then submit purchase orders, exchanging data with a selected supplier. This work involves two separate business processes: the Query for Price and Availability (QPA) business process, and the Purchase Order (PO) business process. WebLogic Integration manages the business conversations and collaboration agreements between business partners, and it automates the business message exchange between the buyer and suppliers. The workflows are referenced in the collaboration agreements and conversations. For details, see Chapter 3, "Avitek Purchasing Agents Connect with Suppliers."

Real-time Inventory Checks Via WebLogic Integration AI

The database for the sample application includes an inventory table. It keeps data about the current, minimum, and maximum inventory for products and parts. The b2cPortal includes a catalog with SKUs for product items sold on the Web site. The b2bPortal uses SKUs for both Products and the parts that comprise them.

For both portals, b2cPortal and b2bPortal, the inventory table is accessed in read-only mode via the Application Integration (AI) component of WebLogic Integration. When the user tries to add an item to the shopping cart, the inventory for that item will be checked to make sure the order can be fulfilled. For example, this check occurs when you click the Buy Now button on several portlets in this b2cPortal's Products page. An inventory check is also performed if the user tries to update the quantity of the item already in the shopping cart by entering a new value on the shopping cart's `step1.jsp` portlet, and then clicking the RECALCULATE button.

2 Business-to-Consumer (B2C) Portal Tour

The business-to-consumer (B2C) portal tour describes the key features that were implemented for the Avitek Digital Imaging portal. The site demonstrates a combination of product features, including the portal framework in WebLogic Portal, the use of two Web services built in WebLogic Workshop, plus inventory checks and order management provided by WebLogic Integration. All these components run on the Web application server environment provided by WebLogic Server.

Note: The information that is presented in this online book is also available in a context-sensitive tour guide portlet that runs as part of the application.

This business-to-consumer (B2C) portal tour contains the following sections:

Outline of Initial Processing

When you started the WebLogic Platform sample application, what happened that resulted in server startup and initial display of the Introduction or "splash" page? First, there were several ways you could have invoked the sample application, for example, from the WebLogic Platform Quick Start Application or Windows Start menu, or directly by running a `startE2E` script.

Regardless of which option you used, the `startE2E.bat` (Windows) or `startE2E.sh` (UNIX) script was invoked. It started a WebLogic Server instance for the application, which runs in a domain named `e2eDomain`. ("`e2e`" is an abbreviation for "end-to-end," meaning a sample that shows a full range of key features.) The word "domain" has many meanings in the computing industry. BEA products use **domain** to mean a collection of servers, services, interfaces, machines, and associated resource managers, all defined by a single configuration file.

When the `startE2E` script runs, it reads configuration information from the enterprise application's `config.xml` file. By default this configuration file resides in the following `BEA_HOME` installed directory:

```
weblogic700/samples/platform/e2eDomain
```

The `config.xml` file includes the following definition, setting `splashPage` as the default Web application in the domain.

```
<WebServer

       DefaultWebApp="splashPage"
       LogFileName="./logs/access.log"
       LoggingEnabled="true"
       Name="e2eServer"
   />
```

With the e2eServer running, specifying a URL such as `http://localhost:7501` in a browser results in running the `splashPage` Web application (or "Webapp") in the following directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\splashPage
```

The `splashPage` webapp's `web.xml` configuration file defines `index.jsp` in the `<welcome-file-list>` definition. This `web.xml` resides in the following directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\splashPage\
WEB-INF
```

On the splash page, you clicked the Login as Rachel Adams button graphic to arrive on this My Avitek page. This action resulted in the splash page passing in the URL for the portal application, plus predefined login credentials.

The splash page sets up requests that allow you to automatically log into the b2cPortal or b2bPortal. This is done to help simplify this sample application. You would not want to embed usernames and passwords in a JSP page. We used a scriptlet to define the login credentials, but do not recommend that you use this approach for your Web applications.

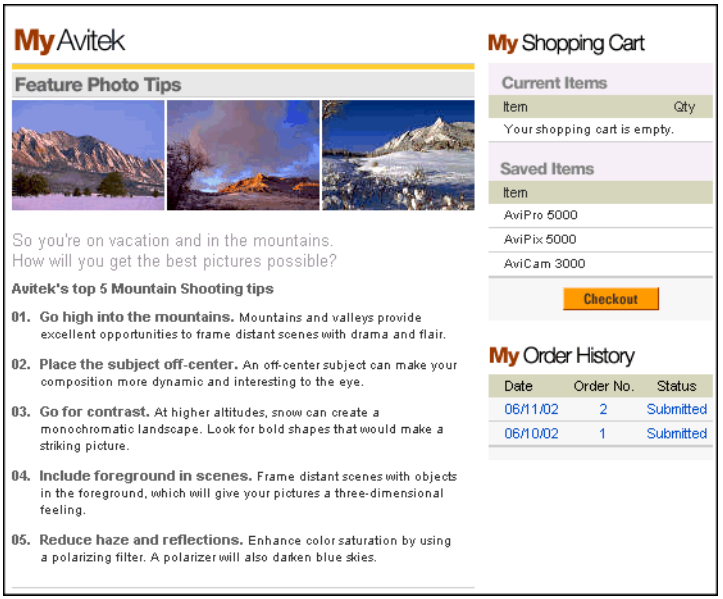
Note: WebLogic Portal provides other samples that demonstrate login authentication code, plus techniques to gather demographic information for Web applications. For details, please refer to the WebLogic Portal documentation.

Once the URL was passed to the browser for the b2cPortal application, why was the My Avitek page the first page displayed? The default page was defined in the WebLogic Portal Administration Tools. For details on this and other characteristics of the My Avitek page, see the next section.

The My Avitek Page

The B2C portal tour starts on the My Avitek page, which presents content that has been personalized for logged-in user Rachel Adams. WebLogic Portal tools were used to organize sets of information into portlets. Personalized content on this page includes Rachel's current shopping cart, her order history, and listing her name next to the Logout button.

If this is your initial tour, and no other user has logged in as Rachel Adams on a shared server instance, the shopping cart and order history are empty. The following screen shows a My Avitek page for a user who has already completed two orders.



Technical Details for the My Avitek Page

This section provides details about the processing that occurs on this page.

Introduction

The My Avitek portal page presents a number of portlets to give the logged-in user a personalized view of the site. The My Avitek page is one of three tabbed pages in the b2cPortal. In the `BEA_HOME` directory where you installed WebLogic Platform, you will find the files that comprise the `e2eApp` in the following locations:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\...
```

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\...
```

The My Avitek page can include the following portlets:

- The **`mybanner`** portlet, which provides the top banner and the three tabs for My Avitek, Products, and Shopping Cart pages.
- The **`login`** portlet, used when an authenticated user has been logged into the application. It provides the Logout button and the name of the currently logged-in user.
- The **`search`** portlet, which provides a keyword-based search input box.
- The **`myavitek`** portlet, which provides the Feature Photo Tips graphics and text on the My Avitek page. (The content shown in this sample is static, but could be personalized for different types of users using features in WebLogic Portal. For information about personalizing application content, see the *WebLogic Portal Development Guide*.)
- The **`summarycart`** portlet, which provides an abbreviated list of the logged-in user's Shopping Cart.
- The **`orderhistory`** portlet, which provides an abbreviated list of the logged-in user's order history.
- The **`tourguide`** portlet, which provides the context-sensitive documentation in the application. It has two forms: the smaller version on the left side of the running application, and the current maximized version that includes Technical Details, View Code, and e-docs pointers.

All of the portlets that comprise the My Avitek page are identified in the following file:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project
\application-sync\webapps\b2cPortal\b2cPortal.portal
```

For example:

```
<page-name>My Avitek</page-name>
...
  <portlet-pool>
      <portlet-name>login</portlet-name>
      <portlet-name>search</portlet-name>
      <portlet-name>tourguide</portlet-name>
      <portlet-name>subnav</portlet-name>
      <portlet-name>summarycart</portlet-name>
      <portlet-name>myavitek</portlet-name>
      <portlet-name>orderhistory</portlet-name>
      <portlet-name>mybanner</portlet-name>
      <portlet-name>anonUser</portlet-name>
  </portlet-pool>
```

During the development cycle, these portlets were added to the My Avitek page using the E-Business Control Center. The EBCC is a Java client-based tool suite. It provides graphical interfaces that simplify complex tasks such as rule definitions, Webflow editing, and portal creation and management. As users of the E-Business Control Center work with its point-and-click interface, it generates XML files that are synchronized with the server.

In addition to the EBCC, the browser-based Portal Administration tools were used for administering and managing the portal at runtime. For information about that process, see the *WebLogic Portal Administration Guide*.

As you run this sample application, it is important to understand that as a developer you are interested in the portlet JSP code before it was rendered by the browser. That's why the code fragments in this section, and the View Code link, describe the prerendered JSP file for a particular portlet.

You can find the b2cPortal's portlet JSP files in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\portlets\...
```

For this My Avitek page, the View Code link opens the source for the `\orderhistory\content.jsp` portlet file. It presents the personalized order history for the logged-in user. This portlet's file resides in the orderhistory subdirectory under the path shown above.

How do we know that the Order History portlet uses the content.jsp in the `...\e2eApp\b2cPortal\portlets\orderhistory` directory? This was specified in the E-Business Control Center provided by WebLogic Platform. This information is also defined in the following file:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project
\application-sync\portlets\orderhistory.portlet
```

This XML file contains the following definition:

```
<portlet-name>orderhistory</portlet-name>
...
<content-url>/portlets/orderhistory/content.jsp</content-url>
```

Notice how the content URL is relative to the root of the b2cPortal Web application, which by default is the following installed directory location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
```

Location in the Default Webflow

Before we start our discussion about the default webflow for the My Avitek page, let's define some terms.

Webflow is a BEA technology that allows developers to control the flow of a Web site, without hard-coding in the presentation JSP the URL of the next page. WebLogic Portal uses Webflow to determine which pages the application should display in a browser and which pieces of business logic it should execute. One or more Webflow namespace files allow you to configure the Webflow for a particular Web application.

A **Webflow namespace file** is an XML file that configures the Webflow for a Web application, controlling the order in which your site's Web pages are displayed and initiating the execution of the business logic that is associated with them. You edit Webflow namespace files using the Webflow and Pipeline Editors, available in the BEA E-Business Control Center. These editors are graphical tools designed to help you visually create, modify, and validate Webflows for your Web applications.

Pipelines and Input Processors are the two different types of processor nodes that come packaged with the Webflow implementation.

A **Pipeline** is a BEA technology that allows developers to bind a sequence of services into a single named service. You can build Pipelines to manage the processing of business data. Generally, Pipelines control the flow of business logic that is executed resulting from Webflow, and they can be transactional or nontransactional. For example, if a visitor attempts to move to another page on your Web site but you want to persist the visitor's information to a database first, you could use a Pipeline. Pipelines contain business logic that may apply to multiple Web applications within a larger enterprise application, and are therefore loaded by the Enterprise JavaBean (EJB) container.

Pipelines are represented as Pipeline Nodes in the Webflow and Pipeline Editors, available in the BEA E-Business Control Center. The Webflow and Pipeline Editors generate underlying Webflow namespace and Pipeline namespace XML files, which you should not hand-edit.

For the b2cPortal application, the Webflow namespace files are located in the following directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\application-sync\webapps\b2cPortal
```

Input Processors are predefined, specialized Java classes that carry out more complex tasks when invoked by the Webflow mechanism. Input Processors are typically used to validate HTML form data, or to provide conditional branching within a Web page. For example, an Input Processor may contain code that verifies whether a date has been entered in the correct format, as opposed to embedding that code within the same JSP that displays the form fields. Input Processors contain logic that is specific to the Web application, and are therefore loaded by the Web application's container.

For details about the Webflow and Pipeline technology and its editors, see the section Setting up Portal Navigation in the *WebLogic Portal Development Guide*.

That covers the key definitions for the Webflow technology. Now let's focus on the My Avitek page. On a page that preceded the display of the My Avitek page, you performed one of the following actions:

- Clicked the Login as Rachel Adams button graphic on the sample's Introduction page. The form used on the page constructed the URL value from predefined values and login credentials. (It was implemented this way to simplify the sample application, but is not a recommended practice.) Please see the WebLogic Portal documentation for information about using login authentication code, plus techniques to gather demographic information for Web applications.
- Clicked the My Avitek tab from a Products page or Shopping Cart page.

See the next section for information about events on the My Avitek page.

Events on the My Avitek Page

Every time a customer clicks a link or button on a JSP, it is considered an event. Events trigger particular responses in the default Webflow that allow customers to continue. While this response might be to load another JSP, it is usually the case that an Input Processor and/or Pipeline is invoked first.

When you are on a Products or Shopping Cart page and then click the My Avitek tab, you will notice a resulting URL similar to the following (shown here on several lines to improve readability).

```
http://<host>/<port>/b2cPortal/application?
origin=hnav_bar.jsp&event=bea.portal.framework.internal.refresh
&pageid=My+Avitek
```

The refresh event causes any page to be displayed again with the latest data. All the portlets on the My Avitek page that contain dynamic data (My Shopping Cart, My Order History, Tour Guide) have their data refreshed.

The tabs are provided via the `hnav_bar.jsp` file, which resides in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\f
ramework
```

The `hnav_bar.jsp` file imports two JSP Tag libraries:

```
<%@ taglib uri="webflow.tld" prefix="wf" %>
<%@ taglib uri="portal.tld" prefix="ptl" %>
```

When you clicked the My Avitek tab from another page in the application, the `hnav_bar.jsp` used the JSP tag in the following link for the target tab:

```
<a href="<ptl:createPortalPageChangeURL pageName='<%=
portalPageName %>'/>"><%=portalPageName%></a>
```

The (portal) `ptl:createPortalPageChangeURL` JSP tag generates a webflow URL for a page change event.

Dynamic Data Display on My Avitek Page

On the My Avitek page, let's look at one of the dynamic portlets, `summarycart` (shown as "My Shopping Cart" on the My Avitek page).

The webflow namespace file for the `summarycart` portlet is:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\app
lication-sync\webapps\b2cPortal\summarycartportlet.wf
```

It contains:

```
<presentation-origin node-name="step1" node-type="jsp">
    <node-processor-info page-name="step1.jsp"
         page-relative-path="/portlets/summarycart"/>
</presentation-origin>
```

The `step1.jsp` file is located in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\p
ortlets\summarycart\step1.jsp
```

The `step1.jsp` file uses a combination of JSP tags, including webflow tags to get the current shopping cart and saved shopping cart from the pipeline session:

```
<webflow:getProperty id="shoppingCart"
 property="<%=PipelineSessionConstants.SHOPPING_CART%>
 "type="com.beasys.commerce.ebusiness.shoppingcart.ShoppingCart"
 scope="session" namespace="portal" />

<webflow:getProperty id="savedShoppingCart"
 property="<%=PipelineSessionConstants.SAVED_SHOPPING_CART%>"
 type="com.beasys.commerce.ebusiness.shoppingcart.ShoppingCart"
 scope="session" namespace="portal" />
```

Administration Task to Designate Default Portal Page

Most of the properties discussed in this My Avitek section were not edited by hand, but were defined using the E-Business Control Center, and the Portal Management area of the WebLogic Portal Administration Tools. For example, designating that the My Avitek page was the default starting page in the b2cPortal application was accomplished using the Portal Administration Tools. This section explains how to start the tool and shows a sample screen for the default page setting.

1. In a new browser window, point to the following URL, which starts the WebLogic Portal Administration Tools:

   `http://<host>:<port>/e2eAppTools/index.jsp`

   These steps assume that the server for the e2eDomain is still running. You can run the `e2eAppTools` Web application and the sample application at the same

time. Both Web applications are part of the same `e2eApp` enterprise application and run in the same domain.

2. Substitute the name of your local machine (`localhost`) or the remote host name, and substitute the port number on which WebLogic Server is listening. For WebLogic Portal applications, the default port number is 7501. The `e2eAppTools` portion of the URL is case sensitive.

3. You are prompted for a username and password. These values may be different for your site, but by default they are:

   ```
   Username: administrator
   Password: password
   ```

   The login values are case sensitive. If these credentials do not work, please check with the administrator, who may have changed the default values for the administrator account after the sample was installed.

4. On the main Administration Tools page, click the icon in the Portal Management banner.

5. On the Portal Management Home page, click "Default Portal (Everyone)" for the b2cportal.

6. On the Group Portal Management Home page, click "Manage Pages and Portlets".

7. On the Pages and Portlets page, click "Select and Order Pages."

   The following figure shows a portion of the page:

Note how the My Avitek page has been designated as the default Home Page.

For more information about the WebLogic Portal Administration Tools, see the *WebLogic Portal Administration Guide*.

To exit the Administration Tools, close the browser window.

Next Step

To continue the tour, select the Products tab in the banner.

The Products Page

Having clicked the Products tab, this page loaded the top-level categories from the product catalog in the center portlet, `catalog.jsp`. Text and images for the catalog were organized earlier using the Catalog Management area of the WebLogic Portal Administration Tools. For a logged-in user, the latest data for My Shopping Cart and My Order History were refreshed.

The following screen shows the categories on the initial Products page.

Technical Details for the Products Page

This section provides details about the processing that occurs on this page.

Introduction

The Products page includes many of the portlets you saw on the My Avitek page:

- The **mybanner** portlet, which provides the top banner and the three tabs for My Avitek, Products, and Shopping Cart pages.
- The **login** portlet, used when an authenticated user has been logged into the application. It provides the Logout button and the name of the currently logged-in user.
- The **search** portlet, which provides a keyword-based search input box.
- The **summarycart** portlet, which provides an abbreviated list of the logged-in user's Shopping Cart.
- The **orderhistory** portlet, which provides an abbreviated list of the logged-in user's order history.
- The **tourguide** portlet, which provides the context-sensitive documentation in the running sample.

In addition, this Products page adds the `catalog.jsp` portlet. Its source files reside in the following subdirectory under your installed `BEA_HOME` directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\portlets\catalog
```

The `catalog.jsp` portlet includes JSP code from different include files, depending on the catalog event that just occurred, as you will see as you select different items in the catalog on subsequent pages.

Besides the `catalog.jsp` portlet, an additional portlet that you will see on subsequent Products pages is called the Product Evaluator. This portlet will appear on catalog pages that include one or more product items in a category. The Product Evaluator portlet uses a Web service created earlier in WebLogic Workshop to look-up consumer ratings for the selected product item. The portlet's source files (`evaluator.jsp`, and included files `step1.jsp` and `step2.jsp`) are located in the following subdirectory under your installed `BEA_HOME` directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\p
ortlets\evaluator
```

All of the portlets that comprise the Products page are identified in the following file:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\app
lication-sync\webapps\b2cPortal\b2cPortal.portal
```

For example:

```
    <page-name>Products</page-name
     ...
        <portlet-pool>
        <portlet-name>login</portlet-name>
        <portlet-name>search</portlet-name>
        <portlet-name>evaluator</portlet-name>
        <portlet-name>summarycart</portlet-name>
              <portlet-name>tourguide</portlet-name>
        <portlet-name>catalog</portlet-name>
        <portlet-name>subnav</portlet-name>
        <portlet-name>orderhistory</portlet-name>
   </portlet-pool>
```

Location in Default Webflow

The Webflow for this application displayed this catalog page when you selected the Products tab.

As mentioned in the previous section, the `catalog.jsp` portlet includes JSP code from different included files, depending on the catalog event that just occurred. For example, the following `webflow:getProperty` tag in `catalog.jsp` gets the last catalog event from the pipeline session:

```
<webflow:getProperty
    id="event"
    type="java.lang.String"
    property="<%= B2CPortalConstants.LAST_CATALOG_EVENT_ATTRIB %>"
    scope="session"
    namespace="portal" />
```

- If the catalog event is null, the following JSP is included into the `catalog.jsp` portlet:

  ```
  <jsp:include page="/portlets/catalog/catalog_index.jsp"
  flush="true" />
  ```

  `Catalog_index.jsp` performs the processing to load the top-level categories that you saw on the initial Products page.

- If the user clicked one of the specific categories, the following JSP is included into the `catalog.jsp` portlet:

  ```
  <jsp:include page="/portlets/catalog/category.jsp" flush="true"
  />
  ```

  Information about the `category.jsp` processing is provided in the Technical Details section of a category page.

- If the user clicked one product item in a specific category (not shown from the initial Products page), the following JSP is included into the `catalog.jsp` portlet:

  ```
  <jsp:include page="/portlets/catalog/item.jsp" flush="true" />
  ```

  For information about the item.jsp processing, see the section "The Product Item Portlet" on page 2-29.

- If the user entered a search keyword and clicked the Go button, the following JSP is included into the `catalog.jsp` portlet:

  ```
  <jsp:include page="/portlets/catalog/search_results.jsp"
  flush="true" />
  ```

  Information about the `search_results.jsp` processing is provided in the section "The Search Results Portlet" on page 2-43.

Dynamic Data Display

On the initial Products page that loaded the six product categories, the catalog event was null. Consequently the `catalog_index.jsp` file was included into the `catalog.jsp` portlet. If you want, take a look at the code for `catalog_index.jsp`, which is located in the following subdirectory under your installed `BEA_HOME` directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\p
ortlets\catalog
```

The `catalog_index.jsp` file includes a view iterator that cycles through the catalog and retrieves properties for each category. First we use a `webflow:getProperty` JSP tag to get the catalog categories view iterator from the pipeline session:

```
<webflow:getProperty
    id="categories"
    type="com.beasys.commerce.ebusiness.catalog.ViewIterator"
    property="<%=PipelineSessionConstants.CATALOG_CATEGORIES%>"
```

```
    scope="session"
    namespace="portal" />
```

Then we iterate through the categories and display each one. A subset of the coding is shown here:

```
<catalog:iterateViewIterator
    id="category"
    returnType="com.beasys.commerce.ebusiness.catalog.Category"
    iterator="<%=categories%>">

<catalog:getProperty
    id="categoryKey"
    returnType="com.beasys.commerce.ebusiness.catalog.CategoryKey"
    object="<%= category %>"
    propertyName="key" />

<catalog:getProperty
    id="largeImage"
    returnType="com.beasys.commerce.ebusiness.catalog.ImageInfo"
    object="<%= category %>"
    propertyName="image"
getterArgument="<%= new Integer(Category.LARGE_IMAGE_INDEX ) %>" />
...
```

Still in the same iterator in `catalog_index.jsp`, each table cell displays some of the properties on the page. A subset of the coding is shown here:

```
<a href="<portlet:createWebflowURL namespace="catalogportlet"
event="link.category"extraParams="<%=linkParams +
java.net.URLEncoder.encode( categoryKey.getIdentifier())
%>"/>"><img src="<webflow:createResourceURL
resource='<%=largeImage.getUrl()%>'/>" width="180" height="136"
alt="<%=shortDesc%>" border="0"></a>
```

That processing results in the display of the categories on the initial Product page, in the `catalog.jsp` portlet. You will notice in the HTML above that if the user then clicks the category's image, it will generate an event called `link.category` for the transition to the next page. But what event occurred that preceded the display of this initial Products page? See the next section for details.

Events on the Initial Products Page

When you are on a My Avitek or Shopping Cart page and then click the Products tab, you will notice a resulting URL similar to the following (shown here on several lines to improve readability).

```
http://<host>/<port>/b2cPortal/application?
origin=hnav_bar.jsp&event=bea.portal.framework.internal.refresh
&pageid=Products
```

The refresh event causes any page to be displayed again with the latest data. All the portlets on the Products page that contain dynamic data have their data refreshed. This particular refreshed Products page contains the initial one showing the six top-level categories.

The tabs are provided via the `hnav_bar.jsp` file, which resides in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\framework
```

The `hnav_bar.jsp` file imports two JSP Tag libraries:

```
<%@ taglib uri="webflow.tld" prefix="wf" %>
<%@ taglib uri="portal.tld" prefix="ptl" %>
```

When you clicked the Products tab from another page in the application, the `hnav_bar.jsp` used the following JSP tag in the link for the target tab:

```
<a href="<ptl:createPortalPageChangeURL pageName='<%=
portalPageName %>'/>"><%=portalPageName%></a>
```

The (portal) `ptl:createPortalPageChangeURL` JSP tag generates a webflow URL for a page change event.

Also, as mentioned previously, on this initial Products page the catalog event was null. This resulted in loading the `catalog_index.jsp` into the `catalog.jsp` portlet. From the coding in the `catalog.jsp` portlet:

```
<webflow:getProperty
    id="event"
    type="java.lang.String"
    property="<%= B2CPortalConstants.LAST_CATALOG_EVENT_ATTRIB %>"
    scope="session"
    namespace="portal" />
<%
    if ( event == null )
    {

%>

<jsp:include page="/portlets/catalog/catalog_index.jsp"
flush="true" />
```

Category Administration Tasks

The catalog in this sample application uses a simple, two-level hierarchy of categories. The first level is the default root category, and the second level is comprised of the six categories you saw on the initial Products page. However, the WebLogic Portal catalog feature supports multi-level hierarchical categories, as explained in the WebLogic Portal documentation.

The six categories in this sample application were previously defined in the Catalog Management area of the browser-based WebLogic Portal Administration Tools. If you want, you can follow the instructions in this section to see the existing definitions.

The following steps assume that the server for the `e2eDomain` is still running. You can run the `e2eAppTools` Web application and the sample application at the same time. Both Web applications are part of the same `e2eApp` enterprise application and run in the same domain.

1. In a new browser window, use the following URL format, which starts the WebLogic Portal Administration Tools:

   ```
   http://<host>:<port>/e2eAppTools/index.jsp
   ```

   In the URL, substitute for the name of your local machine (`localhost`) or the remote host name, and substitute the port number on which WebLogic Server is listening. For WebLogic Portal applications, the default port number is 7501. The `e2eAppTools` portion of the URL is case sensitive. For example:

   ```
   http://localhost:7501/e2eAppTools/index.jsp
   ```

2. You are prompted for a username and password. These values may be different for your site, but by default they are:

   ```
   Username: administrator
   Password: password
   ```

   The login values are case sensitive. If these credentials do not work, please check with the administrator, who may have changed these default values for the administrator account after the sample was installed.

3. On the main Administration Tools page, click the icon in the Catalog Management banner.

4. On the Catalog Management page, click the underlined word **Categories**.

The following screen is displayed:

Click one of the underlined categories. For example, click `AviPix (0avipix)`. (Warning: do not click the red X icon, which deletes the category from the catalog. Keep in mind that you are using the administration pages for the sample catalog data.)

5. On the resulting page, scroll down to see the location for an image representing that category.

For more information about catalog administration tasks, see the WebLogic Portal documentation.

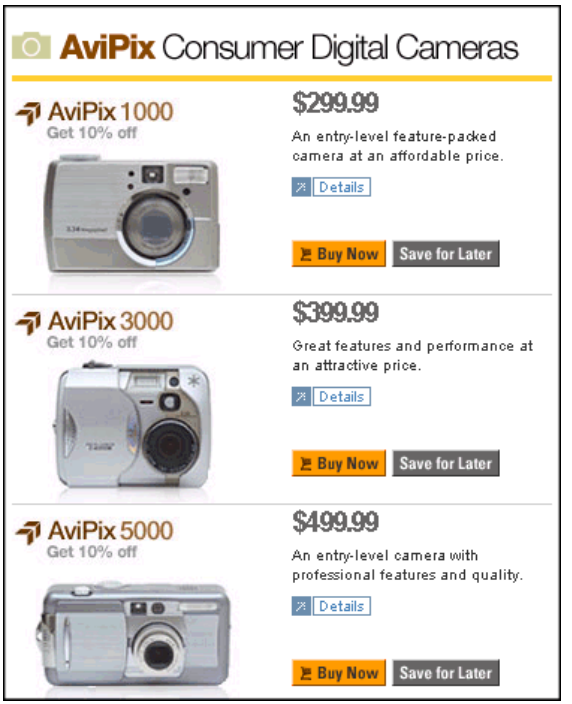
To exit the Administration Tools, close the browser window.

Next Step

To continue the tour, click the **AviPix** Consumer Digital Cameras category.

The Category Portlet

Product items in this specific category have been loaded from the catalog and displayed on this page. For example:



Category Page Technical Details

This section provides details about the processing that occurs on this page.

Introduction

Product items in this specific category have been loaded from the catalog and displayed on this page. The top banner includes a graphic that highlights the current category. The Product Evaluator portlet near the bottom of the page includes a menu that allows logged-in user Rachel Adams to get ratings on product items in this category.

Processing of the Webflow event determined that a link.category event occurred, which resulted in loading the `category.jsp` file into the centered `catalog.jsp` portlet.

Notice a sample URL that resulted for the current sample application's page, shown over several lines to improve formatting:

```
http://localhost:7501/b2cPortal/application
?origin=catalog_index.jsp
&event=bea.portal.framework.internal.portlet.event
&pageid=Products&portletid=catalog
&wfevent=link.category&wlcs_catalog_category_id=0avipix
```

In the sample URL, the origin state is shown in the second, third, and fourth lines. The Webflow event is shown in the last line. A link.category event occurred, which resulted in the page you saw in the sample application. Product items are displayed for (in this example) the `0avipix` category that was selected on the prior page. The number zero is an index into the six product categories.

Location in Default Webflow

The Webflow for this application resulted in loading the data for the category selected on a prior page. The `catalog.jsp` portlet includes JSP code from different included files, depending on the catalog event that just occurred. For example, the following `webflow:getProperty` tag in `catalog.jsp` gets the last catalog event from the pipeline session:

```
<webflow:getProperty
    id="event"
    type="java.lang.String"
```

```
    property="<%= B2CPortalConstants.LAST_CATALOG_EVENT_ATTRIB %>"
    scope="session"
    namespace="portal" />
```

In the `catalog.jsp` portlet, the returned event attribute is checked:

```
...
<%
    }
    else if ( ( event.equals( "link.category" ) ) ||
              ( event.equals( "button.category.buy" ) ) ||
              ( event.equals( "button.category.save") )   )
    {
%>
...
<jsp:include page="/portlets/catalog/category.jsp" flush="true" />
```

In this case, the `category.jsp` file was included into the `catalog.jsp` portlet because the `link.category` event occurred.

Dynamic Data Display

In the `category.jsp` file, we get the category that was selected on the prior page from the pipeline session:

```
<webflow:getProperty
    id="category"
    type="com.beasys.commerce.ebusiness.catalog.Category"
    property="<%= PipelineSessionConstants.CATALOG_CATEGORY %>"
    scope="session"
    namespace="portal" />
```

We then use a series of JSP tags to get properties about the current category. An example:

```
<catalog:getProperty
    id="headerImage"
    returnType="com.beasys.commerce.ebusiness.catalog.ImageInfo"
    object="<%= category %>"
    propertyName="image"
    getterArgument="<%= new Integer( CatalogItem.SMALL_IMAGE_INDEX
    ) %>" />
```

Next we use a view iterator to get the products items in this category from the pipeline session:

```
<webflow:getProperty
    id="items"
    type="com.beasys.commerce.ebusiness.catalog.ViewIterator"
    property="<%= PipelineSessionConstants.CATALOG_ITEMS %>"
    scope="session"
    namespace="portal" />
```

We then iterate through the product items and display some properties about each one in the `catalog.jsp` portlet. A subset of the coding is shown here:

```
<catalog:iterateViewIterator
    id="item"
    returnType="com.beasys.commerce.ebusiness.catalog.ProductItem"
    iterator="<%= items %>">

<catalog:getProperty
    id="itemImage"
    returnType="com.beasys.commerce.ebusiness.catalog.ImageInfo"
    object="<%= item %>"
    propertyName="image"
    getterArgument="<%= new Integer( CatalogItem.SMALL_IMAGE_INDEX
    ) %>" />

<catalog:getProperty
    id="currentPrice"
    returnType="com.beasys.commerce.axiom.units.Money"
    object="<%= item %>"
    propertyName="currentPrice" />
          ...
```

Because `category.jsp` includes a number of buttons (Details, Buy Now, and Save for Later), additional work is done. For example, to prepare the Details link, we set up the HTTP request parameters.

```
<%
    String linkParams =
        HttpRequestConstants.CATALOG_CATEGORY_ID + "=" +
            java.net.URLEncoder.encode( categoryKey.getIdentifier()
              ) + "&" +
        HttpRequestConstants.CATALOG_ITEM_SKU + "=" +
            itemKey.getIdentifier();
%>
```

Still in the same iterator in `category.jsp`, each table cell displays properties on the page. A small subset of the coding is shown here. Please see the table in `category.jsp` for additional coding:

```
<a href="<portlet:createWebflowURL namespace="catalogportlet" event="link.item"
extraParams="<%=linkParams%>" />"><img src="<webflow:createResourceURL
resource="<%=itemImage.getUrl()%>"/>" width="187" height="139" alt=""
border="0"></a>
```

The code shown above is used if the user clicks the item's image or the Details button graphic.

You will notice in the HTML subset above that if the user clicks the product item's image, it will generate an event called link.item for the transition to the next page. But what event occurred that preceded the display of this initial Products page? See the next section for details.

Events

On a prior Products page, after the `catalog_index.jsp` file was included into the `catalog.jsp` portlet, an HTML table in the portlet included the following code:

```
<a href="<portlet:createWebflowURL namespace="catalogportlet"
event="link.category"
extraParams="<%=linkParams + java.net.URLEncoder.encode(
categoryKey.getIdentifier() )%>"
/>"><img src="<webflow:createResourceURL resource='<%=largeImage.getUrl()%>'/>"
width="180" height="136" alt="<%=shortDesc%>" border="0"></a>
```

When you clicked on a specific category, either in the banner graphic or in the `catalog.jsp` (which was displaying the `catalog_index.jsp` file), it triggered the `link.category` Webflow event. That resulted in loading the `category.jsp` file into the `catalog.jsp` portlet, which in turn displayed the product items that comprise the selected category.

All the catalog events are defined in the following Webflow file for the `catalog.jsp` portlet:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\app
lication-sync\webapps\b2cPortal\catalogportlet.wf
```

One of the events defined in `catalogportlet.wf` is:

```
<event event-name="link.category">
   <destination namespace="catalogportlet"
       node-name="getCategoryIP" node-type="inputprocessor"/>
```

```
  </event>
                ...
  <processor-origin node-name="getCategoryIP"
     node-type="inputprocessor"><node-processor-info
     class-name="examples.e2e.b2c.catalog.webflow.GetCategoryIP"/>
     <event-list>
         <event event-name="success">
              <destination namespace="catalogportlet"
                 node-name="getCategory" node-type="pipeline"/>
         </event>
     </event-list>
  </processor-origin>
...
```

The `link.category` event uses an input processor named `getCategoryIP`, which is a Java program that takes a Category ID string from the HTTP request, validates the ID String, creates a `CategoryKey` based on the ID String, and adds it to the `PipelineSession` in the session scope. You can view the source file for this input processor in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\WEB-INF\src\examples\e2e\b2c\catalog\webflow\GetCategoryIP.java
```

Assigning Items to a Category

For the sample application, product items were assigned to categories using the Catalog Management section of the browser-based WebLogic Portal Administration Tools. This section outlines the basic steps. For details, see the WebLogic Portal documentation.

Note: If you have a catalog with hundreds of categories and thousands or tens of thousands of items, making the assignments via the Administration Tools is obviously not practical. An alternative is to use the DBLoader tool provided by WebLogic Portal.

The following steps assume that the server for the `e2eDomain` is still running. You can run the `e2eAppTools` Web application and the sample application at the same time. Both Web applications are part of the same `e2eApp` enterprise application and run in the same domain.

1. In a new browser window, use the following URL format, which starts the WebLogic Portal Administration Tools:

   ```
   http://<host>:<port>/e2eAppTools/index.jsp
   ```

In the URL, substitute for the name of your local machine (localhost) or the remote host name, and substitute the port number on which WebLogic Server is listening. For WebLogic Portal applications, the default port number is 7501. The "e2eAppTools" portion of the URL is case sensitive. For example:

```
http://localhost:7501/e2eAppTools/index.jsp
```

2. You are prompted for a username and password. These values may be different for your site, but by default they are:

   ```
   Username: administrator
   Password: password
   ```

   The login values are case sensitive. If these credentials do not work, please check with the administrator, who may have changed these default values for the administrator account after the sample was installed.

3. On the main Administration Tools page, click the icon in the Catalog Management banner.

4. On the Catalog Management page, click the underlined word **`Categories`**.

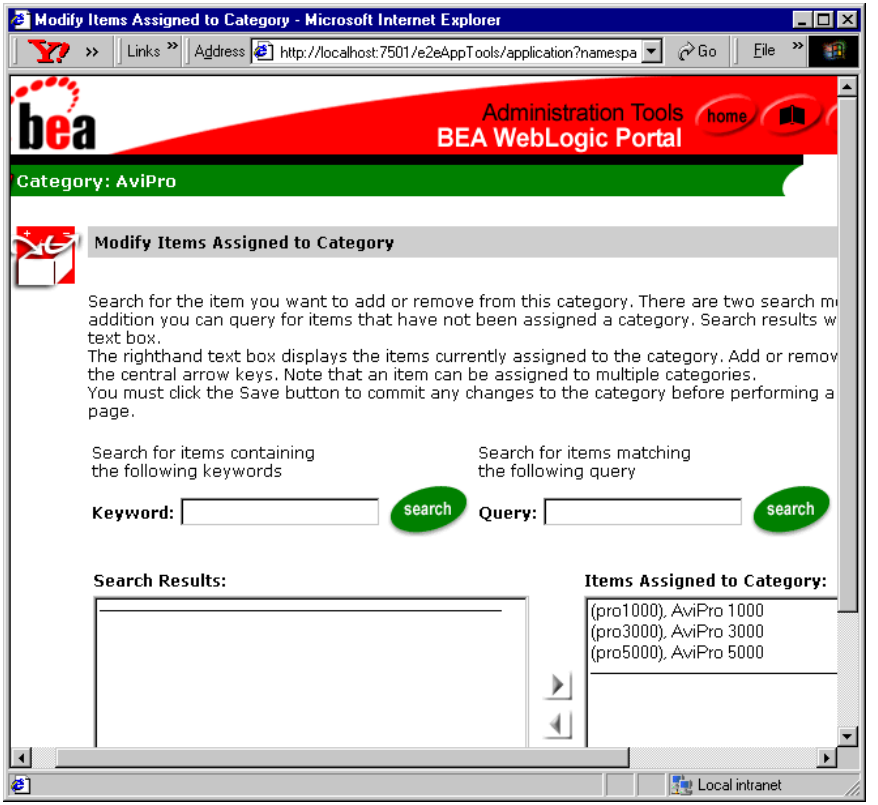
5. In the Catalog hierarchy display, click the category or subcategory into which you want to add or remove an item. (The categories in this sample application do not have subcategories.)

6. When the category is shown in the hierarchy, click its underlined link. For example, assume that you clicked the `AviPro` category. Near the top of the page, notice the text:

```
Editing Category : AviPro
  Enter the appropriate information then click Save.
  To modify the items assigned to this category, please click here.
```

   On the page, click the link in the text "...please click here."

   WebLogic Portal displays a screen similar to the following:



The Items Assigned to Category text box shows the items that are already in this category. You can search for the item you want to add or remove via three modes: keyword, query-based, or orphaned-items (uncategorized items). The search results are displayed on the left-side text box. To add an item to the category, move the item to the right-side text box by clicking on the right arrow.

For more details, see the WebLogic Portal documentation.

To exit the Administration Tools, close the browser window.

Next Step

To continue the tour, click the image of the **AviPix 5000** product item in the category.

The Product Item Portlet

Details for the product item are loaded from the catalog. Earlier in the process, the design team worked with an administrator to identify each item's larger image in the catalog, plus the longer description. The Catalog Management area of the WebLogic Portal Administration Tool was used to specify the details for the item. The following screen shows a sample resulting display in the portlet:

Technical Details for the Product Item Portlet

This section provides details about the processing that occurs on this page.

Introduction

Processing of the Webflow event determined that a **`link.item`** event occurred, which resulted in loading the `category.jsp` file into the centered `catalog.jsp` portlet.

Notice a sample URL that resulted for the current sample application's page, shown over several lines to improve formatting. The URL on your sample page will be different if you selected another product item:

```
http://blues:7501/b2cPortal/application
?origin=category.jsp
&event=bea.portal.framework.internal.portlet.event
&pageid=Products&portletid=catalog
&wfevent=link.item
&wlcs_catalog_category_id=0avipix&wlcs_catalog_item_sku=pix5000
```

In the sample URL, the origin state is shown in the second, third, and fourth lines. The Webflow event is shown in the last line. A `link.item` event occurred, which resulted in the page you saw in the sample application. Product items are displayed for (in this example) the `0avipix` category that was selected on the prior page. The number zero is an index into the six product categories. And in this example, the specific item selected was the AviPix 5000. Its Stock Keeping Unit (SKU) is `pix5000`.

Location in Default Webflow

The Webflow for this application resulted in loading the data for the specific product item that was selected on a prior page. The `catalog.jsp` portlet includes JSP code from different included files, depending on the catalog event that just occurred. For example, the following `webflow:getProperty` tag in `catalog.jsp` gets the last catalog event from the pipeline session:

```
<webflow:getProperty
    id="event"
    type="java.lang.String"
    property="<%= B2CPortalConstants.LAST_CATALOG_EVENT_ATTRIB %>"
    scope="session"
    namespace="portal" />
```

In the `catalog.jsp` portlet, the returned event attribute is checked:

```
...
<%
    }
    else if ( ( event.equals( "link.item" ) ) ||
              ( event.equals( "button.item.buy" ) ) ||
```

```
                ( event.equals( "button.item.save") )    )
        {
%>
...
<jsp:include page="/portlets/catalog/item.jsp" flush="true" />
```

In this case, the `item.jsp` file was included into the `catalog.jsp` portlet because the `link.item` event occurred.

Dynamic Data Display

In the `item.jsp` file, we get the item's category from the pipeline session. We then use a series of JSP tags to get properties about the current category and its current item. The following example shows two of the JSP tags:

```
<catalog:getProperty
  id="itemImage"
  returnType="com.beasys.commerce.ebusiness.catalog.ImageInfo"
  object="<%= item %>"
  propertyName="image"
  getterArgument="<%= new Integer( CatalogItem.LARGE_IMAGE_INDEX
  %>" />

<catalog:getProperty
  id="currentPrice"
  returnType="com.beasys.commerce.axiom.units.Money"
  object="<%= item %>"
  propertyName="currentPrice" />
```

Because `item.jsp` includes a number of buttons (Buy Now, Save for Later), additional work is done. For example, to prepare the Buy Now link, we set up the HTTP request parameters for the graphic:

```
<catalog:getProperty
  id="itemKey"
  returnType="com.beasys.commerce.ebusiness.catalog.ProductItemKey"
  object="<%= item %>"
  propertyName="key" />

<%
  String linkParams =
     HttpRequestConstants.CATALOG_ITEM_SKU + "=" + itemKey.getIdentifier();
%>
```

The following code from item.jsp shows how the URL will be constructed for the user who clicks the Buy Now button graphic.

```
<a href="<portlet:createWebflowURL namespace="catalogportlet"
   event="button.item.buy"
extraParams="<%= linkParams %>" />"><img src="<webflow:createResourceURL
resource='<%=imagesPath + "prod_step2_buynow.gif"%>'/>" width="63" height="18"
   alt=""
border="0"></a>
```

Also in the `item.jsp` file, as we are preparing to display the item's price, we use an Internationalization (I18N) JSP tag to get the currency type that is defined for the catalog:

```
<i18n:getMessage bundleName="/commerce/currency"
messageName="<%=currentPrice.getCurrency()
%>"/><%=WebflowJSPHelper.priceFormat(
currentPrice.getValue() )%>
```

Events

On a prior Products page, when you clicked on a specific product item, it triggered the `link.item` Webflow event. That resulted in loading the `item.jsp` file into the `catalog.jsp` portlet.

All the catalog events are defined in the following Webflow file for the `catalog.jsp` portlet:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project
\application-sync\webapps\b2cPortal\catalogportlet.wf
```

One of the events defined in `catalogportlet.wf` is:

```
<event event-name="link.item">
       <destination namespace="catalogportlet"
              node-name="getItemCategoryIP"
                     node-type="inputprocessor"/>
</event>
...
```

The `link.item` event uses an input processor named `getItemCategoryIP`, which is a Java program that takes a SKU string from the HTTP request, validates the SKU string, creates a `ProductItemKey` based on the SKU String, and adds it to the `PipelineSession` in the session scope. You can view the source file for this input processor in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\
WEB-INF\src\examples\e2e\b2c\catalog\webflow\GetItemCategoryIP.ja
va
```

Adding or Modifying Product Item Data

In a similar fashion to the way category data is added to the catalog, specific item data is added using the WebLogic Portal Administration Tools. If you have not read about this administration task, please see the section "Assigning Items to a Category" on page 2-26.

Next Step

To continue the tour, scroll down on the Products page and select the **AviPix 5000** item from the Product Evaluator portlet. Click the Go button and view the ratings returned by the Web service and displayed in the portlet. Then read the next section.

The Product Evaluator Portlet and Web Service

After you selected a product item in the Product Evaluator portlet and then clicked the Go button, the portlet was refreshed with the results of a product rating Web service. (In the sample, you may need to scroll down to see the results.)

During the application design and development cycle, we used BEA WebLogic Workshop to create a product rating Web service. We then used the Portlet Wizard to generate the interfaces for this Web services.

Finally, we included the interfaces code into the Product Evaluator portlet and then completed the presentation coding based on earlier design prototype work with a graphic artist. We also packaged the Web service as a Web application within the e2eApp, as explained in Chapter 4, "Web Services Tour."

Technical Details for the Product Evaluator Portlet

The Product Evaluator portlet, `evaluator.jsp`, appears near the bottom of Products pages. Initially the portlet displays a discount ad via an included `step1.jsp` file. When the Products page is refreshed and the Product Evaluator portlet determines that specific product items have been loaded, it includes a `step2.jsp` file that allows the logged-in user to get product ratings. This portlet is interesting because it uses a Web service that we created earlier with WebLogic Workshop to look-up the product ratings.

WebLogic Workshop is a visual development environment that makes it easy for application developers and J2EE experts alike to build and deploy enterprise-class web services. The product is comprised of two major components:

1. A design-time tool that lets developers write Java code to implement Web services.
2. A run-time framework that provides the Web services infrastructure, testing, debugging, and deployment environment for applications.

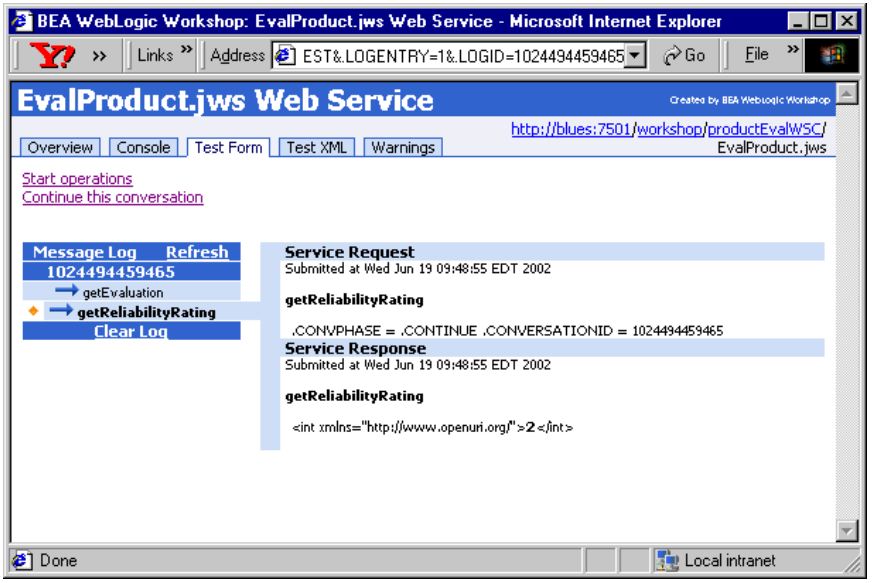
The meeting place between the design-time tool and the run-time framework is the Java Web Service (JWS) file and any associated control (CTRL) files. JWS files are standard Java files with annotations (using the Javadoc syntax) to express additional functionality. Annotations are used to display the Web service and its properties graphically. And the annotations are used by the framework to generate the EJB and J2EE code to execute the Web service. Control files typically include a collection of method definitions that allow you to easily access a resource such as a database or another Web service.

Another key file is the Web Service Description Language (WSDL) file that describes the Web service. WSDL files describe all the methods a Web service exposes (in the form of XML messages it can accept and send), as well as the protocols over which the Web service is available. The WSDL file provides all the information a client application needs to use the Web service.

In development, we used a browser-based test form provided by WebLogic Workshop to check whether the expected results were being returned by the Web service. To set this up, we deployed a **`workshop`** webapp as part of the **`e2eApp`** enterprise application. The workshop webapp contains the `productEvalWSC` and `paymentWS` Web services that we created with WebLogic Workshop. Consequently you can run the WebLogic Workshop **test pages** for these Web services in the server instance for the e2eDomain. The test pages are browser-based. For example, in a new browser window, open:

```
http://localhost:7501/workshop/productEvalWSC/EvalProduct.jws
```

The URL is case sensitive. For a complete walk-through of the Product Evaluator Web service's test form screens, please refer to Chapter 4, "Web Services Tour." The following sample screen shows only one of the test forms in mid-conversation with the Web service. On a prior test form screen, we specified **`pix1000`** as the `productId` (SKU) on which to get evaluation data. In the following screen, we just ran a test of the `getReliabilityRating` method:



Notice the Web service response is the integer **2**. On the Product Evaluator portlet, this results in displaying two out of five possible stars. For example:

After you create a Web service with WebLogic Workshop, you can use the Portlet Wizard that comes with WebLogic Workshop, point to the WSDL file, and generate the client interfaces. For a description of this process, see the Chapter 4, "Web Services Tour," which is also available from the sample application's Introduction page.

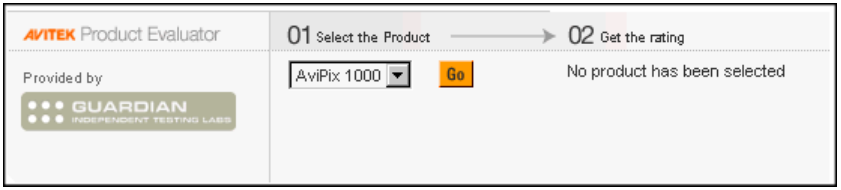
Determine Items to Display

The Product Evaluator portlet's JSP file is:

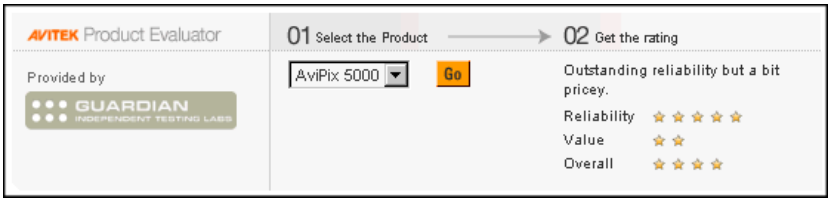
```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\portlets\evaluator\evaluator.jsp
```

It initially includes the `step1.jsp` file, which displays a discount ad:

When an event from another portlet on the Products page allows the Product Evaluator portlet to determine that specific product item data is present, the Product Evaluator portlet includes the `step2.jsp` file that can get product ratings. The first view of the `step2.jsp` portlet contains one or more items on the pull-down menu. For example:



After you select an item from the portlet's menu and click the Go button, the `step2.jsp` file in the portlet is refreshed with the results returned from the Web service. For example, if you selected the AviPix 5000 camera:



The first action taken by the `evaluator.jsp` portlet is get the last catalog event from the pipeline session:

```
<webflow:getProperty
  id="event"
  type="java.lang.String"
  property="<%= B2CPortalConstants.LAST_CATALOG_EVENT_ATTRIB %>"
```

```
  scope="session"
  namespace="portal" />
```

If the catalog event is null, it displays the static discount ad:

```
<jsp:include page="/portlets/evaluator/step1.jsp" flush="true" />
```

That jsp:include tag brings in the graphic you saw near the bottom of the initial Products page:

If the catalog event is not null, we determine which event type occurred. If the event was `link.category`, `button.category.buy`, or `button.category.save`, we set the `evalEvent` to "`browse`". If the event was `link.item`, `button.item.buy`, or `button.item.save`, we set the `evalEvent` to "`detail`". Finally, if the event was `button.search`, `button.search.buy`, or `button.search.save`, we set `evalEvent` to "`search`". The value is significant to the portlet because it determines which item or items to list on the product rating pull-down menu, and which rating data to get from the Web service.

We use a `webflow:setProperty` JSP tag to pass the `evalEvent` value to the pipeline:

```
<webflow:setProperty
    property="B2CPortalConstants.PRODUCT_EVAL_ATTRIB"
        scope="request"
               value="<%=evalEvent%>"
               namespace="portal"/>
```

Using the Web Service

`Evaluator.jsp` then includes the `step2.jsp` portlet. If the Product Evaluator portlet is posting to itself, it will display the results of the `productEvalIP` input processor with the select list focused on the requested product item. Otherwise, it will display an unselected list of product items and no rating information (yet).

The step2.jsp file contains the Web services includes:

```
<%@ page import="org.openuri.www.* "%>
<%@ page import="org.openuri.www.x2002.x04.soap.conversation.* "%>
<%@ page import="java.io.IOException" %>
<%@ page import="java.rmi.RemoteException" %>
<%@ page import="productEvalWSC.EvalProduct_Impl" %>
<%@ page import="productEvalWSC.EvalProductSoap" %>
```

We get the `evalEvent` value from the pipeline session:

```
<webflow:getProperty
    id="evalEvent"
        type="java.lang.String"
        property="B2CPortalConstants.PRODUCT_EVAL_ATTRIB"
        scope="request"
        namespace="portal" />
```

In `step2.jsp`, we then use a view iterator to get the items for a category or a specific item. If the user selects an item and clicks the Go button, we invoke the EvalProduct.jws Web service that we defined in WebLogic Workshop. We also used the Portlet Wizard to create the Web services interface code, which includes referencing the Web service's WSDL.

```
<%
  if (request.getParameter("origin").equals("step2.jsp") &&
      request.getParameter("wfevent").equals("button.evaluator.go"))
  {
        String productEvaluation = request.getParameter("productEvaluation");
%>
<!-- Portlet Wizard generated web services interfaces code -->

<%
        EvalProduct_Impl         m_Proxy = null;
        EvalProductSoap     m_ProxySoap = null;
        String m_conversationID = session.getId();
        String comments = "";
        int valueRating = 0;
        int reliabilityRating = 0;
        int overallRating = 0;
        boolean webServiceAvail = true;

        String serverNamePort = "http://" + request.getServerName() + ":"
           + request.getServerPort() + "/";

        try
        {
          m_Proxy = new EvalProduct_Impl(serverNamePort +
                    "workshop/productEvalWSC/EvalProduct.jws?WSDL");
        }
        catch (IOException ex)
        {
          ex.printStackTrace();
          webServiceAvail = false;
        }
          m_ProxySoap = m_Proxy.getEvalProductSoap();

        try
        {
          // Set up the header objects we'll need
```

```
        StartHeader startHeader =
              new StartHeader( m_conversationID,
                   serverNamePort
                   + "workshop/productEvalWSC/EvalProduct.jws");
                   ContinueHeader continueHeader =
                   new ContinueHeader( m_conversationID );
        // Start the conversation
        GetEvaluation getEvaluation = new GetEvaluation(productEvaluation);

        if (m_ProxySoap.getEvaluation(getEvaluation,
              startHeader).getGetEvaluationResult().equals("SUCCESS"))
        {
        // Continue the conversation
        comments =  m_ProxySoap.getComments(null,
                   continueHeader).getGetCommentsResult();

        // Continue the conversation
           reliabilityRating = m_ProxySoap.getReliabilityRating(null,
               continueHeader).getGetReliabilityRatingResult();

        // Continue the conversation
           overallRating = m_ProxySoap.getOverallRating(null,
               continueHeader).getGetOverallRatingResult();

        // Continue the conversation
           valueRating = m_ProxySoap.getValueRating(null,
               continueHeader).getGetValueRatingResult();
}
        ...
```

See the `step2.jsp` code for additional coding, including the error handling. After receiving the product rating from the Web service, we then display the results in the portlet as part of the `step2.jsp` work.

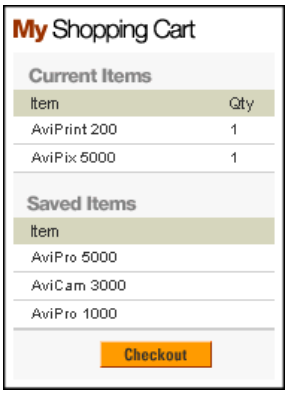
For more information about how we used WebLogic Workshop and then Portlet Wizard during the development of the Product Evaluator portlet, see the Web Services Technical Tour that is available from this sample's Introduction page.

Next Step

To continue the tour, click the **Buy Now button** that appears next to the AviPix 5000 product item.

The Buy Now Button and Inventory Checks Via WebLogic Integration AI

The Buy Now button triggered a Webflow event that added the item to the Current Items list in the `summarycart` portlet ("My Shopping Cart") on the refreshed page. If you click Buy Now multiple times for the same product item, the quantity column is updated. The full Shopping Cart, with price and discount information, is not seen until you click the Checkout button or the Shopping Cart tab. In the following sample screen, the AviPix 5000 camera was added to a cart that already included the AviPrint 200 camera. Your cart's Current Items and Saved Items lists in the sample application may be different.



Technical Details for the Buy Now Button

The Buy Now button can be selected for a specific item when the user is browsing a category page, or an item's detail page, or a search results page. The event type, depending on the page, could be one of the following, as defined in the `catalogportlet.wf` Webflow file:

- `button.category.buy`
- `button.item.buy`

- button.search.buy

When the Buy Now button is selected, the sample application performs an inventory check to make sure the order can be fulfilled before adding the item to the user's shopping cart. The database includes an inventory table that keeps data about the current, minimum, and maximum inventory for a product item. The inventory table is accessed in read-only mode via the Application Integration (AI) component of WebLogic Integration.

For example, this check occurs when you click the Buy Now button on several portlets in this b2cPortal's Products page. An inventory check is also performed if user tries to update the quantity of the item already in the shopping cart, by entering a new value on the Shopping Cart's `step1.jsp` portlet, and then clicking the RECALCULATE button.

Before we discuss the inventory check's implementation, let's look at the way several portlet pages set up the Buy Now link. For example, from the item.jsp file, which is included into the `catalog.jsp` portlet:

```
<a href="<portlet:createWebflowURL namespace="catalogportlet"
   event="button.item.buy"
extraParams="<%= linkParams %>" />"><img src="<webflow:createResourceURL
resource='<%=imagesPath + "prod_step2_buynow.gif"%>'/>" width="63" height="18"
   alt=""
border="0"></a>
```

In the HTML subset above, if the user clicks the Buy Now image, it will generate an event called button.item.buy for the transition to the next page. The `linkParams` parameters contain the information about the specific item. This data was collected earlier in the `item.jsp` processing by a series of `<catalog:getProperty...>` JSP tags, followed by:

```
<%
    String linkParams =
        HttpRequestConstants.CATALOG_ITEM_SKU + "=" + itemKey.getIdentifier();
%>
```

Inventory Checks

The inventory check is implemented in the `CheckInventoryPC` pipeline component. You can find its source file in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\
examples\e2e\b2c\shoppingcart\pipeline\CheckInventoryPC.java
```

This pipeline component checks the Inventory and places an additional value. In this sample application, it is invoked every time the shopping cart content is changed. In your production environment, you may want to do it differently for performance reason, based on your business logic. For example, you may want to check inventory only when the item is placed in the shopping cart the first time, by keeping the actual quantity in stock, and showing different messages based on the difference between what is in stock and what is in the shopping cart.

The `CheckInventoryPC` pipeline component works with an `InventoryProvider` SPI. Its source files are in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\examples\e2e\common\inventory\spi\*.java
```

This SPI is a stateless session bean that includes the `checkInventory` remote method. This method:

- Uses Application Integration (AI) to make a call via a pre-configured AI service. This service is used to query the inventory table.
- Calls an XML Helper to parse the response from AI.
- Returns available inventory (`int`) based on the response.

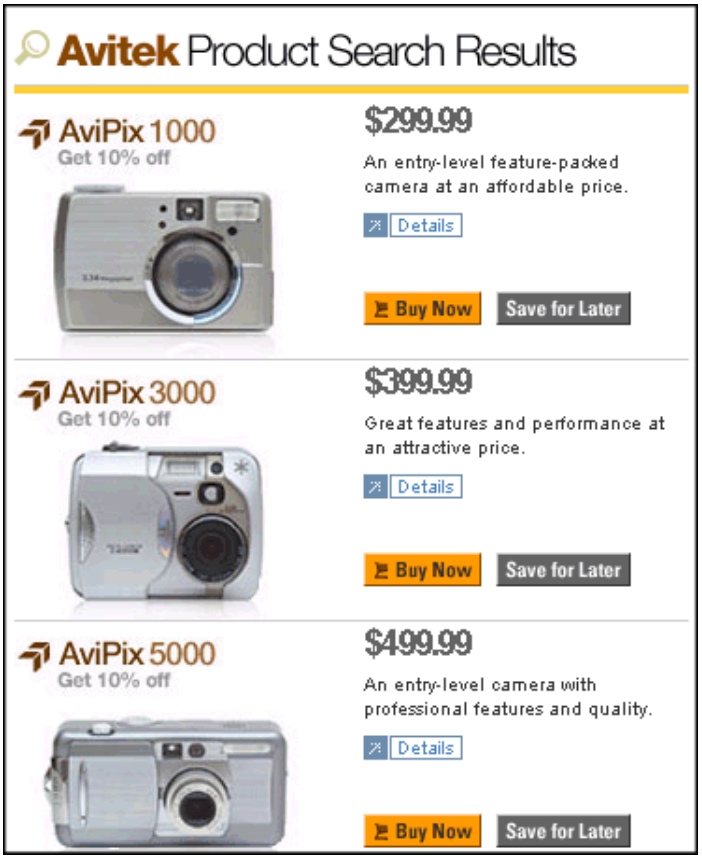
Next Steps

To continue the tour, please browse through the product catalog and add about four or five product items to your cart's Current Items list and Saved Items list. Having a lot of items in the cart will make it easier to try a number of options on the Shopping Cart page.

As part of that exercise, use the Search portlet. For example, enter the keyword **camera** and click the Go button. On the search results page, add some of the items to the Current Items list and Saved Items list. Then read the following explanation about the Search Results portlet.

The Search Results Portlet

The search feature provided with the product catalog is based on keywords that an administrator assigned to product items. The Catalog Management area of the WebLogic Portal Administration Tools was used to make the assignments. The development team and administrator can work together to determine the best keywords for each item. For information about the existing keywords, plus information about the Webflow events associated with the search Go button, and other events, read this section. The following screen shows a portion of the search results portlet's display after entering "camera" and clicking Go:



Technical Details for the Search Results Portlet

This section provides details about the processing that occurs on this page.

Introduction

The `search_results.jsp` portlet presents information about product items that resulted from the keyword-based search. The customer can then browse through the results. Assembling the search results was accomplished using a combination of portlet, webflow, catalog, and i18n (internationalization) JSP tags.

For a match to occur, the customer must have entered a keyword that had already been associated with one or more product items. The assignment of keywords for product items was done by an administrator in the Catalog Management area of the Weblogic Portal Administration Tools. Information about keywords and the Administration Tools appears later in this discussion.

Note: WebLogic Portal commerce services allow you to use query-based searches, in addition to keyword searches. To simplify the scope of this sample application, only a keyword-based search has been implemented here. For details about query-based searches, see the WebLogic Portal documentation.

By default, the `search_results.jsp` portlet file resides in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\portlets\catalog
```

The `searchform.jsp` portlet, which provides the search input box, the Go button's graphic, and processing, resides by default in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\portlets\search
```

As you run this sample application, it is important to understand that as a developer you are primarily interested in the JSP code before it was rendered by the browser. That's why the code fragments in this section, and the View Code link above, describe and show you the pre-rendered JSP file for a particular portlet.

Location in the Default Webflow

Customers see the `search_results.jsp` portlet on the Products page after they enter a keyword in the `searchform.jsp` portlet and click the Go button. From the `search_results.jsp` portlet, customers can:

- View more details about a particular item shown in the results list (which loads the items.jsp portlet, which includes content from the itemdetails.jsp portlet)
- Add a product item shown in the search results list to their shopping cart by clicking the Buy Now or Save for Later buttons.

WebLogic Platform provides the E-Business Control Center (EBCC), a graphical tool that simplifies complex tasks such as rule definition, Webflow editing, and portal creation and management. The EBCC Webflow Editor and Pipeline Editor are designed to help you create, modify, and validate Webflow and Pipeline XML configuration files. If you want, you an start the EBCC and examine the existing Webflow and Pipeline definitions for the `e2eApps`.

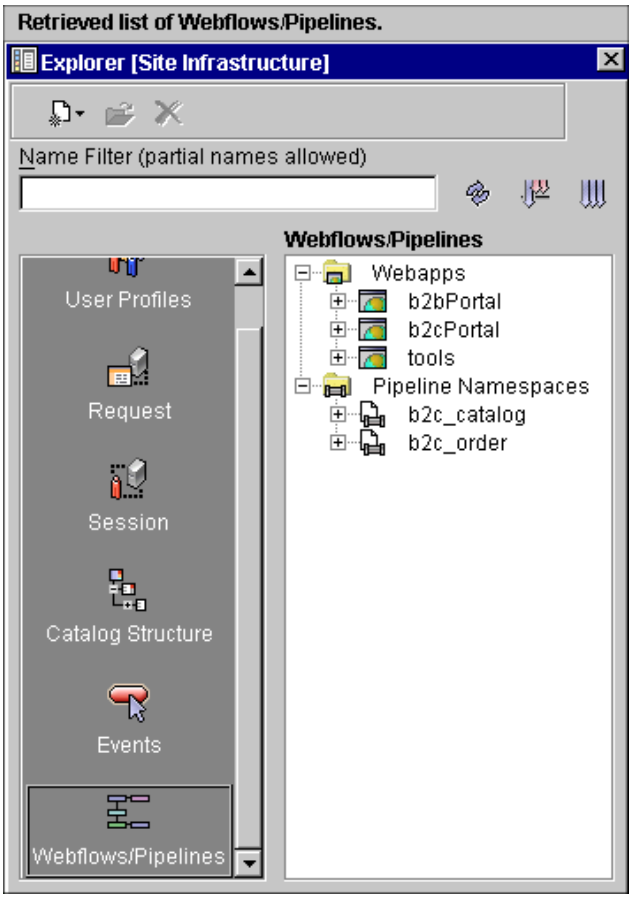
For example, complete the following steps:

1. On a Windows system, go to the Start menu and select Programs →BEA WebLogic Platform 7.0 →WebLogic Portal 7.0 →E-Business Control Center.
2. Click the Open Project icon on the upper left side of the screen. For example:

3. In the Open Project dialog window, navigate to the following directory:

   `weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project`

4. Select and open the `e2eApp-project.eaprj` project file that was installed there.
5. In the EBCC Explorer window, select the Site Infrastructure tab near the bottom of the window. Then click the Webflows/Pipelines icon — you may need to scroll down to see it. The EBCC displays a screen similar to the following:



6. Click the plus sign next to `b2cPortal` and then double-click the `searchportlet` Webflow item. The Webflow editor displays a large graphical representation of the `searchportlet` Webflow components. You can close the adjacent Explorer window if you want more screen space to view the components. The Webflow Editor and Pipeline Editor contain many options, not described here. However, you can read a detailed description of the editors in the section Setting up Portal Navigation in the *WebLogic Portal Development Guide*.

Events

On a page that preceded the display of the search results, the user entered a keyword in the search input box and then clicked the Go button. This took place in the searchform.jsp portlet. (Another Webflow scenario is that you returned to this Products page by clicking the Product tab from another page, and the Products page was refreshed with the latest catalog event, which happened to be search results. In either event, this explanation will focus on the original search-related event.) The form defined in the portlet includes the following:

A table cell containing the search input box.

```
<input name="<%=HttpRequestConstants.CATALOG_SEARCH_STRING%>"
type="text" class="searchPortlet" size="27">
```

A scriptlet that defines additional Webflow parameters to display the results on the Products page (because the search can be submitted from other portal pages) and executes the search portlet's Webflow:

```
<%
String formParams=
  PortalAppflowConstants.PORTLET_WEBFLOW_EVENT_PARAMETER + "=" + "button.search"
    + "&" +
  PortalAppflowConstants.PAGE_PARAMETER + "=" +
  B2CPortalConstants.PRODUCTS_PAGE_NAME + "&" +
  PortalAppflowConstants.PORTLET_PARAMETER + "=" + "search" ;
%>
```

And the form definition itself:

```
<form method="POST" action="<webflow:createWebflowURL
event="bea.portal.framework.internal.portlet.event" origin="searchform.jsp"
extraParams="<%= formParams %>"/>" onsubmit="return ValidateSearchForm(this)" >
```

The event **`button.search`** is defined separately as using `keywordSearchIP` (an input processor) in the `searchportlet` namespace.

This definition is in the searchportlet.wf (Webflow) file that resides in:

```
BEA_HOME\weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-pr
oject\application-sync\webapps\b2cPortal
```

`KeywordSearchIP.java` takes a keyword search String from the HTTP request, validates the search String, creates a `KeywordQuery` based on the search String, adds it to the `PipelineSession` in the session scope, and clears the `PipelineSession` of

any previous search results. If a keyword search `String` is not supplied, the `PipelineSession` will be examined for previous cached search results. If previous results exist and are valid, the results are left in the `PipelineSession`.

You can view the source file for this input processor in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\WEB-INF\src\examples\e2e\b2c\catalog\webflow\KeywordSearchIP.jav
a
```

Once the search results are available, the `search_results.jsp` portlet page is displayed. See the next section for more on the dynamic data display.

Now let's look at events defined within the `search_results.jsp`, for subsequent processing:

- The **`link.item`** event is triggered when the customer either clicks the product item's image (for example, an image of an AviPix 3000 camera) or the item's Details button image.
- The **`button.search.buy`** event is triggered when the customer clicks the Buy Now button next to a product item in the search results.
- The **`button.search.save`** event is triggered when the customer clicks the Save for Later button next to a product item in the search results.

The following line from `search_results.jsp` (pre-rendering by the browser) shows how the coding was set up for the Details button on the portlet:

```
<a href="<portlet:createWebflowURL namespace="catalogportlet" event="link.item"
extraParams="<%=linkParams%
<" />"><img src="<webflow:createResourceURL resource='<%=imagesPath +
"prod_step2_details.gif"%>'/>
```

In the example above, the `href` link is constructed using a `webflow:createResourceURL` JSP tag. Notice that the Webflow namespace is `catalogportlet`. Let's look at the `catalogportlet.wf` (Webflow) file in:

```
BEA_HOME\weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-pr
oject\application-sync\webapps\b2cPortal
```

In `catalogportlet.wf`, the event named `link.item` identifies the following input processor: `storeCatalogEventIP`. This input processor is used to store the current portlet event as a session-scoped pipeline attribute. The portlet uses this information to decide which view of the catalog to display.

For example, if on the search results portlet you clicked the Details button next to an AviPix 1000 camera, notice how the value for the `webflow:createResourceURL` parameter results in the following. (The URL shown here spans several lines to improve the formatting.)

```
http://localhost:7501/b2cPortal/application?origin=search_results
.jsp
&event=bea.portal.framework.internal.portlet.event
&pageid=Products&portletid=catalog
&wfevent=link.item&wlcs_catalog_item_sku=pix1000
```

Dynamic Data Display

In `search_results.jsp`, the matched results (if any) will be retrieved from the pipeline session. For example:

```
<webflow:getProperty
  id="searchResults"
  type="com.beasys.commerce.ebusiness.catalog.ViewIterator"
  property="<%= PipelineSessionConstants.CATALOG_SEARCH_RESULTS%>"
  scope="session"
  namespace="portal" />
```

We check to see if there are search results:

```
<%
  if ( ( searchResults != null ) && ( searchResults.size() > 0 ) )
  {
%>
```

If there are search results, we iterate through the catalog, retrieving information about each product item that matched the keyword. For example:

```
<catalog:iterateViewIterator
    id="item"
    returnType="com.beasys.commerce.ebusiness.catalog.ProductItem"
    iterator="<%= searchResults %>">

    <catalog:getProperty
      id="itemImage"
      returnType="com.beasys.commerce.ebusiness.catalog.ImageInfo"
      object="<%= item %>"
      propertyName="image"
      getterArgument="<%= new Integer(CatalogItem.SMALL_IMAGE_INDEX
        ) %>" />
```

Other data from the catalog is similarly retrieved for each product item: `currentPrice`, (short) `description`, and `productItemKey` (SKU).

We then set up the HTTP request parameters for this item's detail link.

Data for each matched product item is cycled through the view iterator until there are no more items. We also reset the search results view iterator in case the portlet is refreshed.

If there are no matches found for the keyword entered on searchform.jsp, we use in search_results.jsp the `i18n:getMessage` JSP tag provided by WebLogic Portal. I18N is an abbreviation for Internationalization, meaning the process of setting up application code so that language-specific files can be added or modified, to customize the message text for non-English customers. For example, in `search_results.jsp`, we have:

```
<i18n:getMessage messageName="noResults" bundleName="search_results" />
```

The `search_results.properties` file, which is in the same directory as `search_results.jsp`, contains:

```
noResults=No matches found.
```

Administration Tasks for Keyword-based Searches

The search function for this sample application uses keywords that were previously defined in the Catalog Management area of the WebLogic Portal Administration Tools. This section explains how to set up keywords for items.

The following steps assume that the server for the `e2eDomain` is still running. You can run the `e2eAppTools` Web application and the sample application at the same time. Both Web applications are part of the same `e2eApp` enterprise application and run in the same domain.

1. In a new browser window, use the following URL format, which starts the WebLogic Portal Administration Tools:

   ```
   http://<host>:<port>/e2eAppTools/index.jsp
   ```

   In the URL, substitute for the name of your local machine (`localhost`) or the remote host name, and substitute the port number on which WebLogic Server is listening. For WebLogic Portal applications, the default port number is 7501. The `e2eAppTools` portion of the URL is case sensitive. For example:

   ```
   http://localhost:7501/e2eAppTools/index.jsp
   ```

2. You are prompted for a username and password. These values may be different for your site, but by default they are:

   ```
   Username: administrator
   Password: password
   ```

   The login values are case sensitive. If these credentials do not work, please check with the administrator, who may have changed these default values for the administrator account after the sample was installed.

3. On the main Administration Tools page, click the icon in the Catalog Management banner.

4. On the Catalog Management page, click the underlined word Items.

5. In the Keywords input box, enter the following keyword: **camera**

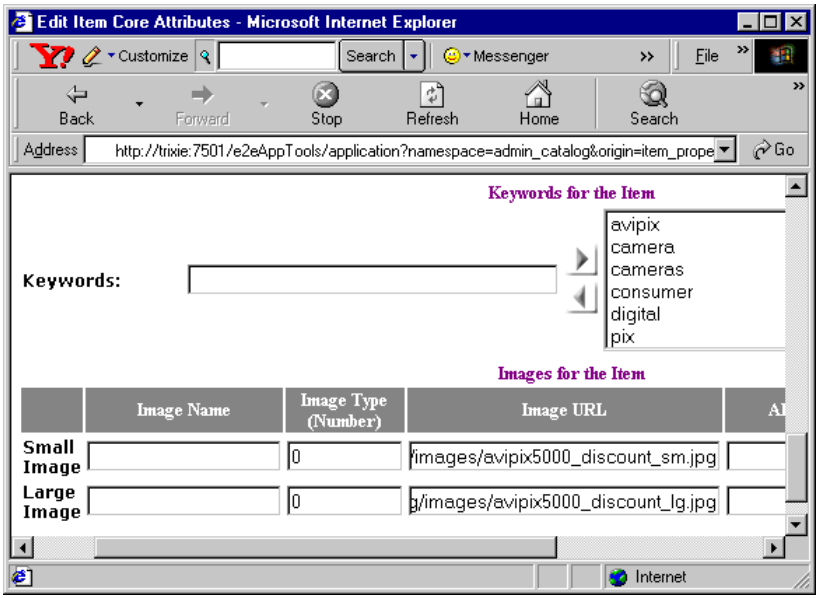
6. Click one of the underlined search results. For example, click AviPix 5000. (Warning: do not click the **red X icon**, which deletes the item from the catalog. Keep in mind that you are using the administration pages for the catalog data.)

7. On the resulting page, click the Edit icon that is on the same line with "Item Core Attributes."

Some of the values on the Edit Item Information page will be familiar if you have browsed through the sample application's catalog. For example, notice the short description and long descriptions values. Also notice how the small and large graphics' location was specified (near the bottom of the page). Initially the data records were loaded from scripts or using the DBloader program provided with WebLogic Portal. These catalog administration pages provide a view into the values and the ability to subsequently change values, if desired. (Most organizations will run catalog data scripts again, instead of using the Administration Tools to modify values.)

The search keywords can be viewed or modified on this Edit Item Information page. In the sample application, we defined a set of keywords for each product item in the catalog. Scroll down the page to find the Keywords input box and the set of existing keywords.

The following graphic shows a portion of the page:



Existing keywords are: avipix, camera, cameras, consumer, digital, pix.

The administrator can work with the development team to determine additional keywords to help site visitors find product items.

For more information, see the WebLogic Portal documentation.

To exit the Administration Tools, close the browser window.

Next Steps

If your search returned “No matches found” enter one of the existing keywords such as camera in the search input box. Again, the existing keywords are avipix, camera, cameras, consumer, digital, pix. Then click the Go button.

To continue the tour, add several items to your cart from among the search results. Click the Save for Later button for an item as the last event before moving on to the next section.

The Save for Later Button

The Save for Later button triggered a Webflow event that added the item to the Saved Items list in the `summarycart` portlet (“My Shopping Cart”) on the refreshed page.

Technical Details for the Save for Later Button Event

The Save for Later button can be selected for a specific item when the user is browsing a category page, or an item’s detail page, or a search results page. The event type, depending on the page, could be one of the following, as defined in the `catalogportlet.wf` Webflow file:

- `button.category.save`
- `button.item.save`
- `button.search.save`

When the Save for Later button is selected, the product item is moved into the user's Saved Items list. Several portlet pages set up the Save for Later link in the following way. This example is from the `item.jsp` file, which is included into the `catalog.jsp` portlet:

```
<a href="<a href="<portlet:createWebflowURL namespace="catalogportlet"
event="button.item.save" extraParams="<%= linkParams %>" />">
<img src="<webflow:createResourceURL resource='<%=imagesPath
+ "prod_step2_savelater.gif"%>'/>" width="72" height="18" alt="" border="0"></a>
```

In the HTML subset above, if the user clicks the Save for Later image, it will generate an event called button.item.save for the transition to the next page. The `linkParams` parameters contain the information about the specific item. This data was collected earlier in the item.jsp processing by a series of `<catalog:getProperty...>` JSP tags, followed by:

```
<%
    String linkParams =
        HttpRequestConstants.CATALOG_ITEM_SKU + "=" + itemKey.getIdentifier();
%>
```

Note that an inventory check is not done when you click Save for Later on a Products page. However, an inventory check is done on a shopping cart page if you click Add to Cart for an item that is currently in the Saved List.

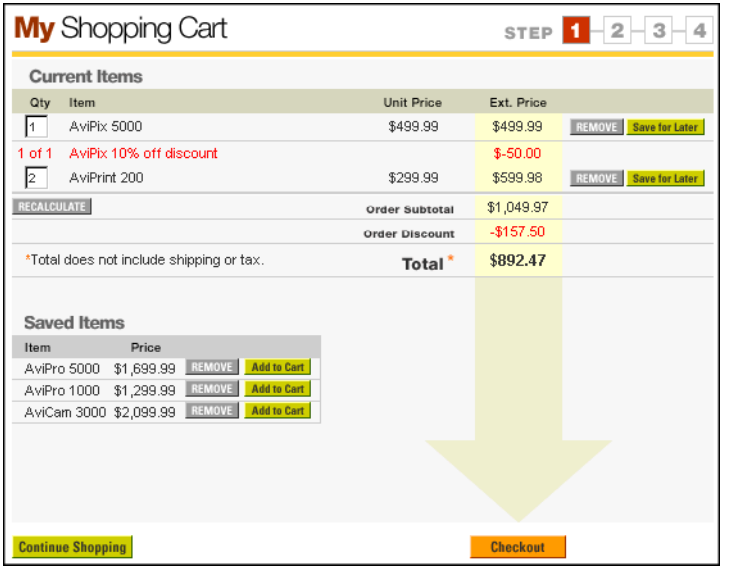
Next Step

To continue the tour, click the Checkout button to proceed to the My Shopping Cart `step1.jsp` portlet.

The My Shopping Cart Portlet, Step1.jsp

Having clicked the Checkout button in the `summarycart` portlet on a Products page, the portal application's Webflow resulted in the display of the `step1.jsp` checkout portlet on this Shopping Cart page. A graphic, `check_step1_header.gif`, was loaded to help customers understand where they are in the order fulfillment process. If there are items in the cart, price and discount information for the order are shown. Shown are prices for the unit, a 10% unit discount (for AviPix Consumer Cameras

only), a 15% total order discount for orders over $100, and a net total. If there are no items in the cart, the string "You do not have any item in your shopping cart" is displayed.



Technical Details for the Step1.jsp Portlet

This section provides details about the processing that occurs on this page.

Introduction

The shopping cart pages present a series of portlets that allow the customer to checkout and complete their order. At any time during the checkout process, an inventory check is done if the user's shopping cart changes; this inventory check uses a `CheckInventoryPC` pipeline component that works with a `InventoryProvider` SPI, which in turn uses the Application Integration (AI) component of WebLogic Integration to ensure that the order can be fulfilled.

On subsequent shopping cart pages, when the order is submitted, a pipeline component calls a Payment Web service to authorize the credit card purchase. We created the Web service in WebLogic Workshop.

After the confirmed order is persisted to the database, another pipeline component converts the order to an XML representation, and places it on a Java Message Service (JMS) queue. The WebLogic Integration Business Process Management (BPM) event processor dequeues the order and processes it.

The shopping cart or "`checkout`" portlets are in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\portlets\checkout\*
```

- **`step1.jsp`**, the current portlet, provides the detailed My Shopping Cart data, presenting the current and saved items. It also computes and displays 10% line-item discounts for any AviPix Consumer Camera category items, and a 15% total order discount for orders over $100. Information about the `step1.jsp` processing is contained in this page's Technical Details section.

- **`step2.jsp`** provides the Checkout data, presenting pre-set billing and shipping information for logged-in user Rachel Adams. Information about the `step2.jsp` processing is contained in the section "The Checkout Portlet, Step2.jsp" on page 2-67.

  Note: To keep the scope of this sample application simple, we did not include portlets that would allow the user to enter or modify their credit card or shipping information. However, WebLogic Portal supports this type of processing, and provides a separate sample to demonstrate it. Please refer to the Commerce and Campaign Features Tour in the WebLogic Portal documentation.

- **`step3.jsp`** provides the (pre-submit) Order Submission page, showing a summary of the order about to be submitted. When the user clicks the Submit Order, credit card authorization is performed via a Payment Web service that we created in WebLogic Workshop. Information about the `step3.jsp` processing is contained in the section "The Order Submission Portlet, Step3.jsp" on page 2-70.

- **`step4.jsp`** provides the Order Confirmation page. The confirmed order is converted by a pipeline component to an XML representation. The pipeline component places this copy of the order on a Java Message Service (JMS) queue. The WebLogic Integration BPM event processor dequeues the order and processes it. Information about the `step4.jsp` processing is contained in the section "The Order Confirmation Portlet, Step4.jsp" on page 2-74.

Step1.jsp Processing, Including Inventory Checks and Discounts

The `step1.jsp` My Shopping Cart portlet uses JSP tags to get the shopping cart, saved shopping cart, and inventory counts from the pipeline session:

```
<webflow:getProperty id="shoppingCart"
property="<%=PipelineSessionConstants.SHOPPING_CART%>"
type="com.beasys.commerce.ebusiness.shoppingcart.ShoppingCart" scope="session"
namespace="portal" />
<webflow:getProperty id="savedShoppingCart"
property="<%=PipelineSessionConstants.SAVED_SHOPPING_CART%>"
type="com.beasys.commerce.ebusiness.shoppingcart.ShoppingCart" scope="session"
namespace="portal" />
<webflow:getProperty id="inventoryCount"
property="<%=B2CPortalConstants.INVENTORY_CHECK%>" type="int[]"
scope="session" namespace="portal" />
```

After iterating through all the shopping cart lines, we determine if a message is needed about low inventory. For example:

```
<% if ( (inventoryCount != null) && (inventoryCount[inventoryIndex]
<= shoppingCartLine.getQuantity() ) ) { outOfStock = true; %><%=
shoppingCartLine.getProductItem().getName() %> -
<i><i18n:getMessage bundleName="checkout"
messageName="over_inventory"/></i> - <% ;}
else { %><%= shoppingCartLine.getProductItem().getName() %> <% } %>
```

The inventory check is implemented in the `CheckInventoryPC` pipeline component. You can find its source file in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src
\examples\e2e\b2c\shoppingcart\pipeline\CheckInventoryPC.java
```

This pipeline component checks the Inventory and places an additional value. In this sample application, it is invoked every time the shopping cart content is changed. For example, the `step1.jsp` page includes a Quantity field with an associated Recalculate button. If the user changes the quantity in this portlet, the inventory is checked again.

Note: In your production environment, you may want to do it differently for performance reason, based on your business logic. For example, you may want to check inventory only when the item is placed in the shopping cart the first time, by keeping the actual quantity in stock, and showing different messages based on the difference between what is in stock and what is in the shopping cart.

The `CheckInventoryPC` pipeline component works with an `InventoryProvider` SPI. Its source files are in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\examples\e2e\common\inventory\spi\*.java
```

This SPI is a stateless session bean that includes the `checkInventory` remote method. This method:

- Uses the WebLogic Integration Application Integration (AI) component to make a call via a pre-configured AI service. This service is used to query the inventory table.
- Calls an XML Helper to parse the response from AI.
- Returns available inventory (`int`) based on the response.

In the inventory check, notice the `over_inventory` message name. To support internationalization (I18N) efforts, it is defined in a separate properties file, `checkout.properties`, as:

```
# If the inventory cannot fulfill the quantity requested
over_inventory=out of stock
```

The `step1.jsp` portlet also computes the amount of a line-item discount to offer. In this sample, we elected to give a 10% discount to any cameras in the AviPix Consumer Camera category. A total order discount is also computed for orders that exceed $100.

A number of events can occur on the `step1.jsp` page including:

- `button.updateShoppingCartQuantities`, if the user changes the quantity field and then clicks the RECALCULATE button.
- `button.deleteItemFromShoppingCart`, if the user clicks the DELETE button in the Current Items list.
- `button.moveItemToSavedList`, if the user clicks the Save for Later button in the Current Items list.
- `button.deleteItemFromSavedList`, if the user clicks the DELETE button in the Saved Items list.
- `button.moveItemToShoppingCart`, if the user clicks the Add to Cart button in the Saved Items list.
- `link.next`, if the user clicks the Checkout button.

Information about those events is presented in the technical details sections for those events after you click one of those buttons.

Note: If the user clicks the Continue Shopping button on the `step1.jsp` portlet, we use a JSP tag to take the user to the Products page:

```
<a href="<portal:createPortalPageChangeURL  pagename="Products"
/>">
<img src="<webflow:createResourceURL
resource='<%=imagesPath + "check_step1_continue.gif"%>'/>"
width="94" height="18" alt="" border="0"></a>
```

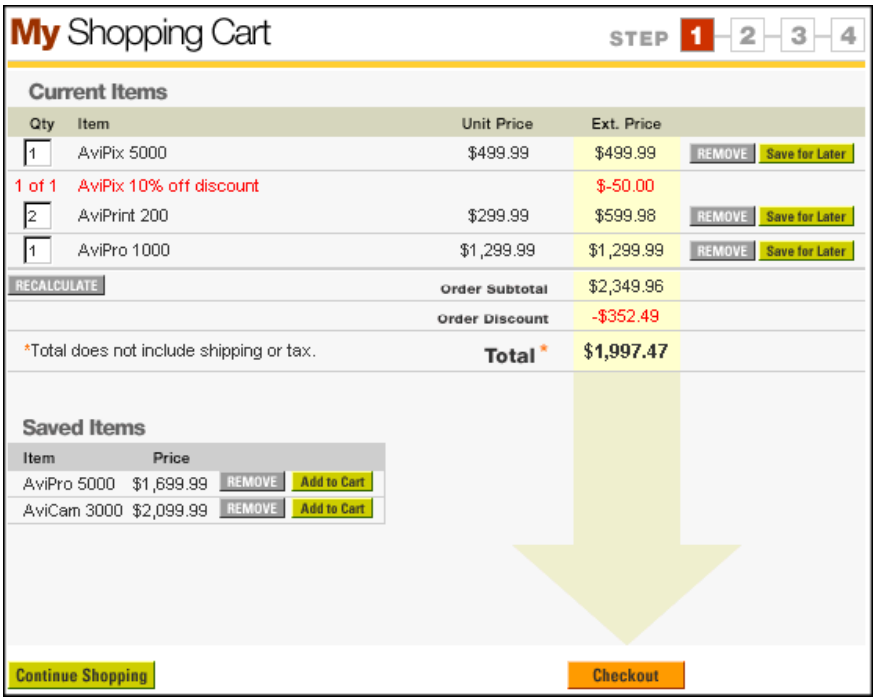
Next Step

To continue the tour, click the Add to Cart button next to an item on the Saved Items list.

The Add to Cart Button in Saved Items List

Having clicked the Add to Cart button in the Shopping Cart `step1.jsp` portlet's "Saved Items" list, the page was refreshed with updated data for the same portlet. The Current Items list now contains the product item you added to the cart. Prices are shown for the unit, a 10% unit discount for items over $100.00, a 15% total order discount for orders over $100, and a net total. The item you selected on the prior page is no longer in the Saved Items list.

The following sample screen may show items that are different from your cart.



Technical Details the Add to Cart Button in the Saved Items List

The `step1.jsp` My Shopping Cart page includes an event named `button.moveItemToShoppingCart`. It is triggered when the user clicks the Add to Cart button in the shopping cart's Saved Items list. On the refreshed page, the item is moved from the Saved Items list to the Current Items list.

You can find the step1.jsp portlet file in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\portlets\checkout\*
```

An example from `step1.jsp`:

```
<a href="<portlet:createWebflowURL namespace="checkoutportlet"
event="button.moveItemToShoppingCart"
extraParams="<%= extraParams %>" />">
<img src="<webflow:createResourceURL
```

```
resource='<%=imagesPath + "check_step1_add.gif"%>'/>"
width="52" height="13" alt="" border="0"></a>
```

The `extraParams` parameter contains data about the product item, collected already in `step1.jsp`:

```
<%
  extraParams = HttpRequestConstants.CATALOG_ITEM_SKU + "="
                shoppingCartLine.getProductItem().getKey().getIdentifier();
%>
```

The `button.moveItemToShoppingCart` event is defined in the `checkoutportlet.wf` Webflow file, which resides in the following directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project
\application-sync\webapps\b2cPortal
```

The event is:

```
<event event-name="button.moveItemToShoppingCart">
       <destination namespace="checkoutportlet"
       node-name="shoppingcart_MoveProductItemToShoppingCartIP"
       node-type="inputprocessor"/>
</event>
```

The pipeline component is `MoveProductItemToShoppingCartPC`. You can view its Java source file in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src
\examples\e2e\b2c\shoppingcart\pipeline
```

`MoveProductItemToShoppingCartPC` removes a `ProductItem` from a customer's list of saved items and adds it to their shopping cart. The customer's login id is specified by the `PipelineSessionConstants.USER_NAME` attribute in the Pipeline Session. The SKU of the item to move is specified by the `PipelineSessionConstants.CATALOG_ITEM_SKU` pipeline session attribute. The `PipelineSessionConstants.SAVED_SHOPPING_CART` and `PipelineSessionConstants.SHOPPING_CART` pipeline session attributes, and the `WLCS_SAVED_ITEM_LIST` table are updated to reflect the change.

Next Step

To continue the tour, click the REMOVE button next to an item on the Saved Items list.

The REMOVE Button in the Saved Items List

Having clicked the REMOVE button in the shopping cart's `step1.jsp` portlet's Saved Items list, the page was refreshed with updated data for the same portlet. The product item you selected has been removed from the Saved Items list.

Technical Details for REMOVE Button in the Saved Items List

The `step1.jsp` My Shopping Cart page includes an event named `button.deleteItemFromSavedList`. It is triggered when the user clicks the DELETE button in their shopping cart's Saved Items list. On the refreshed page, the item no longer exists on the Saved Items list.

You can find the `step1.jsp` portlet file in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\portlets\checkout\*
```

An example from `step1.jsp`:

```
<a href="<portlet:createWebflowURL namespace="checkoutportlet"
event="button.deleteItemFromSavedList"
extraParams="<%= extraParams %>" />">
<img src="<webflow:createResourceURL
resource='<%=imagesPath + "check_step1_remove.gif"%>'/>"
width="42" height="13" alt="" border="0"></a>
```

The `extraParams` parameter contains data about the product item, collected already in `step1.jsp`:

```
<%
   extraParams = HttpRequestConstants.CATALOG_ITEM_SKU + "=" +
                 shoppingCartLine.getProductItem().getKey().getIdentifier();
%>
```

The button.`deleteItemFromSavedList` event is defined in the `checkoutportlet.wf` Webflow file, which resides in the following directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\app
lication-sync\webapps\b2cPortal
```

The event is:

```
<event event-name="button.deleteItemFromSavedList">
     <destination namespace="checkoutportlet"
     node-name="shoppingcart_DeleteProductItemFromSavedListIP"
     node-type="inputprocessor"/>
</event>
```

The pipeline component is `DeleteProductItemFromSavedListPC`. You can view its Java source file in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\example
s\e2e\b2c\shoppingcart\pipeline
```

`DeleteProductItemFromSavedListPC` removes a `ProductItem` from a customer's list of saved items. The customer's login id is specified by the `PipelineSessionConstants.USER_NAME` attribute in the Pipeline Session. The SKU of the item to delete is specified by the `PipelineSessionConstants.CATALOG_ITEM_SKU` pipeline session attribute. The `PipelineSessionConstants.SAVED_SHOPPING_CART` and `PipelineSessionConstants.SHOPPING_CART` pipeline session attributes, and the `WLCS_SAVED_ITEM_LIST` table are updated to reflect the change.

Next Step

To continue the tour, click the REMOVE button next to an item on the cart's Current Items list.

The REMOVE Button on the Current Items List

Having clicked the REMOVE button in the shopping cart `step1.jsp` portlet's Current Items list, the page was refreshed with updated data for the same portlet. The product item and its prices have been removed from the cart.

Technical Details for REMOVE Button on the Current Items List

The `step1.jsp` My Shopping Cart page includes an event named button.`deleteItemFromShoppingCart`. It is triggered when the user clicks the DELETE button in their shopping cart's Current Items list. On the refreshed page, the item no longer exists on the Current Items list.

You can find the step1.jsp portlet file in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\portlets\checkout\*
```

An example from step1.jsp:

```
<a href="<portlet:createWebflowURL namespace="checkoutportlet"
event="button.deleteItemFromShoppingCart"
extraParams="<%= extraParams %>" />">
<img src="<webflow:createResourceURL
resource='<%=imagesPath + "check_step1_remove.gif"%>'/>"
width="42" height="13" alt="" border="0"></a>
```

The extraParams parameter contains data about the product item, collected already in step1.jsp:

```
<%
    extraParams = HttpRequestConstants.CATALOG_ITEM_SKU + "="+
                  shoppingCartLine.getProductItem().getKey().getIdentifier();
%>
```

The button.deleteItemFromShoppingCart event is defined in the checkoutportlet.wf Webflow file, which resides in the following directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project
\application-sync\webapps\b2cPortal
```

The event is:

```
<event event-name="button.deleteItemFromShoppingCart">
    <destination namespace="checkoutportlet"
    node-name="shoppingcart_DeleteProductItemFromShoppingCartIP"
    node-type="inputprocessor"/>
</event>
```

The input processor is DeleteProductItemFromShoppingCartIP. You can view its Java source file in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\W
EB-INF\src\examples\e2e\b2c\shoppingcart\webflow
```

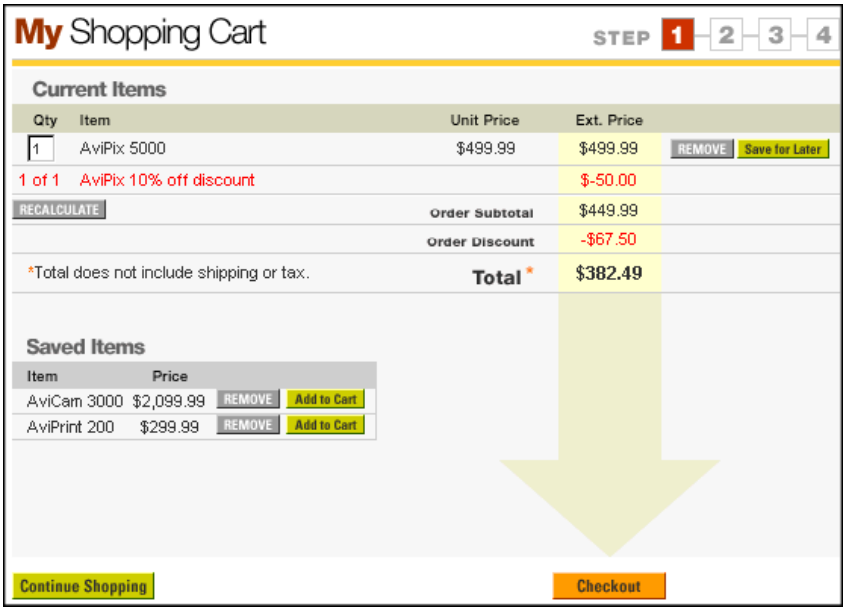
DeleteProductItemFromShoppingCartIP deletes a specified item from the shopping cart. The SKU of the item to be deleted is specified as the HttpRequestConstants.CATALOG_ITEM_SKU parameter in the request object. The shopping cart is retrieved from the PipelineSessionConstants.SHOPPING_CART attribute in the pipeline session. The item is deleted and the pipeline session is updated with the modified shopping cart.

Next Step

To continue the tour, click the Save for Later button next to an item in the cart's Current Items list.

The Save for Later Button on Current Items List

Having clicked the Save for Later button in the "Current Items" portion of the `step1.jsp` Shopping Cart portlet, the page was refreshed with updated data for the same portlet. The item's status was changed and was moved to the Saved Items list on the cart.



Technical Details

The `step1.jsp` My Shopping Cart page includes an event named `button.moveItemToSavedList`. It is triggered when the user clicks the Save for Later button in the shopping cart's Current Items list. On the refreshed page, the item is moved from the Current Items list to the Saved Items list.

You can find the step1.jsp portlet file in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\portlets\checkout\*
```

An example from step1.jsp:

```
<a href="<portlet:createWebflowURL namespace="checkoutportlet"
event="button.moveItemToSavedList" extraParams="<%= extraParams %>" />">
<img src="<webflow:createResourceURL
resource='<%=imagesPath + "check_step1_save.gif"%>'/>"
width="61" height="13" alt="" border="0"></a>
```

The extraParams parameter contains data about the product item, collected already in step1.jsp:

```
<%
    extraParams = HttpRequestConstants.CATALOG_ITEM_SKU + "=" +
                  shoppingCartLine.getProductItem().getKey().getIdentifier();
%>
```

The button.moveItemToSavedList event is defined in the checkoutportlet.wf Webflow file, which resides in the following directory:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\app
lication-sync\webapps\b2cPortal
```

The event is:

```
<event event-name="button.moveItemToSavedList">
       <destination namespace="checkoutportlet"
       node-name="shoppingcart_MoveProductItemToSavedListIP"
       node-type="inputprocessor"/>
</event>
```

The pipeline component is MoveProductItemToSavedListPC. You can view its Java source file in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\example
s\e2e\b2c\shoppingcart\pipeline
```

MoveProductItemToSavedListPC removes a ProductItem from a customer's shopping cart and adds it to their list of saved items. The customer's login id is specified by the PipelineSessionConstants.USER_NAME attribute in the Pipeline Session. The SKU of the item to move is specified by the PipelineSessionConstants.CATALOG_ITEM_SKU pipeline session attribute. The

PipelineSessionConstants.SAVED_SHOPPING_CART and PipelineSessionConstants.SHOPPING_CART pipeline session attributes, and the WLCS_SAVED_ITEM_LIST table are updated to reflect the change.

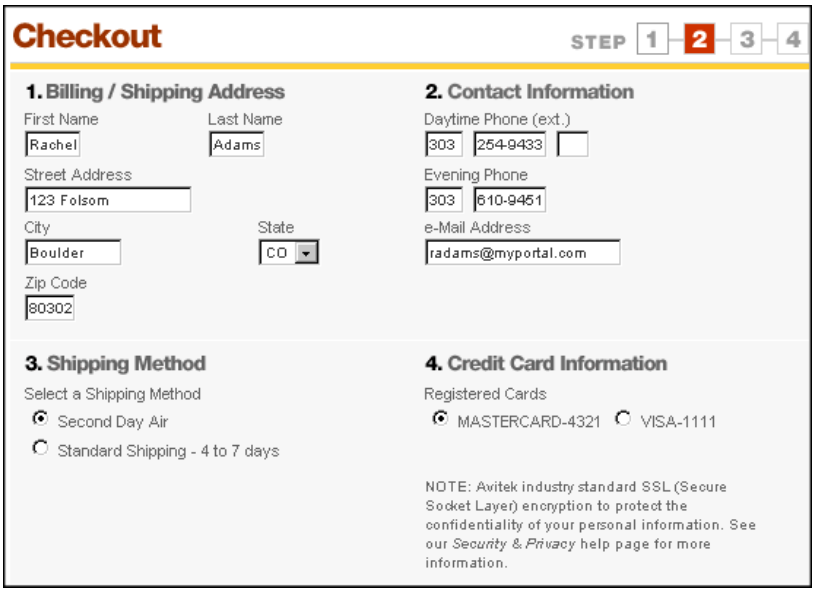
Next Step

To continue the tour, click the Checkout button near the bottom of the shopping cart step1.jsp portlet.

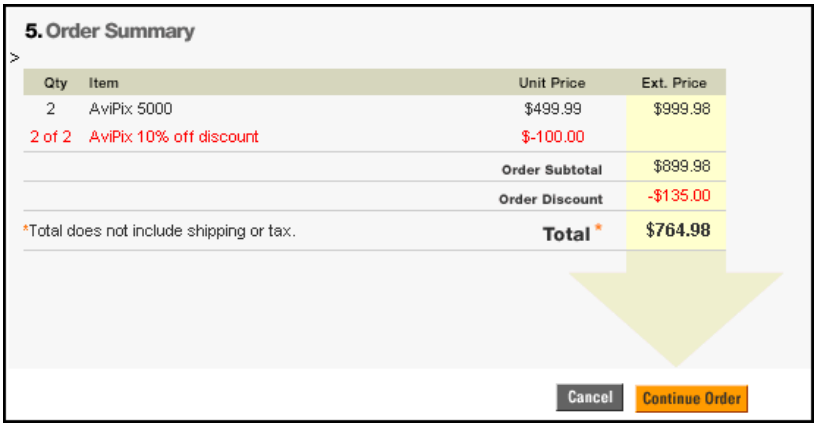
The Checkout Portlet, Step2.jsp

Having clicked the Checkout button on the step1.jsp Shopping Cart, the portal application's Webflow resulted in the display of the step2.jsp checkout portlet on this Shopping Cart page. A graphic, check_step2_header.gif, was loaded to help the customer understand where they are in the order fulfillment process. To simplify this sample, data for the logged-in user has been provided on the page. Notice that for Steps 2 - 4 of the checkout process, the Secure Socket Layer (SSL) protocol is being used for security encryption of the order data (https://... in the application's URL).

The Checkout portlet is shown here in two parts for formatting purposes only:


Checkout
STEP 1 2 3 4
1. Billing / Shipping Address
First Name
Rachel
Last Name
Adams
Street Address
123 Folsom
City
Boulder
State
CO
Zip Code
80302
2. Contact Information
Daytime Phone (ext.)
303
254-9433
Evening Phone
303
610-9451
e-Mail Address
radams@myportal.com
3. Shipping Method
Select a Shipping Method
Second Day Air
Standard Shipping - 4 to 7 days
4. Credit Card Information
Registered Cards
MASTERCARD-4321
VISA-1111
NOTE: Avitek industry standard SSL (Secure Socket Layer) encryption to protect the confidentiality of your personal information. See our Security & Privacy help page for more information.

Here is the lower portion of the same portlet:



Technical Details for the Checkout Portlet

The `step2.jsp` Checkout portlet presents pre-set billing and shipping information for logged-in user Rachel Adams. To keep the scope of this sample application simple, we did not include portlets that would allow the user to enter or modify their credit card or shipping information. However, WebLogic Portal supports this type of processing, and provides a separate sample to demonstrate it. Please refer to the Commerce and Campaign Features Tour in the WebLogic Portal documentation.

The `step2.jsp` Checkout portlet is in the following location:

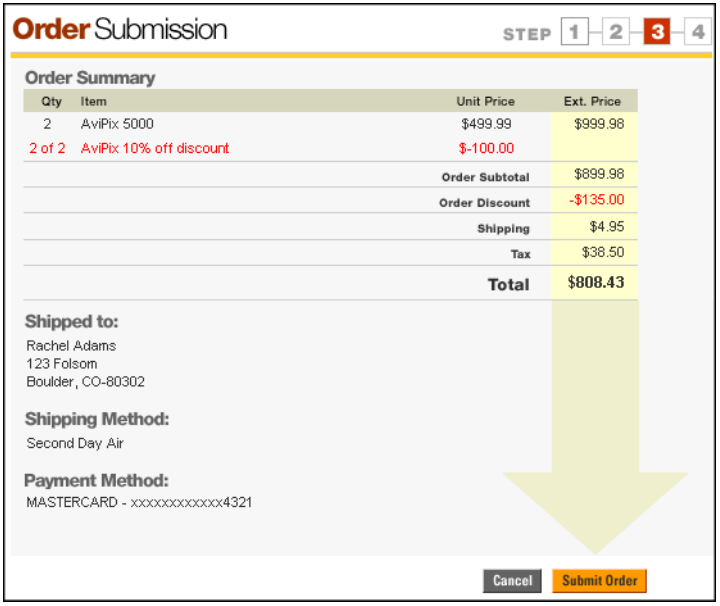
```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\portlets\checkout\*
```

Next Steps

To continue the tour, click the Continue Order button to proceed to the `step3.jsp` Order Submission page.

The Order Submission Portlet, Step3.jsp

Having clicked the Continue Order button in the Shopping Cart's `step2.jsp` Checkout portlet, the portal application's Webflow resulted in the display of this `step3.jsp` page. Summary information is presented. For example:



Technical Details for the Order Submission Portlet

This section provides details about the processing that occurs on this page.

Introduction

The `step3.jsp` checkout portlet provides the (pre-submit) Order Submission page, showing a summary of the order about to be submitted.

On this page, the interesting aspect is what happens after the user clicks the Submit Order button near the bottom of the page. The credit card authorization is performed via a pipeline component named `CajunBasedPaymentPC`. It calls a Payment Web service that we created in WebLogic Workshop.

After the order is confirmed (step 4), a pipeline component named `ConvertOrderRepPC` will convert the persisted order to an XML representation, and places it on a Java Message Server (JMS) queue. The WebLogic Integration Business Process Management (BPM) event processor dequeues the order and processes it. For information about that process, see the section "The Order Confirmation Portlet, Step4.jsp" on page 2-74.

Payment Authorization with a WebLogic Workshop Web Service

The Submit Order button on the `step3.jsp` portlet invokes a `link.next` event. The `checkoutportlet.wf` Webflow file includes the following:

```
<presentation-origin node-name="step3" node-type="jsp">
    <node-processor-infopage-name="step3.jsp"page-relative-path="/portlets
        /checkout"/>
    <event-list>
        <event event-name="link.next">
            <destination namespace="checkoutportlet"
                node-name="Commit" node-type="inputprocessor"/>
```

This Webflow file resides in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project
\application-sync\webapps\b2cPortal
```

The `b2c_order.pln` pipeline file resides in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\app
lication-sync\pipelines
```

It includes the following:

```
<root-component component-name="CommitOrderPC"/>
  <component-branch-item>
    <source-component component-name="CommitOrderPC"/>
    <branch-success destination-component="CajunBasedPaymentPC"/>
  </component-branch-item>
```

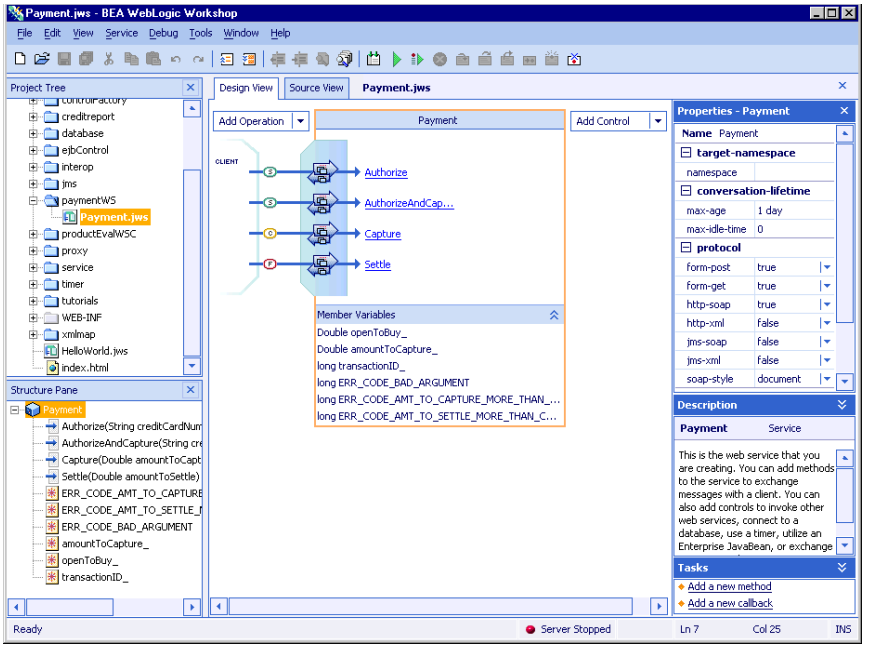
The credit card payment authorization processing is handled by a pipeline component named `CajunBasedPaymentPC`. It calls a Java proxy that lets the pipeline component call the Payment Web service. We created the Web service in WebLogic Workshop.

You can view the source file for the `CajunBasedPaymentPC.java` file in the following location:

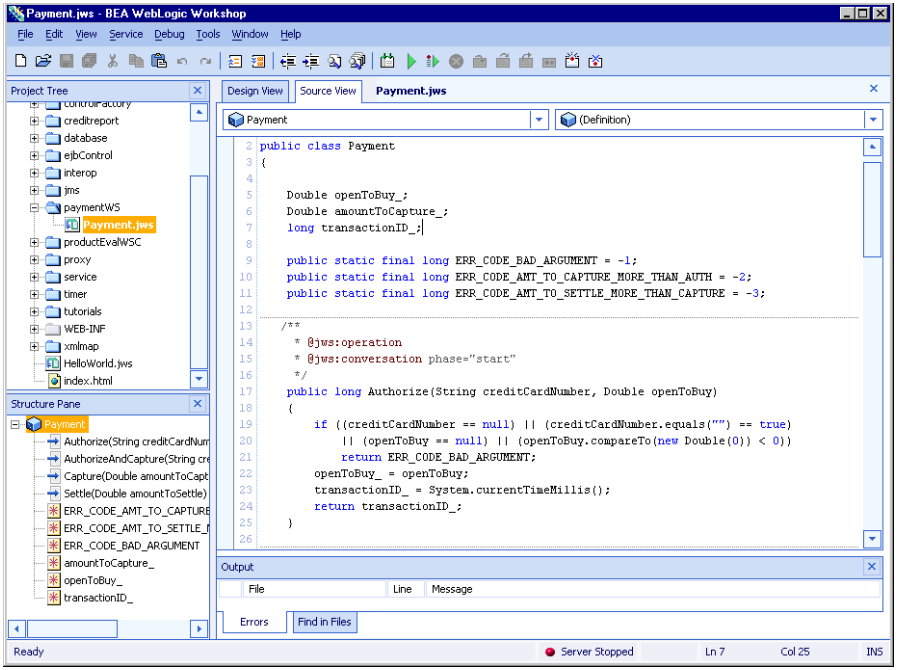
```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\examples\e2e\b2c\payment\pipeline
```

The Payment Web service uses the conversational aspect of WebLogic Workshop framework. The first call is to authorize the credit card; we pass in the credit card number and the amount to be authorized as arguments. After the credit card authorization is complete, a call is made to capture an amount. Eventually a request is made to settle the amount.

The following screens show a portion of the WebLogic Workshop Integrated Development Environment (IDE) that we used to create the Payment Web service. This first screen is the Design View.



In WebLogic Workshop, you can easily switch between Design View and the following Source View.



The codes returned by the Payment Web service to the `CajunBasedPaymentPC` are as follows:

- -1 = Error (Bad argument passed)
- -2 = Error (When amount to capture is more than amount the credit is authorized for)
- -3 = Error (When amount to settle is more than amount for which the credit card was captured for)
- > 0 = Success

Note: The Payment Web service always sends payment information through without any errors, as if it were connected to and approved by a third-party payment service. The processing of payment via the Payment Web service is not designed for production use. You must integrate with your third-party

vendor's payment service to process payment correctly. Note, however, that the code shown in the sample pipeline component is set up to appropriately handle error conditions.

For a complete description of the `CajunBasedPaymentPC` pipeline component and the Payment Web service, please see Chapter 4, "Web Services Tour." The description is also available from this sample's Introduction page.

A Note about the Cancel Button

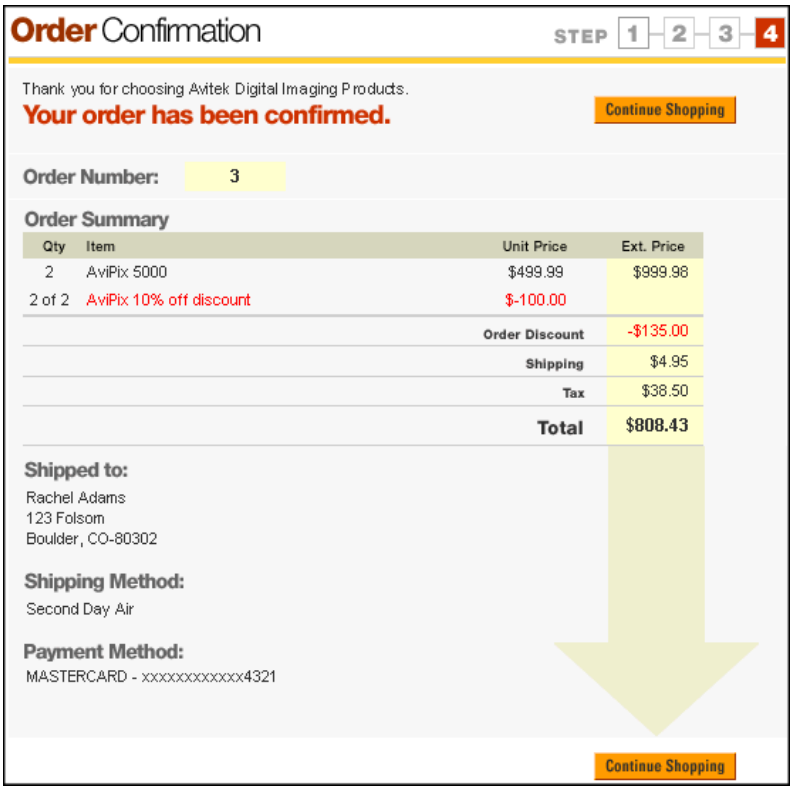
If you click the Cancel button on the `step2.jsp` or `step3.jsp` Shopping Cart portlet, the application's Webflow will bring you back to `step1.jsp` in the cart. The current and any saved items that were in the cart still remain there. The cancellation simply ended step 2 or 3 of the checkout process for that order.

Next Step

To continue the tour, click the Submit Order button.

The Order Confirmation Portlet, Step4.jsp

Having clicked the Submit Order button in the Shopping Cart's `step3.jsp` Order Confirmation portlet, the portal application's Webflow resulted in the display of this `step4.jsp` portlet. It lists a confirmation message. The persisted order is now submitted by a pipeline component that enables asynchronous communication between the portlet and business process workflows, via a JMS queue. This interaction demonstrates application integration between WebLogic Portal and WebLogic Integration, which are running in a single WebLogic Server domain instance.



Technical Details for the Order Confirmation Portlet

When the commerce order is persisted to the database, an XML representation of the same order is queued for order management. The work starts in the `ConvertOrderRepPC` pipeline component. You can find its Java source file in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\examples\e2e\b2c\order\pipeline
```

The `ConvertOrderRepPC` pipeline component works with a `PurchaseManager` SPI. Its source files are in the following location:

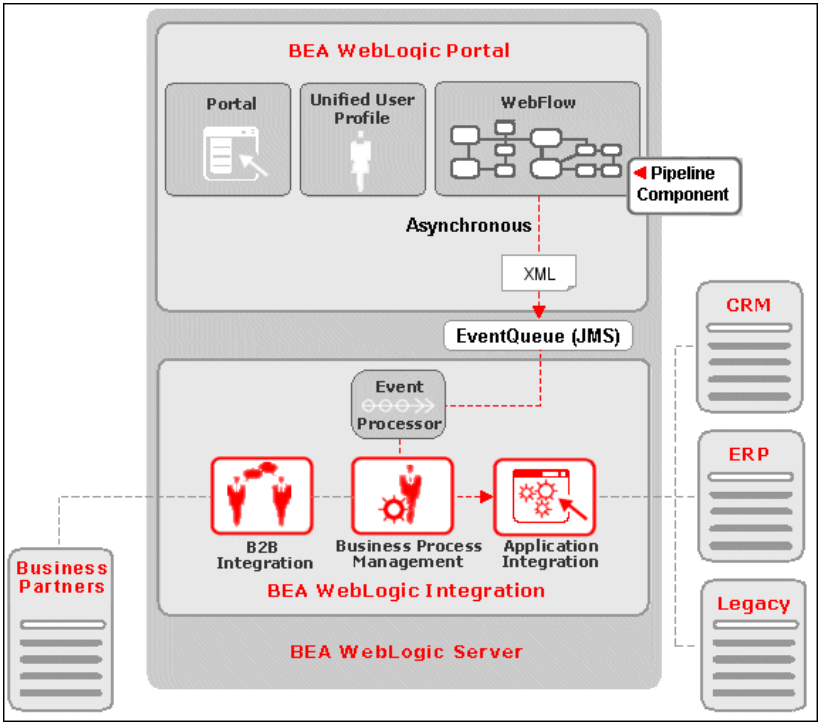
```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\examples\e2e\common\purchase\spi\*.java
```

This SPI is a stateless session bean that includes a `queueOrder` remote method.

WebLogic Integration has the following entry points available for communication with other systems:

- Application Integration (AI)—a system can call a J2EE Connection Architecture (J2EE CA) adapter directly using the AI entry point of WebLogic Integration. Typically, you use this for synchronous communication.
- Business-to-business integration (B2Bi)—a system can directly invoke business-to-business integration through a message API. This provides developers a way to write custom applications that communicate with a full instance of WebLogic Integration. B2Bi also provides a JavaServer Pages (JSP) tag library for building thin-client interfaces that can communicate with a hosted B2Bi enabler.
- WebLogic Integration process engine—a system can invoke the WebLogic Integration process engine synchronously using calls to a native API or asynchronously using JMS. These calls receive and send data, start processes, execute user-assigned tasks, and pass XML data among enterprise systems.

In this sample application, we are using asynchronous communication with a JMS queue. The following illustrates the process flow:



Again, `ConvertOrderRepPC` converts the order to a JMS XML message and sends it to a JMS queue (`com.bea.wlpi.EventQueue`) to which a BPM workflow is subscribed. Then the BPM event listener retrieves the message from the JMS queue and processes the message. The message either starts a workflow or triggers a workflow event listened to by a running workflow instance. WebLogic Integration retrieves the XML message from the JMS queue and forwards the data to a database, using the sample DBMS adapter provided by WebLogic Integration for use with its application integration component.

The XML message is used as the input document to the workflow. The workflow responds with two actions:

It parses the XML message using XPath and passes all the input data to the application integration service.

It calls an application integration service that is defined when the user deploys an application view for the DBMS adapter. The application view service updates the database.

Final Step for the BC2 Portal Tour

Now that the order has been placed successfully, if you wish you may continue shopping in the online tour by clicking the Continue Shopping button or Products tab. Or you can click the Logout button to return to the Introduction page for this sample. Note that if you logout, the Shopping Cart data is not retained for the next session.

CHAPTER

3 Avitek Purchasing Agents Connect with Suppliers

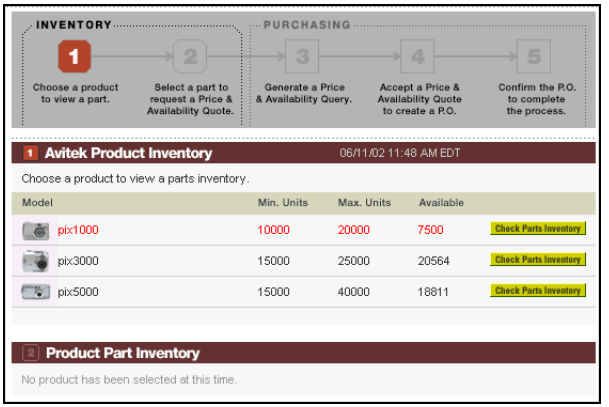
The business-to-business (B2B) portal tour describes a fictitious intranet site that Avitek purchasing agents use to get quotes from suppliers, and to submit purchase orders and get acknowledgements from suppliers. The business processes are managed by WebLogic Integration.

Note: The information that is presented in this online book is also available in a context-sensitive tour guide portlet that runs as part of the application.

This business-to-business (B2B) portal tour contains the following sections:

The Product Inventory Portlet

The B2B portal sample application starts on the Avitek intranet's Inventory page. A graphic, `processStep1.gif`, has been loaded to help logged-in user Jason Tang understand the sequence of steps in this process. If this is your initial visit to the page, the Product Inventory portlet contains preloaded data for some of the products sold by Avitek. If the inventory for any product is below its minimum level, the data is shown in red.



Technical Details for the Product Inventory Portlet

This section provides details about the processing that occurs on this page.

Introduction

The b2bPortal is a sample business-to-business site used by the fictitious company Avitek Digital Imaging. Avitek purchasing agents use the site to get quotes for parts from external suppliers, and to submit purchase orders.

The b2bPortal is part of the `e2eApp` enterprise application. The enterprise application's name includes `e2e` an abbreviation for "end-to-end," meaning a sample that shows a full range of key features in WebLogic Platform.

On this Inventory page, the Product Inventory portlet contains preloaded data for some of the products sold by Avitek. If the inventory for any product is below its minimum level, the data is shown in red. To get current inventory levels, an Inventory table in the database has been accessed in read-only mode via the Application Integration (AI) framework. The AI framework is provided by WebLogic Integration.

In our sample scenario, Avitek Digital Imaging wanted to reduce costs by making its supply chain more efficient through automation, compressed cycle times, and lower inventory levels. To track and replenish inventory levels, Avitek implemented a business-to-businesses solution with a portal user interface for sell-side B2B exchanges, partner collaboration, and supply chain management.

If you have used the business-to-consumer b2cPortal portal in this sample application, you know that the Technical Details sections included explanations about the portlets and Webflow processing. Although they are still integral parts of this b2bPortal, the explanations in this part of the sample tour focus more on the supply-chain solution implemented for Avitek purchasing agents. This section, however, covers some portlet and Webflow details.

Trading Partners

This WebLogic Platform b2bPortal sample scenario involves three business partners: a buyer (Avitek Digital Imaging) and two suppliers. For each business partner, a trading partner is configured in the `BulkLoaderData.xml` file. The following trading partners are defined for the sample: `E2E_Buyer, E2E_SupplierOne,` and `E2E_SupplierTwo`.

Because these trading partners communicate using the XOCP business protocol, Avitek must define its WebLogic Integration system as a hub-and-spoke configuration. To that end, the `BulkLoaderData.xml` file defines a fourth trading partner: `E2E_Hub`. (See the topic "Getting Started with B2B Integration" in the document Introducing B2B Integration for details about configuring B2B integration.)

The `E2E_Hub` trading partner acts as an intermediary. It is responsible for mediating messages between the spoke trading partners: `E2E_Buyer`, `E2E_SupplierOne`, and `E2E_SupplierTwo`. The `E2E_Hub` trading partner is not the sender or receiver of a business message, but it acts as the proxy buyer and proxy supplier, at different times during the transaction.

Each of the three trading partners—`E2E_Buyer`, `E2E_SupplierOne`, and `E2E_SupplierTwo`—has a collaboration agreement with the `E2E_Hub` trading partner. The `E2E_Hub` trading partner is responsible for linking collaboration agreements. Such

linking is essential, for example, when a message arrives as part of one collaboration agreement and must be routed to another trading partner as part of another collaboration agreement. Collaboration agreements that use the same delivery channel—the channel defined for the `E2E_Hub` trading partner—are linked.

Each trading partner element is characterized by various attributes and subelements, some of which contain simple identification information, such as name, e-mail, phone, and fax.

The following table summarizes the role of each trading partner.

Table 3-1 Trading Partner Roles

Partner Name	Role
`E2E_Hub`	Routes the communication between the buyer and suppliers, providing business-to-business integration
`E2E_Buyer`	Coordinates business processes among suppliers and internal functions (for example, back-end database updates), using workflow templates. Provides connectivity to the buyer's database system, using an application view. Handles data presentation and the user interface through HTML and JSP pages.
`E2E_SupplierOne`	Responds to requests from the buyer and invokes internal programs (for example, data transformation and persistence), using workflow templates. Performs data translations to facilitate the exchange of information among applications.
`E2E_SupplierTwo`	Same role as `E2E_SupplierOne`.

Using the WebLogic Integration Studio

The WebLogic Integration Studio allows you to design new workflows and monitor running workflows using a familiar flowchart paradigm. You are not required to run the Studio when you run this b2bPortal sample, but you may find it useful for viewing the details of any workflow or workflow node, and for learning how nodes are defined and configured for this sample. You can also use the Studio to monitor the workflows as you run the sample.

On a Windows system, choose Start →Programs →BEA WebLogic Platform 7.0 → WebLogic Integration 7.0 →Studio.

Log on to the Studio:

- User: `admin`
- Password: `security`
- Server: `t3://localhost:7501`

For example:

Note: Use **t3**, not http.

Viewing Workflow Templates in the Studio

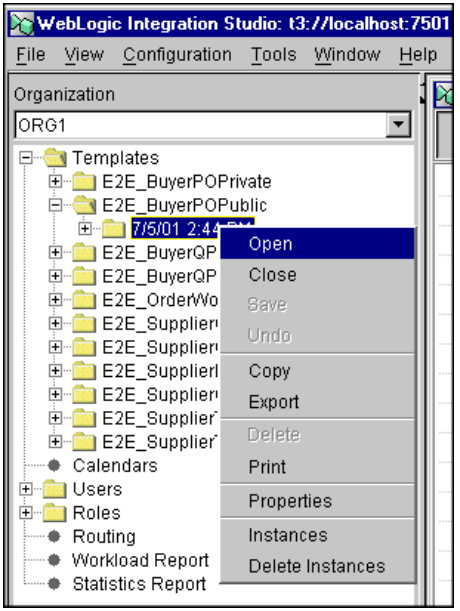
To view a workflow template and its properties in the Studio, complete the following steps:

1. In the left pane of the Studio, make sure ORG1 is selected in the Organization field.
2. In the left pane, double-click the Templates folder to display a list of workflow templates.

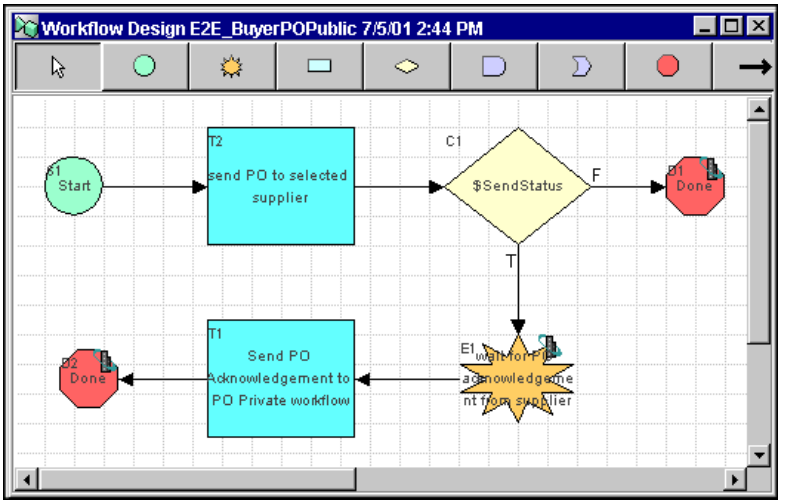
3. Expand the Templates folder to display the list of workflow template definitions. They are defined in the workflows.jar file that is preconfigured by this sample. The worksflows.jar file resides in:

   `weblogic700\samples\integration\samples\e2e\workflows`

4. Right-click a template definition, and select Open to open the workflow template in the Studio. For example:



This step opens a graphical view of the template in the Studio. For example:



Note: You can also expand a particular workflow template definition to display folders containing the Tasks, Decisions, Events, Joins, Starts, Dones, and Variables for that workflow template definition.

Double-click any node in the Studio to display the Properties dialog box for that node. See Using the WebLogic Integration Studio in the WebLogic Integration documentation for complete details about the Studio tools and functionality.

Business Process and Workflow Modeling

This section presents a brief introduction to the business process management (BPM) functionality provided by WebLogic Integration.

Workflows that implement the roles assigned to trading partners in a conversation definition are referred to as *collaborative workflows*.

A workflow template represents a workflow, and combines various workflow template definitions (versions) of its implementation. Workflow templates are designed and edited in the WebLogic Integration Studio. Several BPM plug-ins extend the functionality of the Studio:

- B2B integration plug-in—supports B2B integration, that is, the design and management of collaborative workflows. The Studio allows you to assign

properties to the workflows. These properties make the workflows usable in the B2B integration environment.

- Application integration plug-in—allows you to design workflows that can integrate back-end and legacy enterprise information systems (EIS).
- Data integration plug-in—allows you to design workflows that integrate heterogeneous data formats, by making it possible to share data among heterogeneous EIS applications.

In this sample scenario, trading partners implement both *private and collaborative workflows*. Private workflows work in conjunction with the collaborative workflows, and implement local processes for the trading partners. Local and private processes are not necessarily dictated by the conversation definition. For example, when a trading partner starts a conversation, that trading partner's private workflow can start the collaborative workflow to initiate the conversation.

Please see the WebLogic Integration documentation for complete descriptions. In this sample, the Technical Details sections for subsequent portal pages will describe the two business processes implemented by this sample application: Query Price and Availability (QPA) and Purchase Order (PO).

Inventory Page Portlets

The Inventory page is one of three tabbed pages in the b2bPortal. In the `BEA_HOME` directory where you installed WebLogic Platform, you will find the files that comprise the `e2eApp` in the following locations:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\...

weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\...
```

This Inventory page can include the following portlets:

- The **`login`** portlet, used when an authenticated user has been logged into the application. It provides the Logout button and the name of the currently logged-in user.
- The **`productinventory`** portlet, which will identify the inventory levels of the products that the purchasing agent manages.
- The **`purchasingprocess`** portlet, which provides the appropriate graphic in the banner to show the purchasing agent the sequence of steps in the ordering process.

- The **`b2b-tourguide`** portlet, which provides the context-sensitive documentation in the running sample. . It has two forms: the smaller version on the left side of the running application, and the current maximized version that includes Technical Details, View Code, and e-docs pointers.

All of the portlets that comprise the Inventory page are identified in the following file:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project
\application-sync\webapps\b2bPortal\b2bPortal.portal
```

For example:

```
<page-name>Inventory</page-name>
...
    <portlet-pool>
        <portlet-name>login</portlet-name>
        <portlet-name>partinventory</portlet-name>
        <portlet-name>productinventory</portlet-name>
        <portlet-name>purchasingprocess</portlet-name>
        <portlet-name>b2b-tourguide</portlet-name>
        <portlet-name>debug</portlet-name>
        <portlet-name>anonUser</portlet-name>
    </portlet-pool>
```

During the development cycle, these portlets were added to the Inventory page using the E-Business Control Center. The EBCC is a Java client-based tool suite. It provides graphical interfaces that simplify complex tasks such as rule definitions, Webflow editing, and portal creation and management. As users of the E-Business Control Center work with its point-and-click interface, it generates XML files that are synchronized with the server. In addition to the EBCC, the browser-based Portal Administration tools were used for administering and managing the portal at runtime.

Portal developers are primarily interested in the portlet JSP code before it was rendered by the browser. That's why the code fragments in this section, and the View Code link, describe and show you the pre-rendered JSP file for a particular portlet.

You can find the b2bPortal's portlet JSP files in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2bPortal
\portlets\...
```

For this Inventory page, the View Code link opens the source for the `\productinventory\content.jsp` portlet file. It presents the Avitek Product Inventory (1) portlet. This content.jsp resides in the productinventory subdirectory under the path shown above.

Outline of Initial Portal Processing

When you started the sample application, what happened that resulted in the server startup and then the initial display of the Introduction or "splash" page? First, there were several ways you could have invoked the sample application, such as from the WebLogic Platform Quick Start Application or directly by running a `startE2E` script.

Regardless of which option you used, the `startE2E.bat` (Windows) or startE2E.sh (UNIX) script was invoked. It started a WebLogic Server instance for the application, which runs in a domain named `e2eDomain`. The word "domain" has many meanings in the computing industry. BEA products use **domain** to mean a collection of servers, services, interfaces, machines, and associated resource managers, all defined by a single configuration file.

When the `startE2E` script runs, it reads configuration information from the enterprise application's `config.xml` file. By default this configuration file resides in the following `BEA_HOME` installed directory:

```
weblogic700/samples/platform/e2eApp/config
```

The `config.xml` file includes the following definition, setting `splashPage` as the default Web application in the domain:

```
<WebServer
       DefaultWebApp="splashPage"
       LogFileName="./logs/access.log"
       LoggingEnabled="true"
       Name="e2eServer"
       />
```

With the `e2eServer` running, specifying a URL such as `http://localhost:7501` in a browser results in running the `splashPage` Web application (or "webapp") in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\splashPage
```

The `splashPage` webapp's `web.xml` configuration file designated `index.jsp` in the `<welcome-file-list>` definition. This `web.xml` resides in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\splashPage\
WEB-INF
```

On the splash page, you clicked the "AUTO LOGIN as purchasing agent Jason Tang" button graphic to arrive on this Inventory page. That resulted in the splash page passing in the URL for the portal application, plus predefined login credentials.

The splash page sets up requests that allow you to automatically log into the b2bPortal or b2cPortal. This is done to help simplify this sample application. You would not want to embed usernames and passwords in a JSP page.

The form results in opening the URL constructed as:

```
<%
       String b2bUrl = "http://" +
              request.getServerName() + ":" +
              request.getServerPort() + "/" +
              B2B_PORTAL_NAME + "/application";
...
```

For this WebLogic Platform sample application, we elected to automatically log in an existing user, in order to focus the sample on the most important features. WebLogic Portal provides other samples that demonstrate login authentication code, plus techniques to gather demographic information via Web applications. For details, please refer to the *WebLogic Portal Developer Guide*.

Once the URL was passed to the browser for the b2bPortal application, why was the Inventory page the first page displayed? This property was set in the WebLogic Portal Administration Tools. In the Portal Management section of that tool, under Pages and Portlets, we set the b2bPortal's default page on the Select and Order Pages screen. This value is stored in the database, in the `INDEX_NUMBER` column of the `PORTAL_PAGE_P13N` table.

Page Change Webflow Events

When a customer clicks a link or button on a JSP, it is considered an event. Events trigger particular responses in the default Webflow that allow customers to continue. While this response can be to load another JSP, it is usually the case that an Input Processor and/or Pipeline is invoked first.

When you are on a Purchasing or Order History page and then click the Inventory tab, you will notice a resulting URL similar to the following (shown here on several lines to improve readability).

```
http://<host>/<port>/b2bPortal/application?
origin=hnav_bar.jsp&event=bea.portal.framework.internal.refresh
&pageid=Inventory
```

The refresh event causes any page to be displayed again with the latest data. The Avitek Product Inventory portlet has its data refreshed.

The page tabs are provided via the `hnav_bar.jsp` file, which resides in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2bPortal
\framework
```

The `hnav_bar.jsp` file imports two JSP Tag libraries:

```
<%@ taglib uri="webflow.tld" prefix="wf" %>
<%@ taglib uri="portal.tld" prefix="ptl" %>
```

When you click the Inventory tab from another page in the application, the `hnav_bar.jsp` uses the following JSP tag in the link for the target tab:

```
<a href="<ptl:createPortalPageChangeURL
pageName='<%= portalPageName %>'/>"><%=portalPageName%></a>
```

The (portal) `ptl:createPortalPageChangeURL` JSP tag generates a webflow URL for a page change event.

Dynamic Portlet Display and Inventory Checks Via WebLogic Integration AI

On the Inventory page, let's look at one of the dynamic portlets, `\productinventory\content.jsp`, shown as #2, “Avitek Product Inventory” on the Inventory page.

The webflow namespace file for this portlet is:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project\app
lication-sync\webapps\b2bPortal\product.wf
```

It contains:

```
<presentation-origin node-name="product" node-type="jsp">
       <node-processor-info page-name="content.jsp"
       page-relative-path="/portlets/productinventory"/>
</presentation-origin>
```

The `content.jsp` file is located in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2bPortal
\portlets\productinventory\content.jsp
```

The `content.jsp` file includes the following import statement:

```
<%@ page import="examples.e2e.common.Inventory" %>
```

In a scriptlet in `content.jsp`, we iterate through the product data in the database to get current inventory levels. When levels are below the defined minimum, we use a `CSS_INV_BELOW_MIN` setting from the CSS file, displaying the inventory problem in red. For example:

```
<%
        Iterator it = rState.getProducts().iterator();
        Inventory prod = null;
        String rowCssClass = null;
        String extraParams = null;
        int i = 0;
        for ( ; it.hasNext(); i++ ) {
        prod = (Inventory) it.next();
        if ( prod.isBelowMinimum() ) {
               rowCssClass = CSS_INV_BELOW_MIN;
        }
        else {
               rowCssClass = CSS_PRODUCT_ROW;
        }
        extraParams = PRODUCT_PARAM_EQUALS + prod.id();
        // skip the row divider the first time through
        if ( i != 0 ) {
%>
```

For the b2cPortal and this b2bPortal application, we created a service provider interface (SPI) named `InventoryProvider`. It is implemented as a stateless session EJB and has three methods:

`checkInventory()`

- Uses the WLI AI framework to make a call via an already configured AI service. This service queries the inventory table to select the quantity from `E2E_PRODUCT_INV where sku = value`.
- Calls to an XML helper to parse the response from AI.
- Returns available inventory (`int`) based on response.

`getProductInventory()`

- Uses AI to make a call via an already configured AI service. This service will be to query the inventory table. For example: `select sku, desc, minimum, maximum, quantity from E2E_PRODUCT_INV where parent_sku = NULL`.
- Call to an XML helper to parse the response from AI to create a List of Inventory objects.

getProductPartInventory()

- Uses AI to make a call via an already configured AI service. This service will be to query the inventory table. For example: `select sku, desc, minimum, maximum, quantity from E2E_PRODUCT_INV where parent_sku <> NULL`.
- Calls to an XML helper to parse the response from AI to create a List of Inventory objects.

The `InventoryProvider` SPI is common between the b2bPortal and b2cPortal. Its source files are in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\examples\e2e\common\inventory\spi\*.java
```

For the b2bPortal, the inventory check is implemented by:

- `CheckInventoryAction`
- `GetProductInventoryAction`
- `GetProductPartInventoryAction`

The Java source files for these programs are in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\examples\e2e\common\inventory\ref\*.java
```

These programs use the `getProductInventory()` and `getProductPartInventory()` methods provided in the EJB. To simplify this sample application, when using the `getProductInventory( List SKUs )` method, the List of SKUs will be hardcoded in the pipeline component.

In the `productinventory\content.jsp` portlet, we then display the data returned. For example:

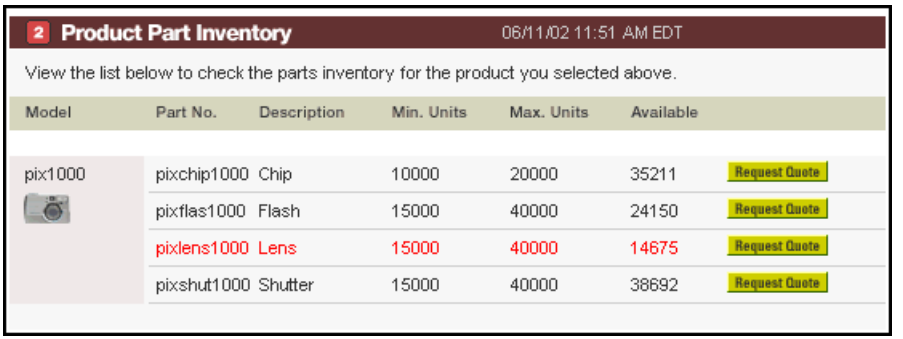
```
<!-- model number -->
    <td width="187" class="<%= rowCssClass %>"><%= prod.id() %></td>

<!-- minimum # of units -->
    <td width="75" class="<%= rowCssClass %>"><%= prod.min() %></td>

<!-- maximum # of units -->
    <td width="75" class="<%= rowCssClass %>"><%= prod.max() %></td>

<!-- available # of units -->
    <td width="67" class="<%= rowCssClass %>"><%= prod.available() %></td>
```

Next Step

To continue the tour, click the Check Parts Inventory button for the **pix1000** camera.

The Product Parts Inventory Portlet

You may need to scroll down to see all of the Product Part Inventory portlet (Step 2), which lists the parts that comprise the Avitek product you selected in Step 1. Inventory levels are shown for each part. Data for any individual parts that are below minimum inventory levels are shown in red.



Technical Details for the Product Parts Inventory Portlet

The processing for the Product Part Inventory portlet, Step 2 in our inventory process, is similar to the Product Inventory portlet. If the inventory for any product part is below its minimum level, the data is shown in red. To get current inventory levels, an Inventory table in the database has been accessed in read-only mode via the Application Integration (AI) framework. The AI framework is provided by WebLogic Integration. In this case, the `getProductPartInventory()` method provided by the `InventoryProvider` SPI was used.

Product Part Inventory Check Via WebLogic Integration AI

On the Inventory page, let's look at one of the dynamic portlets, `\partinventory\content2.jsp`, shown as #2, "Avitek Product Inventory" on the Inventory page. The `content2.jsp` file is active after a product was selected in the Product Inventory portlet. (The `\partinventory\content.jsp` portlet is the inactive version, used when no product has been selected yet on the Inventory page.)

The webflow namespace file for this portlet is:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project
\application-sync\webapps\b2bPortal\part.wf
```

It contains:

```
<presentation-origin node-name="product" node-type="jsp">
       <node-processor-info page-name="content2.jsp"
       page-relative-path="/portlets/productinventory"/>
</presentation-origin>
```

The `content2.jsp` file is located in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2bPortal
\portlets\partinventory\content2.jsp
```

The `content2.jsp` file includes the following import statement:

```
<%@ page import="examples.e2e.common.Inventory" %>
```

In a scriptlet in `content2.jsp`, we iterate through the product and part data in the database to get current inventory levels. When levels are below the defined minimum, we use a `CSS_INV_BELOW_MIN` setting from the CSS file, displaying the inventory problem in red. For example:

```
<%
    String rowCssClass = null;;
    String extraParams = null;

      int i = 0;
      for ( ; parts.hasNext(); i++ ) {

          part = (Inventory) parts.next();

          if ( part.isBelowMinimum() ) {
              rowCssClass = CSS_INV_BELOW_MIN;
          }
          else {
```

```
                rowCssClass = CSS_PART_ROW;
            }

            extraParams = PART_PARAM_EQUALS + part.id();

%>
```

For the b2cPortal and this b2bPortal application, we created a service provider interface (SPI) named InventoryProvider. It is implemented as a stateless session EJB and has three methods:

`checkInventory()`

- Uses the WLI AI framework to make a call via an already configured AI service. This service queries the inventory table to (for example) `select quantity from E2E_PRODUCT_INV where sku = value`.
- Calls to an XML helper to parse the response from AI.
- Returns available inventory (int) based on response.

`getProductInventory()`

- Uses AI to make a call via an already configured AI service. This service will be to query the inventory table. For example: `select sku, desc, minimum, maximum, quantity from E2E_PRODUCT_INV where parent_sku = NULL`.
- Call to an XML helper to parse the response from AI to create a List of Inventory objects.

`getProductPartInventory()`

- Uses AI to make a call via an already configured AI service. This service will be to query the inventory table. For example: `select sku, desc, minimum, maximum, quantity from E2E_PRODUCT_INV where parent_sku <> NULL`.
- Calls to an XML helper to parse the response from AI to create a List of Inventory objects.

The `InventoryProvider` SPI is common between the b2bPortal and b2cPortal. Its source files are in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\examples\e2e\common\inventory\spi\*.java
```

For the b2bPortal, the inventory check is implemented by:

- `CheckInventoryAction`
- `GetProductInventoryAction`
- `GetProductPartInventoryAction`

The Java source files for these programs are in the following location:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\example
s\e2e\common\inventory\ref\*.java
```

These programs use the `getProductInventory()` and `getProductPartInventory()` methods provided in the EJB. To simplify this sample application, when using the `getProductInventory( List SKUs )` method, the List of SKUs will be hardcoded in the pipeline component.

In the `partinventory\content2.jsp` portlet, we then display the data returned. For example:

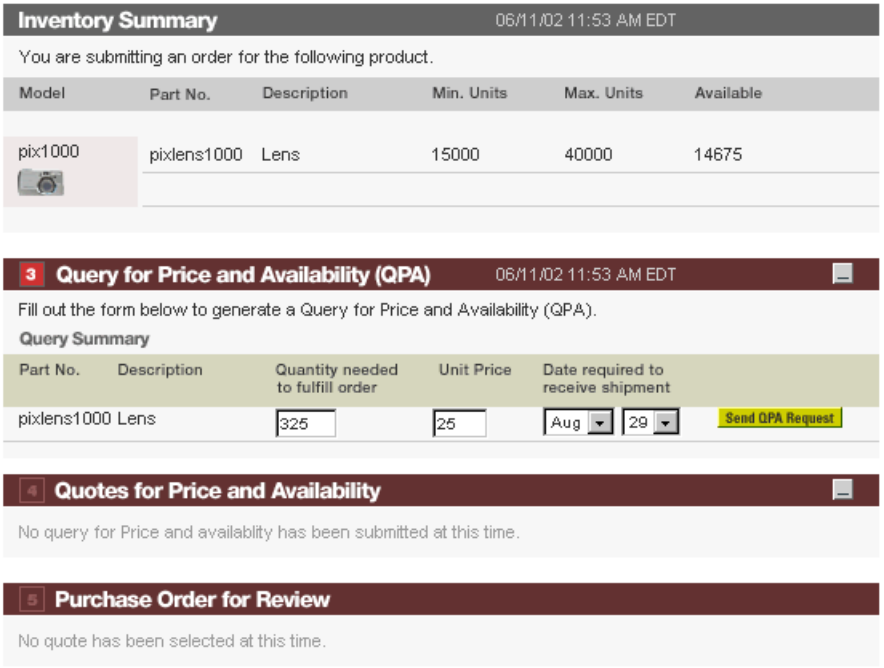
```
<td class="<%= rowCssClass %>" width="64"><%= part.id() %></td>

<td class="<%= rowCssClass %>" width="83"><%= part.description() %></td>

<td class="<%= rowCssClass %>" width="74"><%= part.min() %></td>

<td class="<%= rowCssClass %>" width="75"><%= part.max() %></td>

<td class="<%= rowCssClass %>" width="60"><%= part.available() %></td>
```

Next Step

To continue the tour, in the #2 Product Part Inventory portlet, click the Request Quote button next to a part that is shown in red.

The Query for Price and Availability Portlet

You may need to scroll down to see the Query for Price and Availability (QPA) portlet (Step 3). It provides a form to generate a QPA. To make it easier for the purchasing agent to focus on data needed for the QPA request, the Product Inventory and Product Part Inventory portlets have been replaced by an Inventory Summary portlet. Be sure to see the Next Steps section below for information about filling in the requested input on this form.



Technical Details for the Query for Price and Availability Portlet

The Query for Price and Availability (QPA) portlet simply provides a form to generate a QPA. The QPA business process, which starts when you click the Send QPA Request button on this Step 3 portlet, is described in the section "The Quotes for Price and Availability Portlet, and the QPA Business Process" on page 3-21.

On this Query for Price and Availability portlet, to make it easier for the purchasing agent to focus on data needed for the QPA request, the Product Inventory and Product Part Inventory portlets have been replaced by an Inventory Summary portlet.

When you return to the sample application, be sure to follow the advice in the Next Steps section of the Tour Guide. It is worth repeating here.

On this Query for Price and Availability portlet:

- Enter the quantity needed to fulfill the order.
- Then enter a unit price for the product part; for example, enter 50.00.
- Also enter the date that is required to receive the shipment; for example, enter a date one week from today.

Then click the Send QPA Request button.

If you see the following errors, enter the values again, and then click the Send QPA Request button:

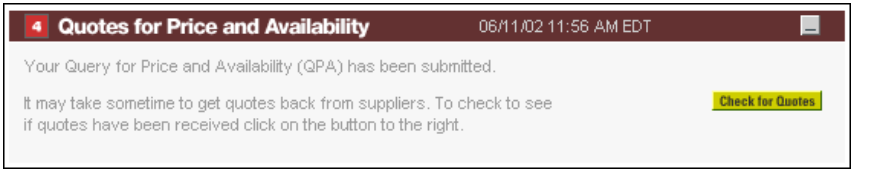
- "The value supplied for the quantity field is below the minimum allowed of 1"
- "A value is required for the unit price field"
- "The date chosen is prior to the current date"

Next Steps

If you have not already done so, enter the quantity needed to fulfill the order. Then enter a unit price for the product part; for example, enter 50. Also enter the date that is required to receive the shipment; for example, enter a date one week from today. Then click the Send QPA Request button to continue the tour.

The Quotes for Price and Availability Portlet, and the QPA Business Process

Initially the Step 4 portlet, Quotes for Price and Availability, presents a message that it may take some time to get quotes back from suppliers.



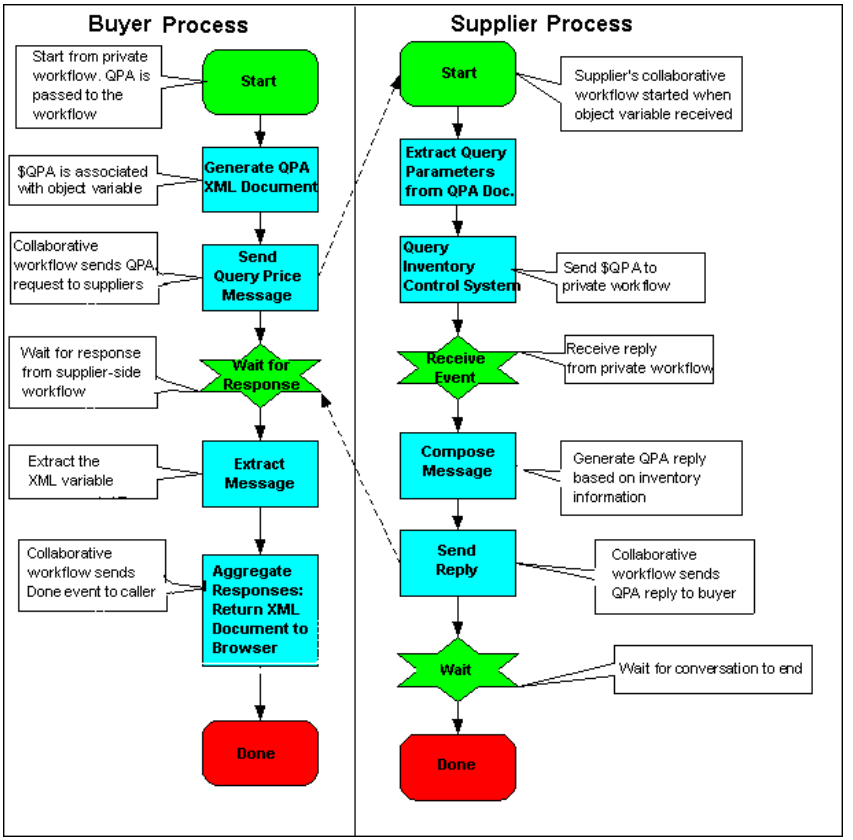
After you select the Check for Quotes button one or more times, the portlet will eventually be refreshed with quotes from suppliers. This processing demonstrates integration between the portal framework in WebLogic Portal and the business-to-business interaction managed by WebLogic Integration.

Technical Details for the QPA Business Process

This section focuses on the Query for Price and Availability (QPA) **business process**, which was started when you clicked the Send QPA Request button on the prior Step 3 portlet.

Due to the shortage of the **pixlens1000 part** for the pix1000 camera, the Avitek purchasing agent sent a QPA message for this part to selected suppliers. The following diagram illustrates the flow of events for the QPA business process:

Process Flow in the QPA Business Process



The following sequence summarizes the events illustrated in the preceding figure:

1. The buyer trading partner creates a QPA.
2. The QPA is sent to suppliers.
3. The suppliers process the QPA and generate responses.
4. The buyer trading partner collects and aggregates responses from the suppliers.

Note: The preceding figure shows a high-level view of the QPA business process. Each side of the process is implemented by a public (collaborative) and a private workflow.

The workflow templates listed in the following table are used in this sample's QPA process.

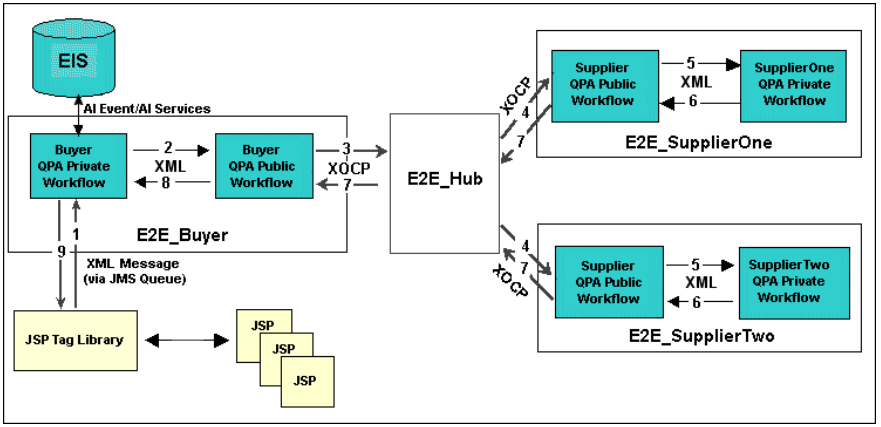
Table 3-2 QPA Process Workflow Templates

Role	Public/Private	Workflow Name
Buyer	Private	`E2E_BuyerQPAPrivate`
Buyer	Public	`E2E_BuyerPOPublic`
Supplier	Public	`E2E_SupplierPOPublic` **Note:** Both suppliers in the scenario use the same public workflow.
Supplier	Private	`E2E_SupplierOnePOPrivate`
Supplier	Private	`E2E_SupplierTwoPOPrivate`

WebLogic Integration manages the business conversations and collaboration agreements between business partners, and it automates the business message exchange between the buyer and suppliers. The workflows are referenced in the collaboration agreements and conversations.

This sample uses JSPs and JSP tag libraries to initiate the QPA process and display QPA request and response data. The following figure illustrates the data flow among the trading partners involved in the QPA business transaction.

Data Flow in the QPA Business Process



The following sequence of events summarizes the data flow among trading partners and workflows:

1. The portlet containing the QPA form sends the QPA request to a JMS queue and triggers the `E2E_BuyerQPAPrivate` workflow.
2. The `E2E_BuyerQPAPrivate` workflow invokes the `E2E_BuyerQPAPublic` workflow, passing it the QPA request XML document. It then initiates the QPA conversation.
3. Based on the collaboration agreement between the `E2E_Buyer` and the `E2E_Hub` trading partners, the `E2E_BuyerQPAPublic` workflow packs the QPA request XML into an XOCP message and sends it to the `E2E_Hub` trading partner.

   Note: When the `E2E_Hub` trading partner receives a message, it is acting as the proxy supplier in the collaboration agreement.
4. The `E2E_Hub` trading partner routes the message to the destination trading partners, `E2E_SupplierOne` and `E2E_SupplierTwo`, based on registered collaboration agreements between itself and each supplier.

   Note: In this step, the `E2E_Hub` trading partner changes roles and becomes the proxy buyer in the collaboration agreements between itself and the suppliers.

Each supplier's public workflow is triggered by receiving the XOCP message. In this scenario, `E2E_SupplierOne` and `E2E_SupplierTwo` share the public workflow (`E2E_SupplierQPAPublic`). The public workflow extracts the QPA request XML document from the message.

5. The `E2E_SupplierQPAPublic` workflow invokes a private workflow for each supplier and passes the QPA request XML document to the private workflows.
6. Each supplier's private workflow creates its own QPA response (an XML document), and attaches it to the return variable of the public workflow.
7. The `E2E_SupplierQPAPublic` workflow extracts the QPA response XML document, packs it into an XOCP message, and sends it to the buyer.

   Note that the `E2E_Hub` trading partner acts as a routing proxy for `E2E_Buyer`. When the supplier trading partners send response messages to `E2E_Hub` (based on collaboration agreements between `E2E_Hub` and each supplier trading partner), `E2E_Hub` acts as the proxy buyer.

   `E2E_Hub` then changes roles to that of proxy supplier, and routes the response messages to the buyer (`E2E_Buyer`), based on the collaboration agreement between `E2E_Hub` and `E2E_Buyer`.
8. The buyer's public workflow (`E2E_BuyerQPAPublic`):
   - Extracts the QPA response XML document from the XOCP message it receives.
   - Aggregates both supplier response documents into a single XML document and posts it, via a JMS queue, to the buyer's private workflow (`E2E_BuyerQPAPrivate`).
   - Terminates the QPA conversation and notifies the supplier public workflow (`E2E_SupplierQPAPublic`).
9. The buyer's private workflow (`E2E_BuyerQPAPrivate`) receives the aggregated QPA response XML document and writes it to an XML file. A JSP parses the XML and displays the aggregated QPA response in the Web browser.

   This step marks the end of the QPA business process.

For more detailed information about this process, see the WebLogic Integration documentation.

Next Steps

On the Quotes for Price and Availability portlet, if you have not received quotes back from the suppliers yet, click the Check for Quotes button again. When quotes have been returned, accept one of the quotes and then click the Create Purchase Order button to continue the tour.

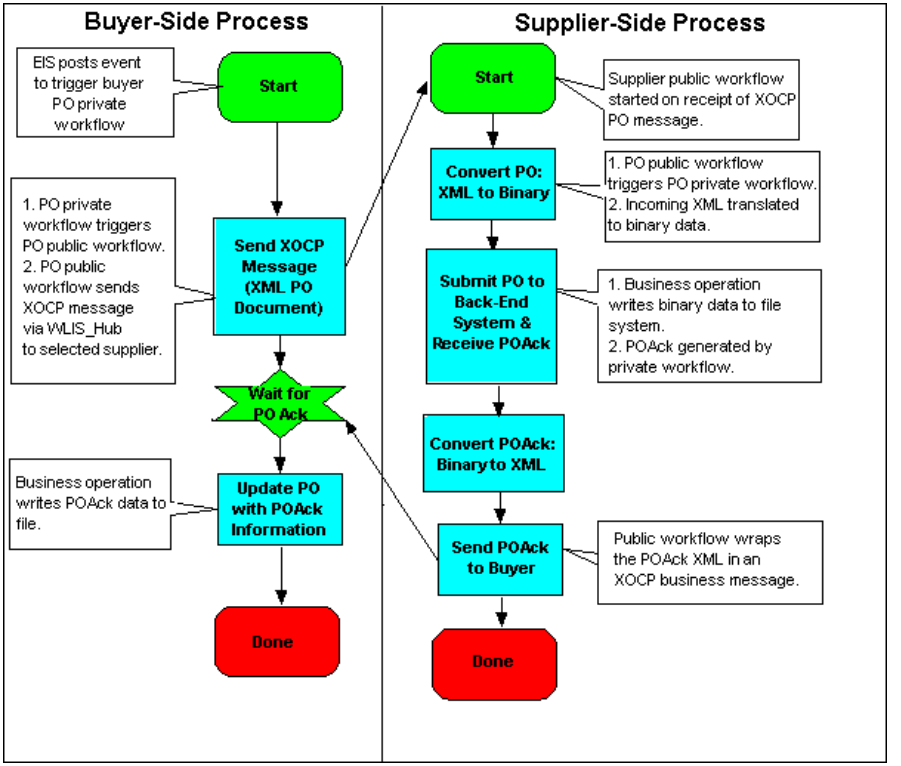
The Purchase Order for Review Portlet and PO Business Process

You may need to scroll down to see the Purchase Order for Review portlet (Step 5). It lists an order summary and purchase summary.

Technical Details for the PO Business Process

This section focuses on the Purchase Order business process, which will start once you (acting as an Avitek purchasing agent) click the Submit Purchase Order button on the Purchase Order for Review portlet. It lists an order summary and purchase summary.

Process Flow in the Purchase Order Business Process



The following sequence summarizes the events illustrated in the preceding figure:

1. The buyer composes a purchase order (PO) document.
2. The buyer sends the PO document to the selected supplier.
3. The supplier receives the PO and converts the XML-based PO document into a binary data file. (In this sample we assume the supplier's order management system expects to receive a binary file to process the order).
4. The supplier generates an XML-based PO acknowledgment document, based on another binary data file.
5. The supplier sends the PO acknowledgment document to the buyer.

6. The buyer updates the PO information, based on the PO acknowledgment.

Note: The preceding figure shows a high-level view of the PO business process. Each side of the process is implemented by a public and a private workflow.

The workflow templates listed in the following table are used in this sample's QPA process.

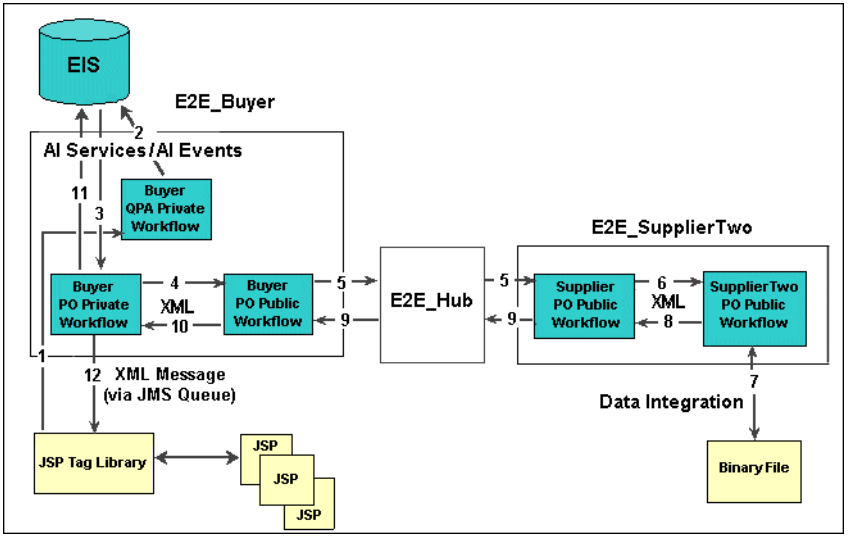
Table 3-3 QPA Process Workflow Templates

Role	Public/Private	Workflow Name
Buyer	Private	`E2E_BuyerQPAPrivate`
Buyer	Private	`E2E_BuyerPOPrivate`
Buyer	Public	`E2E_BuyerPOPublic`
Supplier	Public	`E2E_SupplierPOPublic` **Note:** Both suppliers in the scenario use the same public workflow.
Supplier	Private	`E2E_SupplierOnePOPrivate`
Supplier	Private	`E2E_SupplierTwoPOPrivate`

The PO implementation for this sample requires WebLogic Integration support for application integration, data integration, and management of business processes. This section describes the PO workflows, including their integration with back-end applications and heterogeneous data formats.

The following figure illustrates the data flow among the trading partners involved in the PO business process.

Data Flow in the PO Business Process



The following sequence of events summarizes the data flow among trading partners, workflows, and back-end systems:

1. The PO business process will start when you, as the buyer in this scenario, click the Submit Purchase Order back on the Purchase Order for Review portlet.

   This PO, in the form of an XML message, is posted to the BPM JMS queue.

2. The buyer's private workflow (`E2E_BuyerQPAPrivate`) that subscribes to the XML event is invoked by the event posting.

   The `E2E_BuyerQPAPrivate` workflow invokes the insertPO service on the `E2EAppView.sav` application view to insert the PO information into the Enterprise Information System (EIS). The EIS is simulated by an RDBMS in this sample. (The `E2EAppView.sav` application view, together with its services and events, is configured and deployed in WebLogic Integration when you set up the sample.)

3. The EIS sends an event containing the PO information to the WebLogic Integration process engine.

   The application view event triggers the start of the buyer's private workflow for the PO process (`E2E_BuyerPOPrivate`).

4. The E2E_BuyerPOPrivate workflow starts the buyer's public workflow (E2E_BuyerPOPublic).

5. E2E_BuyerPOPublic sends an XOCP message to the suppliers' public workflow (E2E_SupplierPOPublic). This step initiates the PO conversation.

   The E2E_Hub trading partner routes the message to the destination trading partner, E2E_Buyer, based on a registered collaboration agreement between itself and E2E_Buyer.

   Note: In this step, the E2E_Hub trading partner changes roles and becomes the proxy supplier in the collaboration agreement between itself and the buyer.

6. On the supplier side, an instance of the suppliers' public workflow (E2E_SupplierPOPublic) is started when the supplier receives the XOCP message.

   The E2E_SupplierPOPublic workflow starts the selected supplier's private workflow (E2E_SupplierOnePOPrivate or E2E_SupplierTwoPOPrivate).

7. The selected supplier's private workflow:

   - Translates the XML PO data it receives to binary data, using the data integration functionality of WebLogic Integration.
   - Generates a PO acknowledgment in binary format.
   - Translates the PO acknowledgment from binary format to XML.

8. The supplier's private workflow returns the PO acknowledgment XML data to the E2E_SupplierPOPublic workflow.

9. The E2E_SupplierPOPublic workflow wraps the PO acknowledgment information in an XOCP message and sends it to the buyer's public workflow (E2E_BuyerPOPublic).

10. The E2E_BuyerPOPublic workflow receives the XOCP message, extracts the XML content, and sends an XML event to the buyer's private workflow (E2E_BuyerPOPrivate).

    This step marks the end of the PO conversation.

11. The E2E_BuyerPOPrivate workflow uses the application integration framework provided by WebLogic Integration to update the PO information in the EIS, based on the data provided in the PO acknowledgment. The workflow also writes PO acknowledgment information to the POAcknowledgement.xml file.

12. WebLogic Portal can then read the `POAcknowledgement.xml` file and displays the contents in the appropriate portlets, such as the Order History portlet.

    This step marks the end of the PO business process.

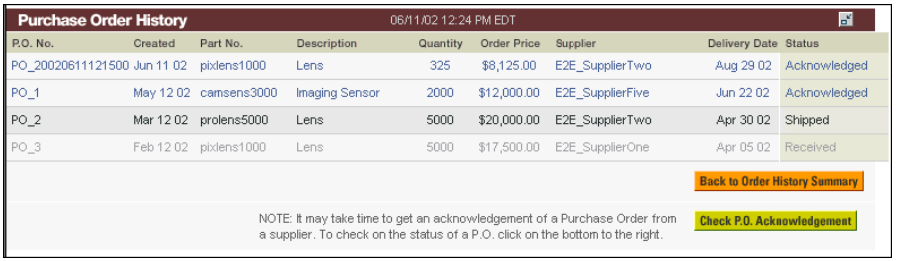
For more detailed information about this process, see the WebLogic Integration documentation.

Next Step

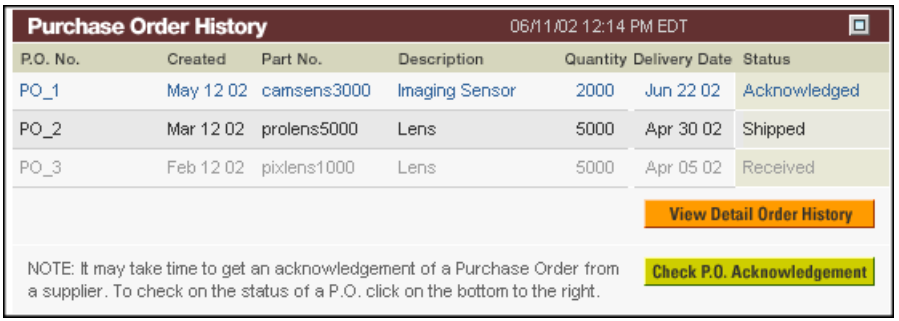
To continue the tour, click the Submit Purchase Order button to send the purchase order to the supplier.

The Purchase Order History Portlet

The Order History page initially presents a summary view of the Purchase Order History portlet. The sample application's data includes a few existing purchase orders. A more detailed view of the same portlet is available by clicking the View Detail Order History button.



You can click the Check P.O. Acknowledgement button to check for acknowledgements back from suppliers on the purchase order. Initially your screen might look similar to the following:



Purchase Order History 06/11/02 12:14 PM EDT

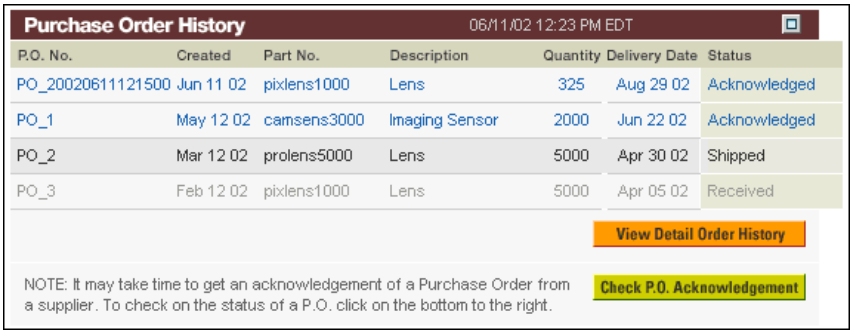
P.O. No.	Created	Part No.	Description	Quantity	Delivery Date	Status
PO_1	May 12 02	camsens3000	Imaging Sensor	2000	Jun 22 02	Acknowledged
PO_2	Mar 12 02	prolens5000	Lens	5000	Apr 30 02	Shipped
PO_3	Feb 12 02	pixlens1000	Lens	5000	Apr 05 02	Received

View Detail Order History

NOTE: It may take time to get an acknowledgement of a Purchase Order from a supplier. To check on the status of a P.O. click on the bottom to the right.

Check P.O. Acknowledgement

After clicking the Check P.O. Acknowledgement button a few minutes later, the following screen shows an updated view with our purchase order and its delivery date of August 29, 2002.



Purchase Order History 06/11/02 12:23 PM EDT

P.O. No.	Created	Part No.	Description	Quantity	Delivery Date	Status
PO_20020611121500	Jun 11 02	pixlens1000	Lens	325	Aug 29 02	Acknowledged
PO_1	May 12 02	camsens3000	Imaging Sensor	2000	Jun 22 02	Acknowledged
PO_2	Mar 12 02	prolens5000	Lens	5000	Apr 30 02	Shipped
PO_3	Feb 12 02	pixlens1000	Lens	5000	Apr 05 02	Received

View Detail Order History

NOTE: It may take time to get an acknowledgement of a Purchase Order from a supplier. To check on the status of a P.O. click on the bottom to the right.

Check P.O. Acknowledgement

Technical Details for the Purchase Order History Portlet

The Purchase Order History portlet on this Order History page displays the Acknowledged, Shipped, or Received status and related information for each purchase order (P.O.). A P.O. that you just submitted might not be shown in this portlet until the acknowledgement has been received back from the supplier and you clicked the Check P.O. Acknowledgement button.

If you just submitted a purchase order and it is not yet listed in the Purchase Order History portlet (its status is pending), click the Check P.O. Acknowledgement button.

Receiving the acknowledgement back from the supplier is, at a high level, the next to last step in the Purchase Order business process. The final step is for the buyer (Avitek) to update the P.O. information based on the P.O. acknowledgement. This Purchase Order business process is illustrated and described in the Technical Details section for the Purchase Order Review portlet.

On this Order History page, you can also click the View Detail Order History button to see more details about purchase orders and their status. On the detailed view of the portlet, you can click the Check P.O. Acknowledgement button to check the status. If it is still pending, the portlet returns the message: "No acknowledgement has been received for your purchase order."

On the detailed Purchase Order History portlet, click the Back to Order History Summary button to return to this view of the portlet.

Final Step for the B2B Portal Tour

This concludes the B2B tour. In the online tour, you can click the Logout button to return to the sample's Introduction page. Note that when you log out, any purchase orders you placed in this session are not stored in the database. That is, the inventory levels are reset to their original values when you log back in as Jason Tang.

CHAPTER

4 Web Services Tour

The sample application includes two Web services that we created in WebLogic Workshop. If you took the tour through the business-to-consumer (B2C) sample portal for Avitek Digital Imaging — section 02 from the sample's Introduction page — you probably saw the results returned by the Web services. One performs the product rating look-up that is displayed in the site's Product Evaluator portlet. The other Web service performs the credit card payment authorization.

This portion of the WebLogic Platform sample takes you behind the scenes to learn how BEA WebLogic Workshop was used during the development cycle to create the Web services. In addition, we show you how the Portlet Wizard can be used to reference the product rating Web service's WSDL and generate code for a portlet.

Note: The information that is presented in this section is also available by clicking the "BEGIN Web Services Tech Tour" button on the sample application's Introduction page.

This Web services tour contains the following sections:

Starting WebLogic Workshop

The first step is to start the WebLogic Workshop visual development environment. This section explains how to do that on supported Microsoft Windows and UNIX platforms.

For supported platforms information, please refer to the Supported Platforms page on the BEA e-docs Web site.

To start WebLogic Workshop from the Microsoft Windows Start menu, choose:

Programs →BEA WebLogic Platform 7.0 →WebLogic Workshop →WebLogic Workshop

To start WebLogic Workshop on UNIX, follow these steps:

1. Open a file system browser or shell.
2. Locate the Workshop.sh file in the BEA_HOME directory structure. For example:

   ```
   $HOME/bea/weblogic700/workshop/Workshop.sh
   ```

3. CD to the directory and type the following command:

   ```
   sh Workshop.sh
   ```

4. Or, from a file system browser, double-click Workshop.sh.

About the Project Locations

For this tour, we are going to look at two Web services that were defined in WebLogic Workshop for the WebLogic Platform Sample Application. The project files that comprise these Web services were installed into the following locations under your BEA_HOME directory:

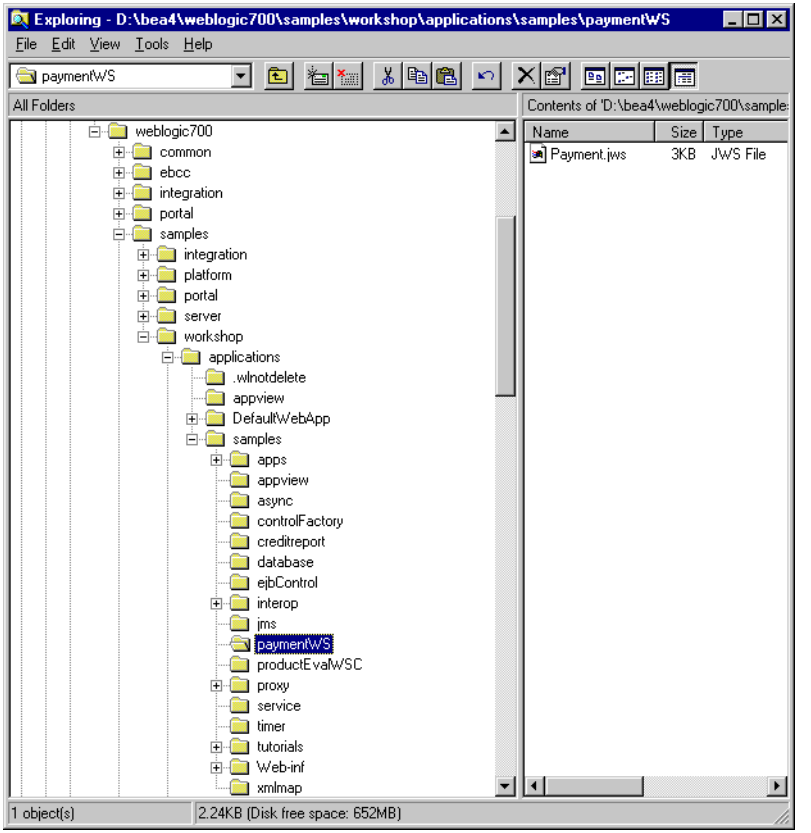
```
weblogic700\samples\platform\e2eDomain\beaApps\e2eWebServicesApp\workshop\productEvalWSC
```

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eWebServicesApp\workshop\paymentWS
```

The easiest way to view and test these Web services is to copy these two folders, shown above in **bold**, to the following folder:

```
weblogic700\samples\workshop\applications\samples
```

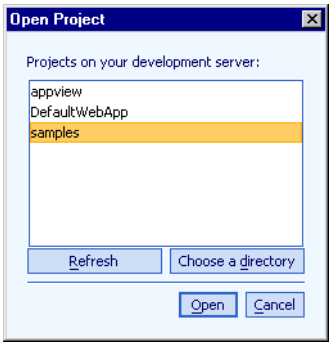
After the copy operation, the folder hierarchy is as follows:



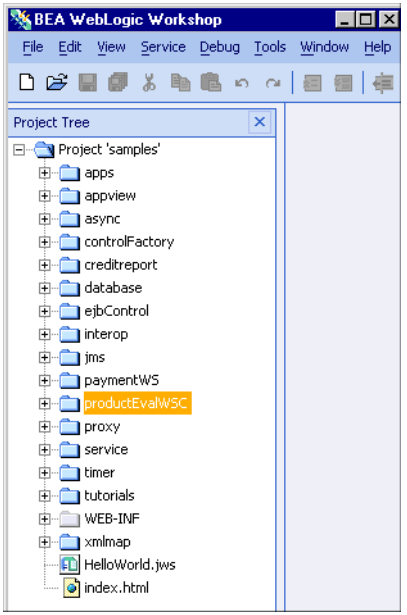
Then in WebLogic Workshop, use the Open Project menu option to navigate to the JWS file for each Web service. When you start WebLogic Workshop for the first time after installation, it prompts you to create a new project, or open a new one. We suggest that you:

1. Click File →Open Project from the top-level menu.
2. Select the **samples** project, as shown in the following figure.

3. Click the Open button.

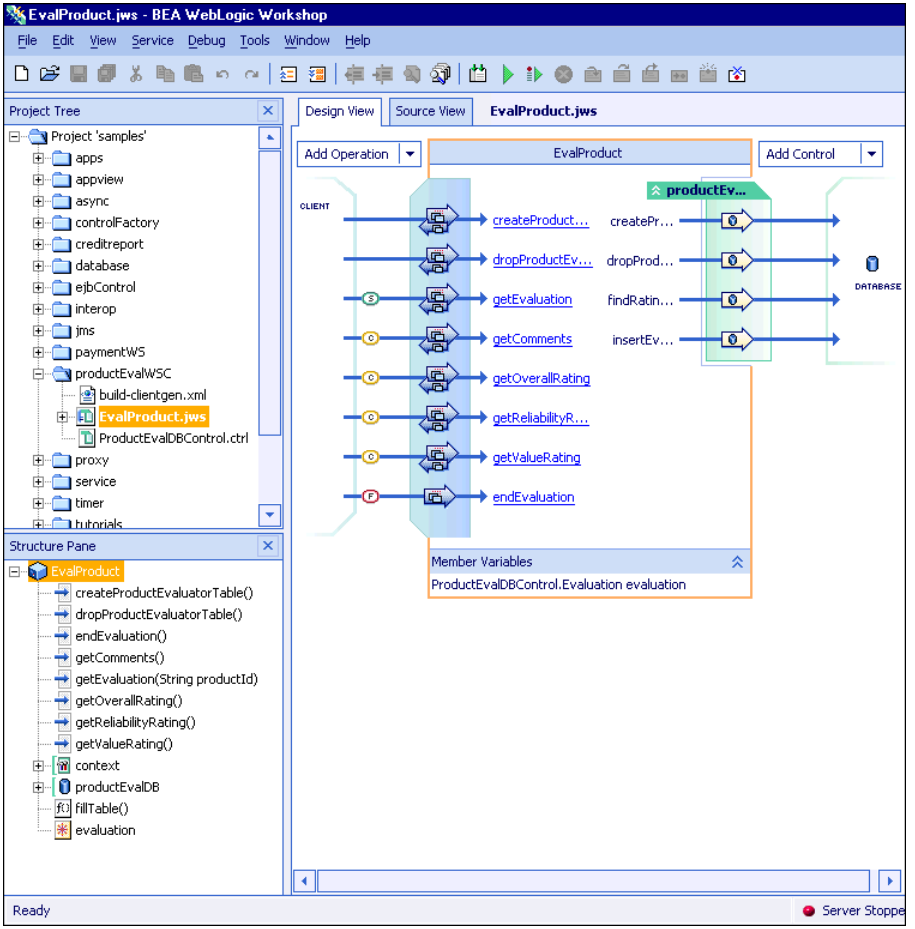


Because you copied the `productEvalWSC` and `paymentWS` folders into the WebLogic Workshop samples project's folder, you will see them in the project tree, as shown in the following figure.



Note: Each WebLogic Workshop project corresponds to a J2EE Web application, as evidenced by the `WEB-INF` folder in the samples folder.

For example, you can expand the `productEvalWSC` folder in the project tree and then double-click the `EvalProduct.jws` file. The WebLogic Workshop visual development environment displays a screen that is similar to the following:



Notice that a WebLogic Server instance is not running. See the next section for information about server startup options.

Server Startup Options

If you are interested in simply viewing the definitions of the Web services, you will not need to start a WebLogic Server instance in the WebLogic Workshop visual development environment. However, if you want to run and test the Web services as part of this tour, follow these steps. Although a server instance is already running for the `e2eDomain`, in this tour we will start a separate server instance to run and test the Web services located under:

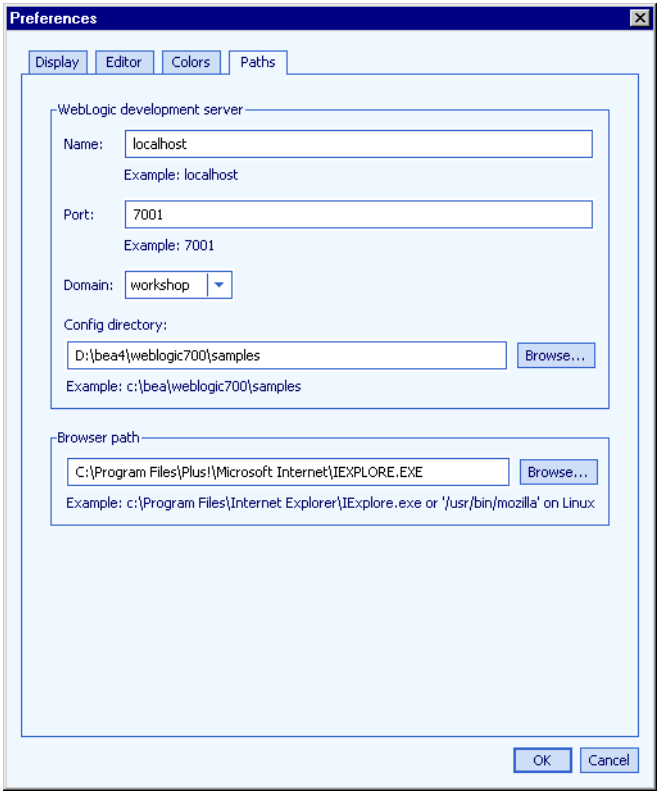
```
weblogic700\samples\workshop\applications\samples
```

This assumes that you have already copied the `paymentWS` and `productEvalWSC` folders from the Platform samples area to the Workshop samples area, as described in the previous section, "About the Project Locations" on page 4-2.

To run and test the Web services, follow these steps:

1. From the WebLogic Workshop visual development environment's top-level menu, click Tools →Preferences.
2. Click the Paths tab.
3. In the information about the server instance to be started, the default information should be correct. Confirm that the domain is "Workshop".
4. Check that the browser path is correct. If it is not correct, browse to the correct location for your browser's executable.

   The following screen shows sample values.



5. On the Preferences screen, click OK.

6. From the top-level menu, click Tools →Start WebLogic Server.

When the server startup has completed, WebLogic Workshop displays a Server Running status near the bottom of the screen.

Note: You can test already deployed Web services that were created in WebLogic Workshop in a domain other than `cgDomain`. For example, the `e2eDomain` is installed with the `paymentWS` and `productEvalWSC` Web services deployed under:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eWebServicesApp\workshop\*
```

With the server instance for the `e2eDomain` running, you could test these Web services, without having to open the WebLogic Workshop visual development environment, by pointing your browser to:

```
http://localhost:7501/workshop/productEvalWSC/EvalProduct.jws
http://localhost:7501/workshop/paymentWS/Payment.jws
```

Whether you invoke the test pages from within WebLogic Workshop or not, the product provides these very useful pages to test your Web services.

The next section presents an overview of WebLogic Workshop features and outlines the process for defining a Web service.

Defining Web Services with WebLogic Workshop — An Overview

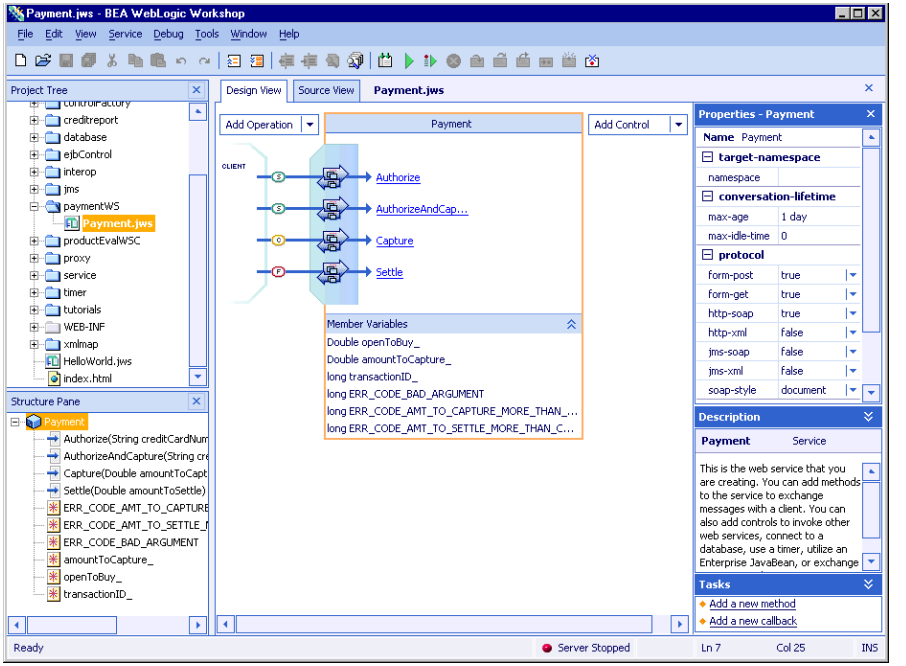
Before we examine the Product Evaluator and Payment Web services that were created for the WebLogic Platform sample application, let's look at the basic steps for defining Web services in WebLogic Workshop. We will cover some introductory topics for this exciting new product, and also provide links to its online documentation for more details.

WebLogic Workshop is a new visual development environment that makes it easy for application developers and J2EE experts alike to build and deploy enterprise-class Web services. The product is comprised of two major components:

- A design-time tool that lets developers write Java code to implement Web services.
- A run-time framework that provides the Web services infrastructure, testing, debugging, and deployment environment for applications.

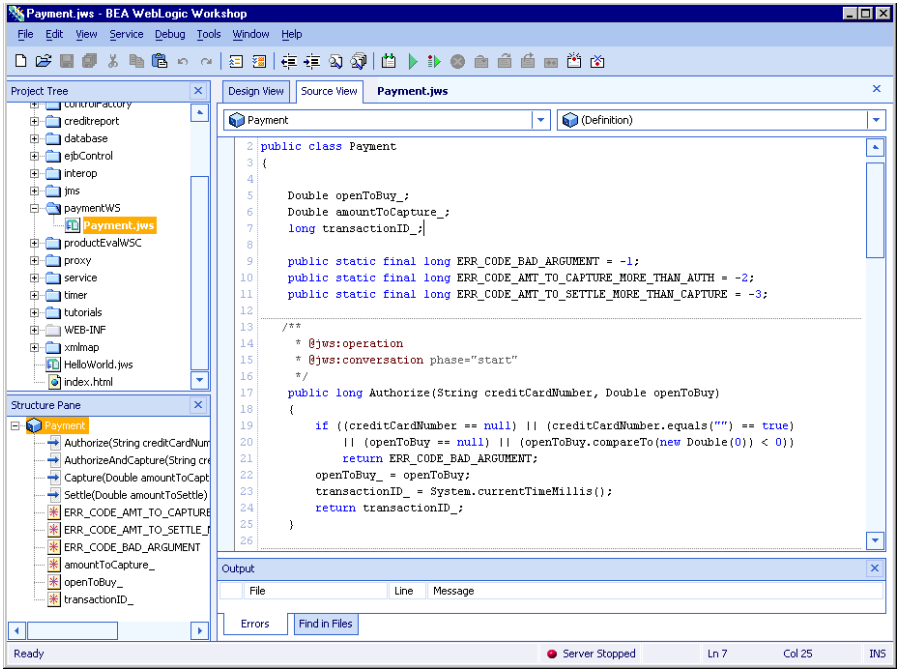
The WebLogic Workshop Visual Development Environment

The WebLogic Workshop visual development environment provides a complete environment for developing a Web service application. Standard features such as project management, syntax highlighting, code completion, and integrated debugging are all included. Also, WebLogic Workshop provides a useful visual approach to Web services. For example, the following shows a sample **Design View** screen.



The Payment Web service under development appears at the center of the screen. The design view lets users graphically create new methods, set properties on controls, and specify the overall structure of a Web service and its relationship to the outside world. The goal is to enable developers to focus on writing business logic — the code that is executed in response to each incoming method — not the machinery of typical Java programming. WebLogic Workshop supports two-way editing so any changes made through the graphical environment are reflected immediately in code, and vice versa.

The WebLogic Workshop Source View screen provides all the standard Java IDE editing features and exposes the Java Web Service (JWS) annotations. For example:



WebLogic Workshop Controls

WebLogic Workshop controls are a key innovation that ease the use of enterprise resources and J2EE APIs. WebLogic Workshop simplifies the complexity of these APIs and reduces the amount of object-oriented programming needed to access external resources. For example, WebLogic Workshop has a database control that simplifies the JDBC API. Database administrators can create a reusable database control that links SQL statements with Java methods in a simple declarative fashion. Developers can then use these components to access database resources with a simple function call.

For example, for the `productEvalWSC` Web service, we used WebLogic Workshop to define a database control named `ProductEvalDBControl`. One of the four methods in the database control is `findRatingData`:

```
/**
 * @jws:sql statement="SELECT RELIABILITY_RATING,
       VALUE_RATING, OVERALL_RATING,
       COMMENTS FROM E2E_PRODUCT_EVAL
       WHERE SKU = {productId}"
 */

public Evaluation findRatingData(String productId)
        throws SQLException, ControlException;
```

Notice how a special WebLogic Workshop Javadoc annotation associates an SQL statement with a Java method. Once a method is on a control this way, users of the control can simply call the function to execute the SQL command.

Java Web Service Files and Control Files

The meeting place between the design-time tool and the run-time framework is the Java Web Service (JWS) file and associated control (CTRL) files.

JWS files are standard Java files with annotations (using the Javadoc syntax) to express additional functionality. Annotations are used to display the Web service and its properties graphically. And the annotations are used by the framework to generate the EJB and J2EE code to execute the Web service. By moving code generation out of the tool and into the framework, developers never have to manage and maintain code that they didn't write.

Files with the extension CTRL are WebLogic Workshop controls. They typically include a collection of method definitions that allow you to easily access a resource such as a database or another Web service. CTRL files can represent the following types of controls:

- Service Control: used to communicate with another Web service from your service.
- Database Control: used to access a database from your Web service.
- EJB Control: used to access an existing Enterprise Java Bean (EJB) from your Web service.

- JMS Control: used to access an existing Java Message Service (JMS) queue or topic from your Web service.

The WebLogic Workshop Framework

Once a JWS file containing all the business logic for a Web service has been created, the WebLogic Workshop framework is responsible for generating the standard EJB code needed to implement it. The framework exposes, through annotations, functionality specifically designed to support building enterprise-class Web services. The WebLogic Workshop framework does the following:

- **Managing asynchronous communication with a conversational metaphor**

  The WebLogic Workshop framework automatically manages asynchronous message correlation and state management across messages in a conversation. What does that mean? Users can simply mark methods as starting, finishing, or continuing a conversation, and the framework takes care of the details. A unique ID is automatically generated to identify the conversation, and any state (class member variables) defined in the Web service is managed persistently with entity Java beans.

- **Enabling loose coupling with XML maps and XML scripts**

  The WebLogic Workshop framework uses simple, declarative XML maps to map between internal Java code and XML messages that are exchanged between Web services. Users indicate the structure of the desired method and associate XML fields with Java variables.

- **Enabling availability with JMS queues**

  To ensure availability under high load, Web services take advantage of message buffers. Users mark a method as requiring a buffer, and the WebLogic Workshop framework handles creating a queue and wiring it to the Web service. This feature enables one-click access to sophisticated J2EE functionality.

- **Supporting the control architecture**

  The WebLogic Workshop framework instantiates the implementation of any control used by the JWS. The framework also hooks up any asynchronous callbacks using a simple naming convention.

Outline of the Web Service Development Process

This section of the tour outlines the basic steps to building a Web service. You can find more information about each of these steps in the WebLogic Workshop online documentation. A good starting point is the tutorial. It is available within the visual development environment by clicking from the top-level menu: Help →Tutorial. Or you can read the tutorial by starting on the BEA e-docs Web site's pages for WebLogic Workshop. Look for the topic titled: "Tutorial: Your First Web Service."

1. **Start WebLogic Workshop and begin defining the methods, events, controls... then test what you have created so far.**

   - Start WebLogic Workshop.
   - For your first Web server, open the Samples project that comes with WebLogic Workshop.
   - In this project, create a subdirectory for the Web service. In the dialog box window, indicate that you are creating a Web service, and name it.
   - Start a WebLogic Server instance.
   - In the graphical Design View, add one or more methods and events for your Web service. You can add controls to represent resources such as other services, databases, and Enterprise Java Beans. You can also specify support for powerful features of the underlying server by setting properties for items in your design.
   - Select the Source View tab to see the results of your work done in Design View. You can add business logic. If you add a member variable to source code in Source View, your new variable will appear in Design View, and vice versa.
   - In Design View, add one or more parameters and return values for methods. Add code for returning a value.
   - Build, test, and deploy your Web service. Use the WebLogic Workshop Test View, a browser-based tool through which you can call the Web service's methods. (Sample screens are shown in the modules for the `paymentWS` and `productEvalWSC` Web services in this tour.)

2. **Add support for asynchronous communication, if needed by the application.**

   In asynchronous communication, the client communicates with the Web service in such a way that it is not forced to halt its processes while the Web service

produces a response. You also add a callback to be informed, for example, when a stock price reaches a particular value.

3. **Add a database control.**

   In this step you add a database control to your Web service. The database control provides your Web service access to a database. Controls act as interfaces between your Web service and other data resources, such as databases, other Web services, Java Message Services, and so on.

4. **Add a service control, if needed by the application.**

   You can add a service control to your Web service that enables it to invoke another external Web service.

5. **Add other controls, such as JMS and EJB controls, as needed.**

   You can access components that are available via the Java Message Service (JMS) using the `JMSControl`. Through this control, you can send and receive messages of various types, including XML. You also can access Enterprise Java Beans (EJBs) through the `EJBControl`. The `EJBControl` simplifies your use of an EJB by providing a single component representing the EJB's interface in your Web service design.

6. **Add a script file for mapping, as needed.**

   When you need to handle or control the specific shape of XML messages your Web service exchanges with other components, you can use XML maps. XML maps customize the WebLogic Server translation of XML to Java, and vice versa.

7. **Add Support for Cancellation and Exception Handling.**

   With the `TimerControl`, you can add timer functionality to your Web service. In this way, you can limit the time-specific operations that are allowed to execute, or force something to happen at regular intervals. You also can use the `onException` callback to handle exceptions thrown from your Web service's operations. When prompted by this callback, your code can perform any necessary clean-up and send a message to the client.

8. **Modify the Web Service to Support a Polling Interface, if needed.**

   As you build your Web service, WebLogic Workshop generates classes that can be used by client software. Through these proxy classes, a client can invoke your service's methods. You can download the proxy classes from Test View.

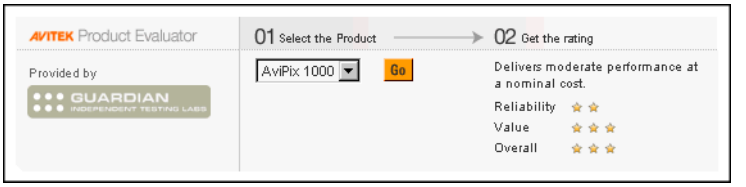
9. **Configure security settings and deploy the Web services as part of a Webapp.**

   You can make your Web service more secure by ensuring that it is exposed via the HTTPS protocol rather than HTTP. To do this, you edit the configuration file associated with your Web service's project. Using the `JwsCompile` command, you can then package your Web service for deployment to a production server.

This covers the basic steps. The next section will discuss the `productEvalWSC` Web service that we created in WebLogic Workshop for the WebLogic Platform sample application.

Defining the Product Evaluator Web Service

In the business-to-consumer (B2C) portal Web site for the fictitious company, Avitek Digital Imaging, we provided a Product Evaluator portlet for users who are browsing the product catalog:



This portlet used a Web service that we created in WebLogic Workshop to return product rating information about the selected catalog item. This section of the tour describes that Web service, `productEvalWSC`. This section also describes the steps taken in the Portlet Wizard to create the Product Evaluator portlet.

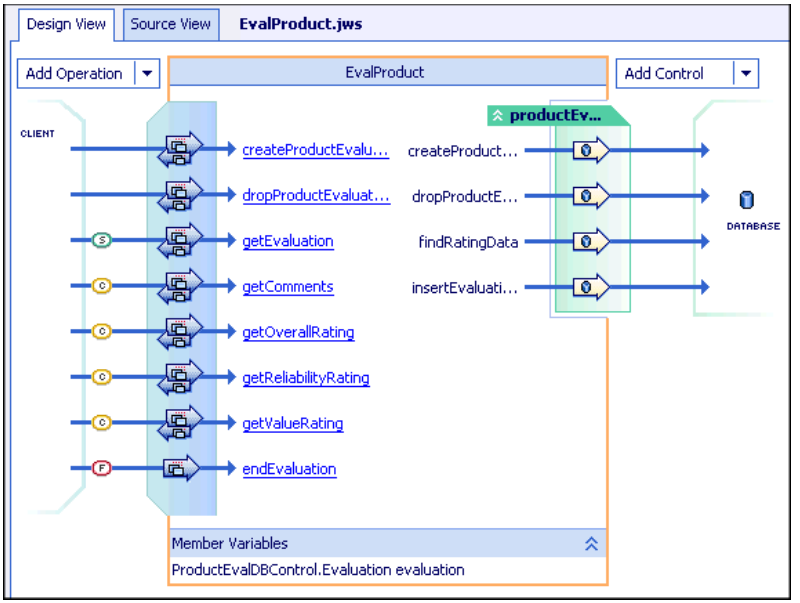
Viewing the Existing productEvalWSC Web Service

In the WebLogic Workshop project tree window, double-click on the `productEvalWSC` folder that you copied to:

```
weblogic700\samples\workshop\applications\samples\productEvalWSC
```

If you skipped this step in a prior tour page, see the section "About the Project Locations" on page 4-2. Then return to this page.

In the expanded project folder for `productEvalWSC`, double-click on the Java Web Service file named `EvalProduct.jws`. The following screen shows the Design View display for this Web service.



We used WebLogic Workshop to define the `ProductEvalDBControl` and its methods:

- `createProductEvalTable`
- `dropProductEvalTable`
- `findRatingData`
- `insertEvaluationRecord`

We then used WebLogic Workshop to define the following methods for the JWS:

- `getEvaluation`
- `getComments`

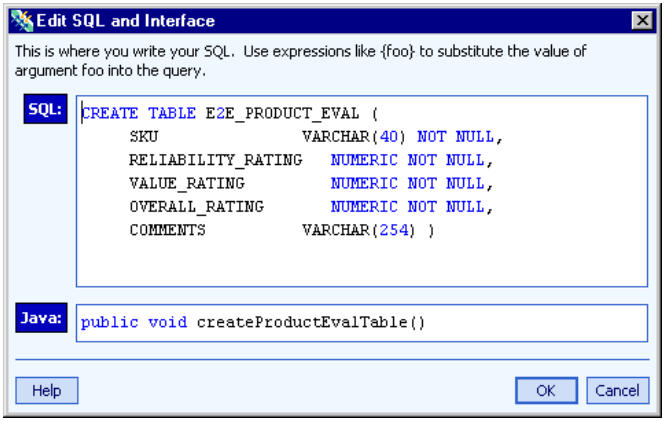
- `getOverallRating`
- `getReliabilityRating`
- `getValueRating`
- `endEvaluation`

In the Design View display, notice the small (s), (c), and (f) notations for the JWS methods. These notations indicate which method **starts** the Web service's conversation (`getEvaluation`), which method(s) continue the conversation (`getComments` through `getValueRating`), and which method **finishes** the conversation (`endEvaluation`).

Note: For a production Web service, we would not make separate calls to return the product evaluation's comments, overall rating, reliability rating, and value rating. Instead, we would create a complex object in the Web service and get all the ratings in a single call. Using a complex object and avoiding the separate calls should provide better performance.

Defining a database control and its methods in the WebLogic Workshop visual development environment is easy. This tour does not show the steps taken to create the control and methods. These steps are documented in the WebLogic Workshop documentation, which is available from within the visual development environment, or on the BEA e-docs Web site's pages for WebLogic Workshop.

However, here is a brief explanation about the purpose of the `ProductEvalDBControl` database control and its methods. During development, before the application's database schema is finalized for the enterprise application being placed into production, it is helpful to define database control methods that create or drop (as needed) the metadata, for testing purposes. As a convenience to our development process, we started by defining the `createProductEvaluatorTable` and `dropProductEvaluatorTable` methods. In WebLogic Workshop, you can double-click on the arrow for each method to see the SQL statements that we provided. For example, `createProductEvaluatorTable` defines:



We then defined the `insertEvaluationRecord` and `findRatingData` methods (please see the SQL statements in the WebLogic Workshop visual development environment).

With those database control methods available, we defined the JWS methods that use the database control methods. For example, the `getEvaluation` method starts the Web service conversation and uses the `findRatingData` method from the database control. In WebLogic Workshop, you can view the code for the JWS by either selecting the Source View tab, or by clicking the `getEvaluation` method's underlined link, which brings up the Source View near the line that starts the selected method. For example:

```
* @jws:operation
* @jws:conversation phase="start"
*/

public String getEvaluation(String productId)
{
   try
   {
     evaluation = productEvalDB.findRatingData(productId);
        }
        catch (java.sql.SQLException e)
        {
            // will be interpreted as productId not found
        }
        if( evaluation != null )
        {
            return "SUCCESS";
        }
```

```
        else

    {
            return("NOT FOUND");
    }
}
```

You can read the code in WebLogic Workshop for the methods that we defined for `EvalProduct.jws`, the JWS for `productEvalWSC`. If you need more information about defining methods in the visual development environment, please see the WebLogic Workshop documentation. It is available from within the visual development environment, or on the BEA e-docs Web site's pages for WebLogic Workshop.

Building and Testing the productEvalWSC Web Service

If the WebLogic Server instance for the `cgDomain` is not already running, start it. If you skipped this step in a prior tour page, see the section "Server Startup Options" on page 4-6. Then return to this section.

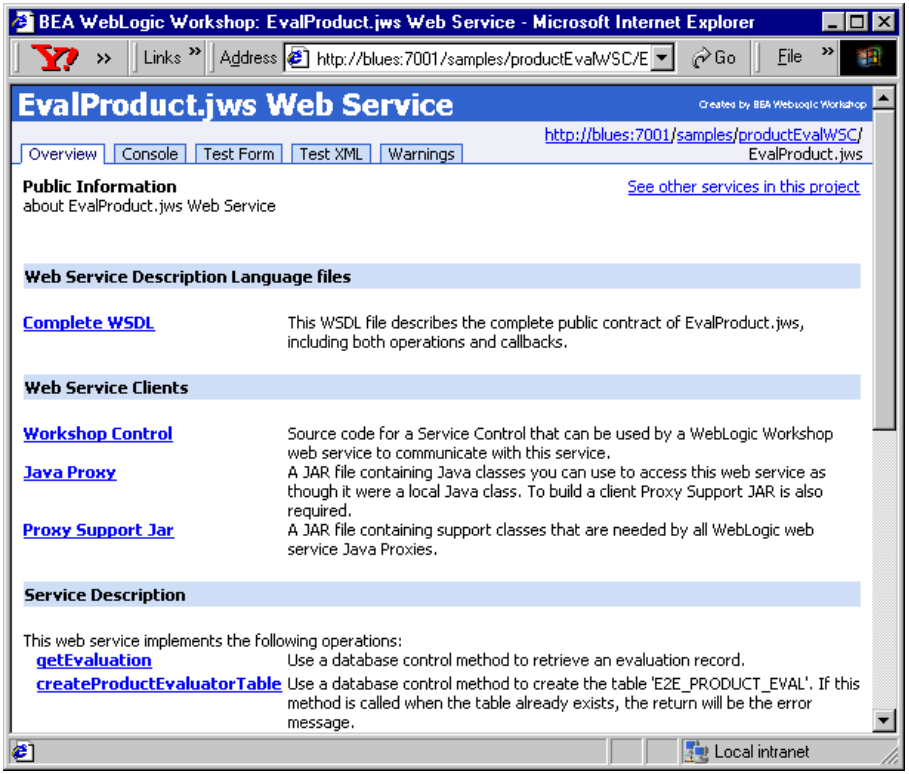
When the server is running, you will notice the following status icon near the bottom of the WebLogic Workshop visual development environment:

With the server running, click Debug →Build. When the build completes for this already defined Web service, WebLogic Workshop will have placed compiled JWS classes in the following location:

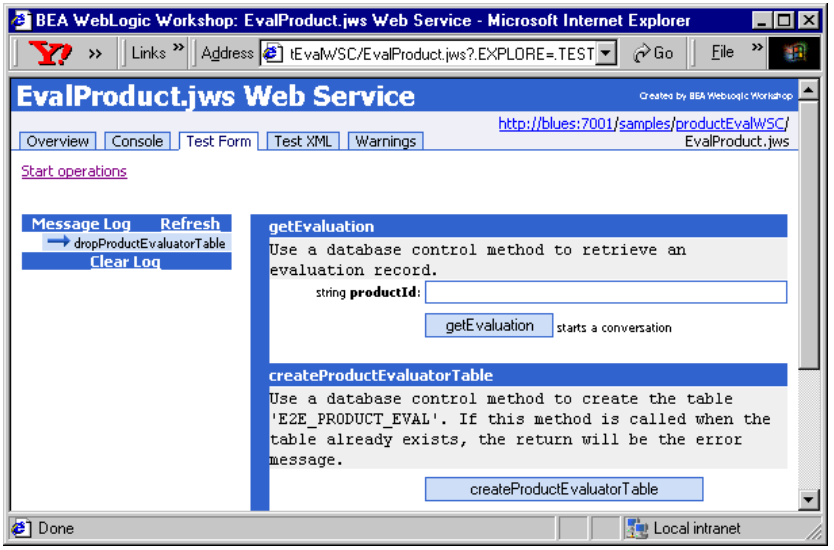
```
weblogic700\samples\workshop\cgServer\.jwscompile\_jwsdir_samples
 \classes\productEvalWSC\*.class
```

With the server running, click Debug →Start

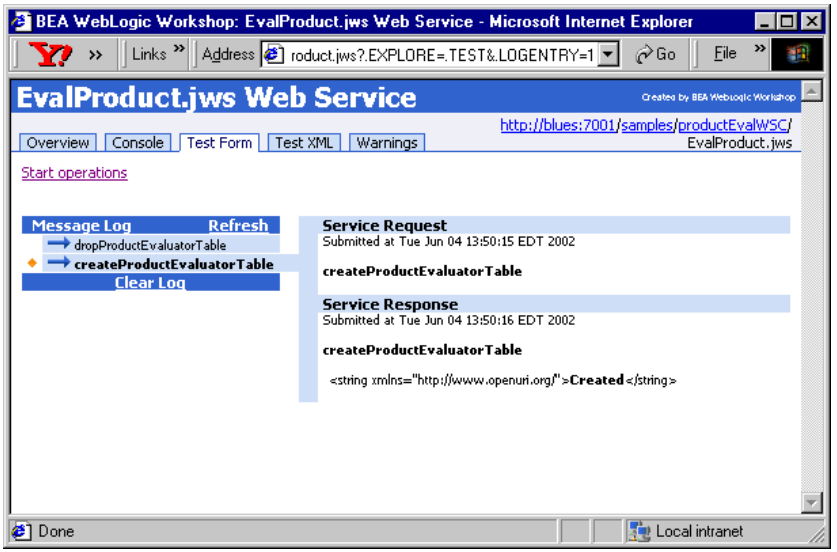
The test pages are browser-based. The initial tabbed page is the Test Form. For now, select the Overview tab. The following screen shows the Overview page with the server running on a remote machine named `blues`.



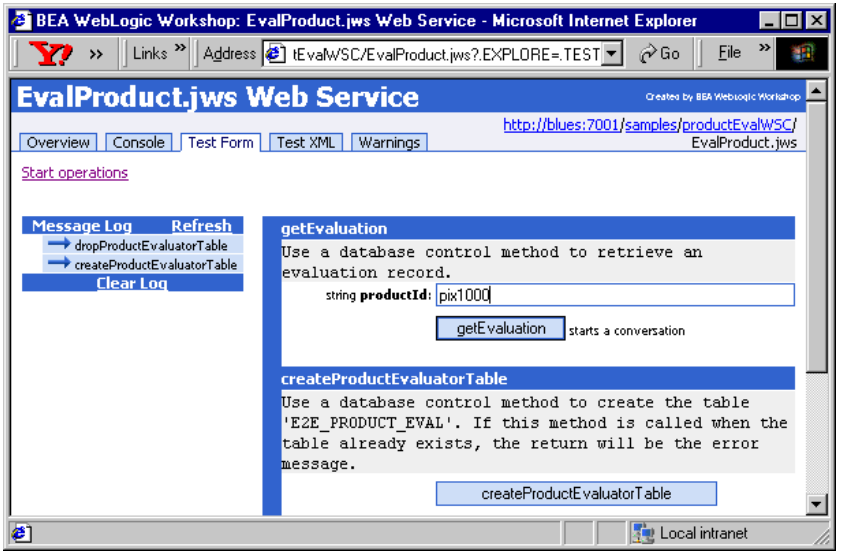
Select the Test Form tab. You can test the `productEvalWSC` Web service by entering sample data. Because you are testing the Web service in the `cgDomain` (the default server used with WebLogic Workshop), you must **first run the `CreateProductEvaluatorTable` method** on the following initial Test Form page, before trying the `getEvaluation` method. (We use the `CreateProductEvaluatorTable` method to conveniently create the database table and populate it, for testing purposes only while using the visual development environment. In production, the Web service will use a production database.)



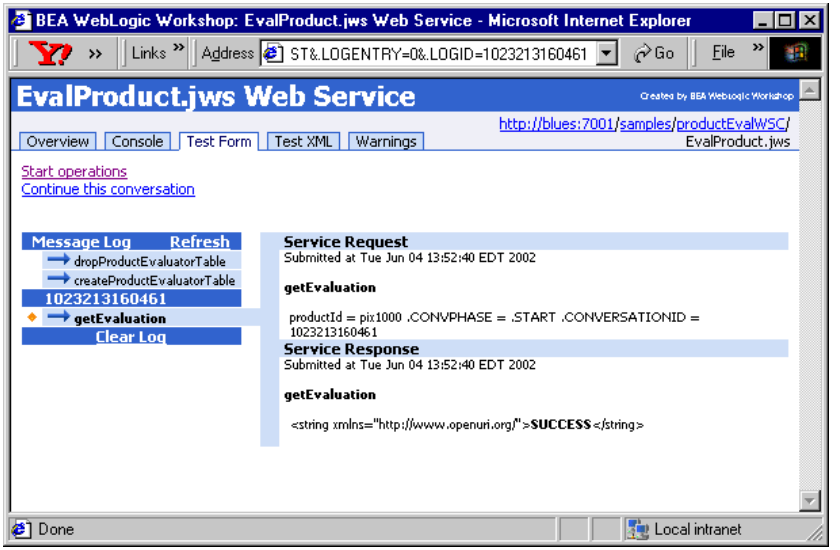
The Web service method should return a `Created` message. For example:



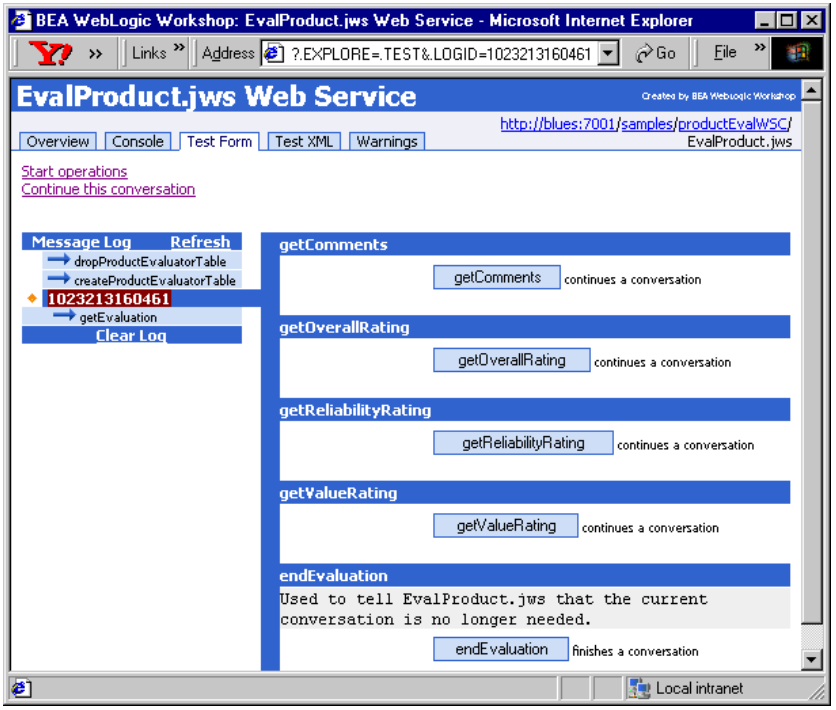
Next, click the Start operations link on the Test Form. On the refreshed page, start the test of the Web service's getEvaluation method. For example, enter **`pix1000`** in the `productId` field, as shown below. Then click the `getEvaluation` button.



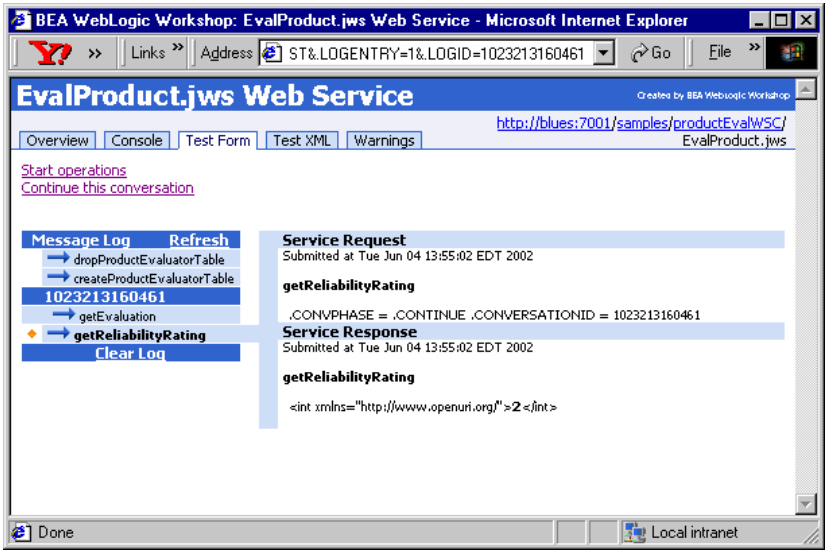
The Web service conversation is invoked, and the WebLogic Workshop run-time engine returns a **`SUCCESS`** status, as shown in the following screen.



To continue the test, click the **`Continue this conversation`** link on the page. WebLogic Workshop returns a list of subsequent methods defined for the `productEvalWSC` Web service as shown in the following screen.


BEA WebLogic Workshop: EvalProduct.jws Web Service - Microsoft Internet Explorer
Links
Address
?.EXPLORE=.TEST&.LOGID=1023213160461
Go
File
EvalProduct.jws Web Service
Created by BEA Weblogic Workshop
http://blues:7001/samples/productEvalWSC/EvalProduct.jws
Overview
Console
Test Form
Test XML
Warnings
Start operations
Continue this conversation
Message Log
Refresh
dropProductEvaluatorTable
createProductEvaluatorTable
1023213160461
getEvaluation
Clear Log
getComments
getComments
continues a conversation
getOverallRating
getOverallRating
continues a conversation
getReliabilityRating
getReliabilityRating
continues a conversation
getValueRating
getValueRating
continues a conversation
endEvaluation
Used to tell EvalProduct.jws that the current conversation is no longer needed.
endEvaluation
finishes a conversation
Local intranet

Click one of the methods to see the Web service return a product rating for this pix1000 camera. For example, when we clicked the `getReliabilityRating` method test, WebLogic Workshop returned:



Notice the Web service response is the integer **2**. On the Product Evaluator portlet, this results in displaying two out of five possible stars. For example:

On the Test Form, you can try other methods in the conversation. As noted earlier, for a production Web service we would not make separate calls to return the product evaluation's comments, overall rating, reliability rating, and value rating. Instead, we would create a complex object in the Web service and get all the ratings in a single call.

After we used WebLogic Workshop to create and test a Java Web Service (JWS) file for `productEvalWSC`, we used the Portlet Wizard to generate a draft version of the portlet (minus final presentation coding, graphics, Webflow, and packaging changes). The Portlet Wizard tool is part of the E-Business Control Center, a graphical tool

provided by WebLogic Portal. Portlet Wizard was used to generate the Web services interfaces code for the portlet. We also used WebLogic Server to generate the client proxy.

Note that when you first accessed the Web service, WebLogic Workshop automatically generated for you all the EJBs, with no action or work required by the developer. In our example, the first access resulted in the `productEvalWSC.EvalProductEJB.jar` file creation in:

```
weblogic700\samples\workshop\cgServer\.jwscompile\_jwsdir_samples
\EJB
```

The Web service class files were created in:

```
weblogic700\samples\workshop\cgServer\.jwscompile\_jwsdir_samples
\classes\productEvalWSC\*.class
```

Save the Web Service's WSDL

At any time in the development process, the Web Service Description Language (WSDL) file describing your Web service is available from WebLogic Server. WSDL is a standard XML document type controlled by the World Wide Web Consortium (W3C, see `http://www.w3.org` for more information).

WSDL files describe all the methods a Web service exposes (in the form of XML messages it can accept and send), as well as the protocols over which the Web service is available. The WSDL file provides all the information a client application needs to use the Web service.

There are several ways to obtain the WSDL file corresponding to a JWS file:

- In the visual development environment's Project tree, browse to the JWS file for which you would like to generate a WSDL file. In our example, browse to `EvalProduct.jws`. Right-click on the JWS file in the Project tree and select Generate WSDL from JWS. A file with the name `EvalProductContract.wsdl` will be created in the same directory. By default, the WSDL file is linked to the JWS file from which it was generated, meaning it will be regenerated whenever the JWS file is changed. For this reason, this is the recommended method of generating the WSDL.

- In a browser, browse to the URL of the Web service with `?WSDL` appended. For example, if the Web service is running on the default server on a machine named blues:

  ```
  http://blues:7001/samples/productEvalWSC/EvalProduct.jws?WSDL
  ```

  Use your browser's File →Save As function to save the WSDL file to your local machine. Note that some browsers will include HTML tags at the top and bottom of the saved file. You must remove these tags to produce a valid WSDL file. Be sure to designate the `.wsdl` file type; we recommend that you follow the naming convention of `<WebServiceName>Contract.wsdl`. For example, name it `EvalProductContract.wsdl`.

- In the WebLogic Workshop visual development environment, you can view the WSDL by clicking the Complete WSDL link on the Overview page while testing your Web service. You can then use the browser's Save As function to write the Web service's file into the same directory as the JWS file. (Again, follow the naming convention shown above, and remember that some browsers will include HTML tags that you must remove from the top and bottom of the saved file.)

Use Portlet Wizard to Generate Initial Code for the Product Evaluator Portlet

The E-Business Control Center (EBCC) is a Java client-based tool suite. It provides graphical interfaces that simplify complex tasks such as rule definitions, Webflow editing, and portal creation and management. As users of the E-Business Control Center work with its point-and-click interface, it generates XML files that are synchronized with the server.

One of the features in the EBCC is the Portlet Wizard. You can use it to reference WSDL for a Web service, and generate the code needed by a portlet. The following list outlines the general steps in the Portlet Wizard. When you work with the Portlet Wizard, be careful that you do not overwrite the existing evaluator portlet's files.

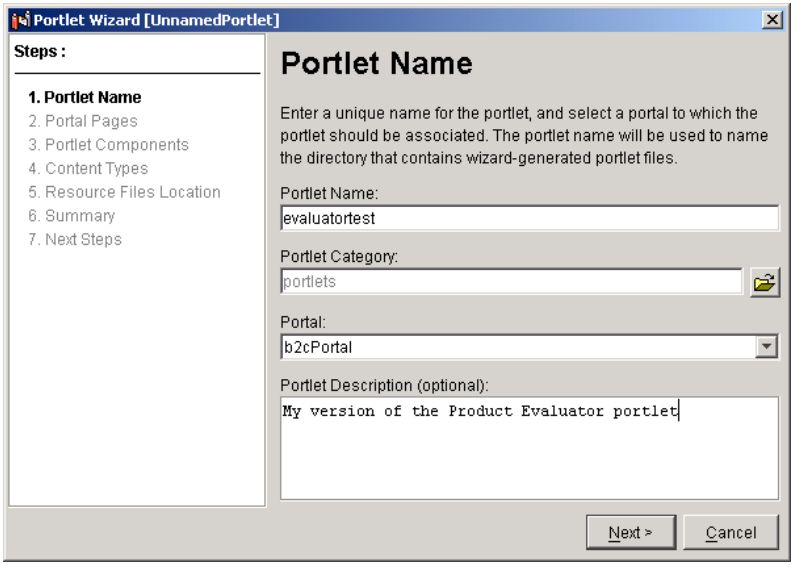
Note: The packaging and the variables used in the installed `evaluator.jsp` portlet are different from the content.jsp file created by Portlet Wizard in the following steps. The purpose of this section is to explain what could be created initially by Portlet Wizard.

To start the EBCC, for example on a Windows system, click the following in the Start menu:

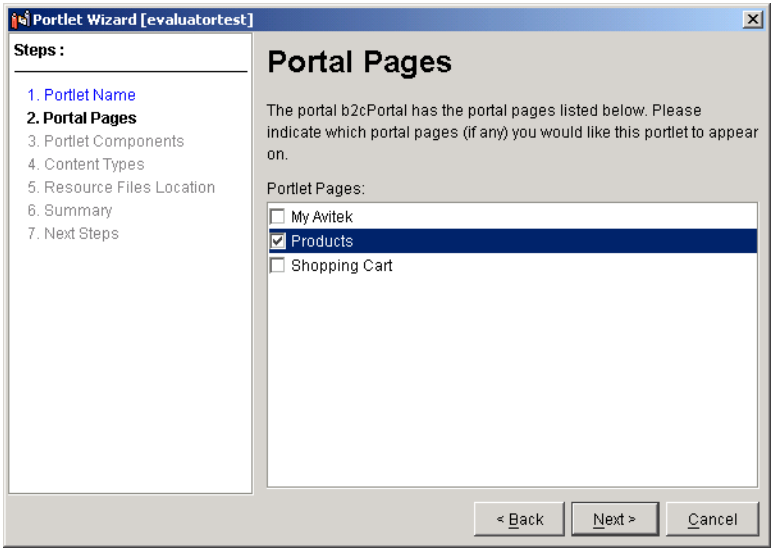
1. Programs →BEA WebLogic Platform 7.0 →WebLOgic Portal 7.0 →E-Business Control Center.
2. From the EBCC top-level menu, click File →Open Project. Navigate to the following folder under your BEA_HOME:

   `weblogic700\samples\platform\e2eDomain\beaApps\e2eApp-project`
3. Select the `e2eApp-project.eaprj` file in that folder, and click the Open button.
4. Then from the EBCC top-level menu, click File →New →Presentation →Portlet.
5. On the initial New Portlet screen, with the following option set, click the OK button:

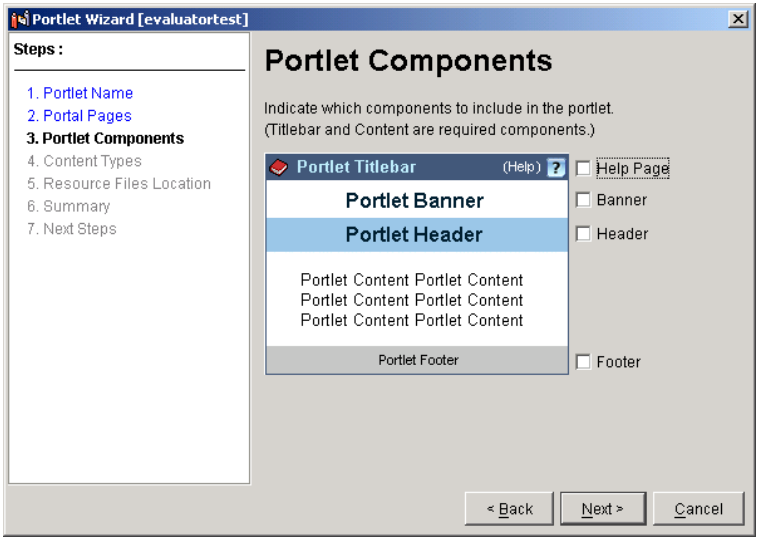
6. On the Portlet Name screen, give a unique name to the portlet, such as **evaluatortest**. In the Portal pull-down menu, select **b2cPortal**. Then click the Next button.



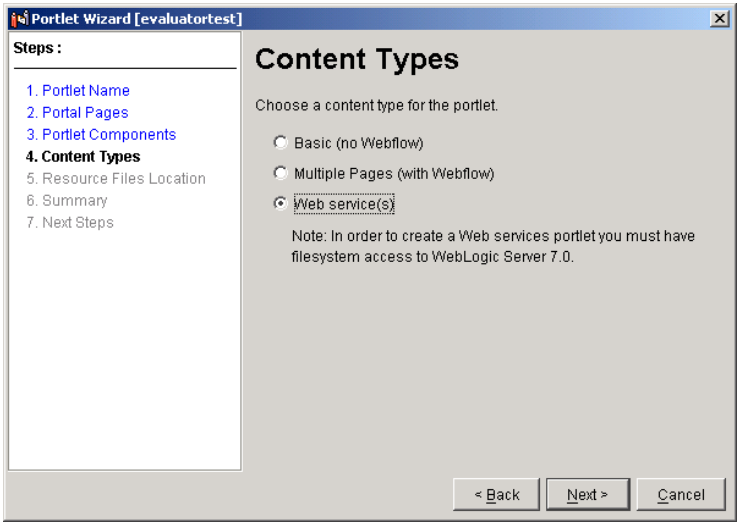
7. On the Portlet Pages screen, select the **Products** page from the list of available pages in the b2cPortal. Then click the Next button.



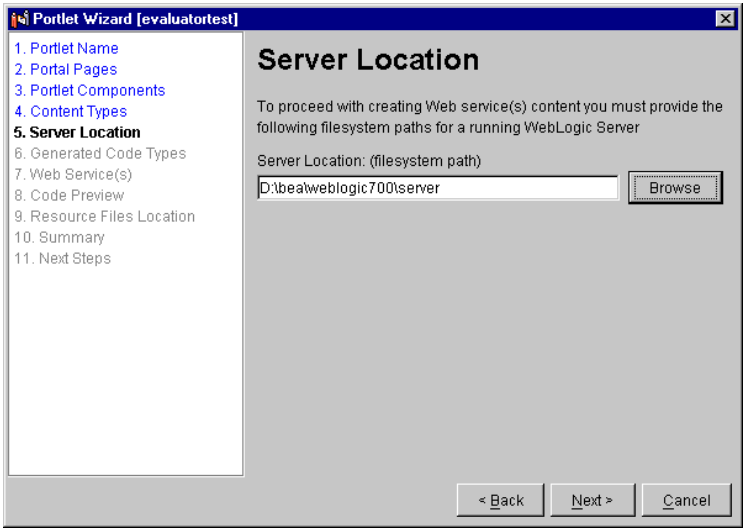
8. On the Portlet Components screen, select any components you want in your portlet, or leave these options unchecked. Then click the Next button.



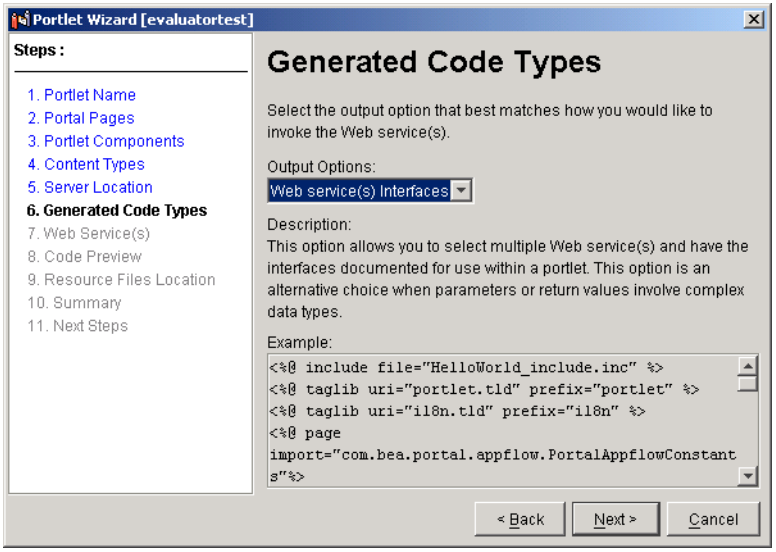
9. On the Content Types screen, select the **Web service(s)** option. Then click the Next button.



10. On the Server Location screen, indicate where WebLogic Server is installed. Then click the Next button.



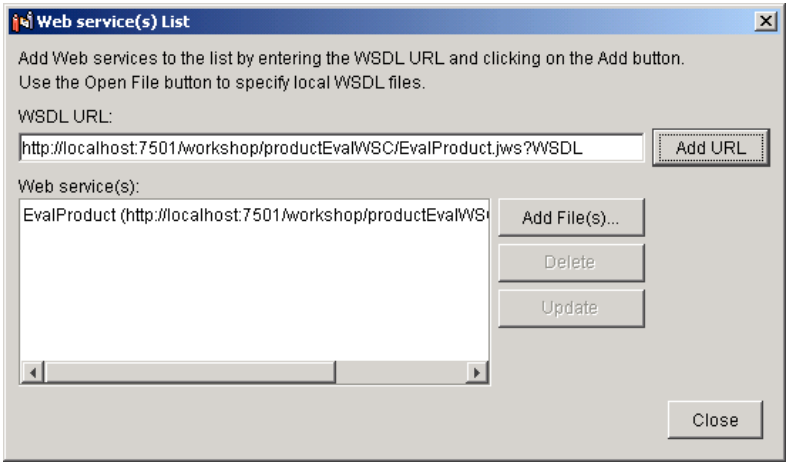
11. On the Generated Code Types screen, select **Web services(s) Interfaces** from the Output Options pull-down menu. The Product Evaluator portlet is conversational and requires the Web services(s) Interfaces options. The Form and Call Generation options are intended for simpler portlets. After making your selection, click the Next button.



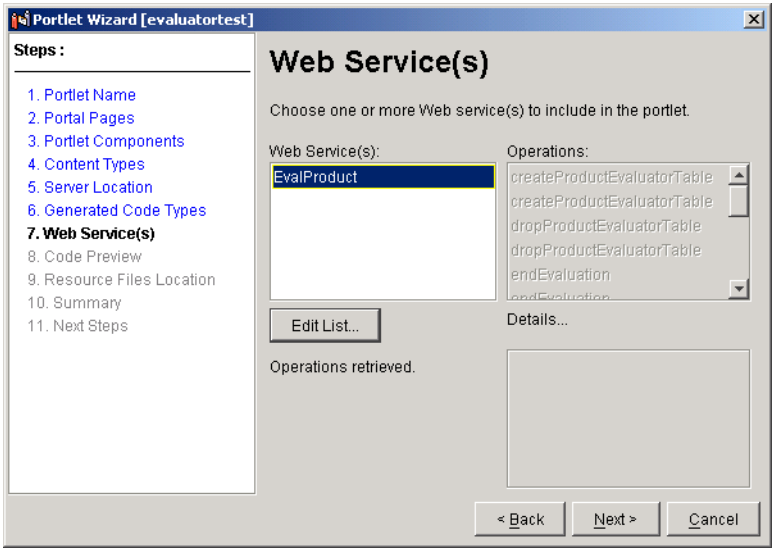
12. On the Web Service(s) screen, if you are using Portlet Wizard for the first time and there are no Web services in the current list, click the Add Web Services... button. If Web services are listed, but not the `EvalProduct` one created by WebLogic Workshop, click the Edit List button. Portlet Wizard displays the Web services(s) List screen. Enter the WSDL URL, such as:

    ```
    http://localhost:7501/workshop/productEvalWSC/EvalProduct.jws?WSDL
    ```

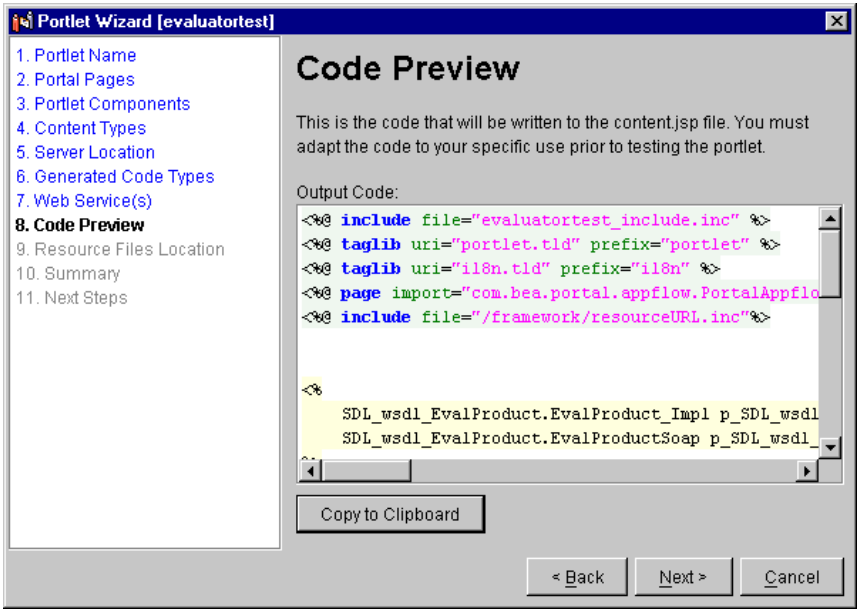
    Note that the URL is case sensitive. In the following example, we used the WSDL URL for the e2eDomain application (note: localhost:7501). After entering the WSDL URL in the input box, click the Add URL button. Portlet Wizard checks the URL and, if valid, adds it to the Web service(s) list. Then click the Close button on the Web service(s) List screen.



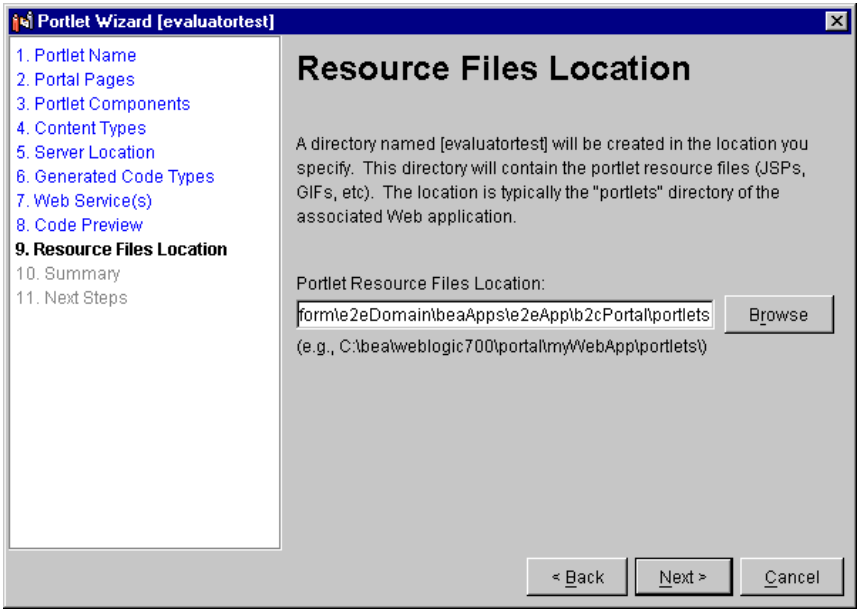
Back on the Web Service(s) screen, click the **EvalProduct** Web service in the list. Portlet Wizard connects with the WSDL URL and retrieves the operations we defined for the self-describing Web service. A progress screen is displayed during this step. After you see the **Operations retrieved** status message on the screen, click the Next button.



13. The Code Preview screen shows you what the Portlet Wizard will generate. You can, optionally, edit the code in the Portlet Wizard (which includes your changes, but does not validate them), or copy it to the clipboard for later pasting into your favorite editor. In either event, after previewing the Web service interfaces code that will be created for you, click the Next button.

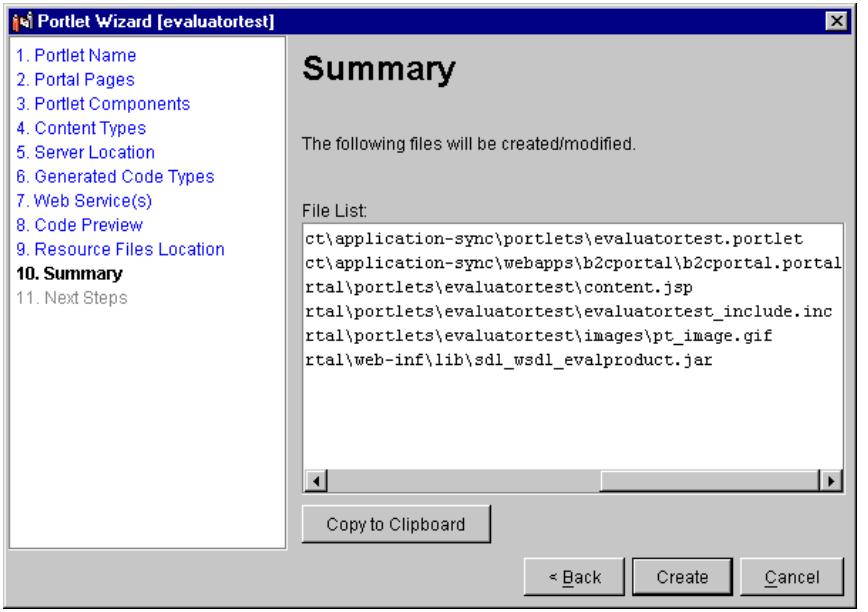


14. On the Resource Files Location screen, Portlet Wizard indicates that it will create a directory using the name of your portlet under the portlets directory of the associated Web application, as shown in the following example.



After indicating the location for the portlet JSP file that will be created, click the Next button.

15. On the Summary screen, Portlet Wizard lists the files that will be created or modified, as shown in the following screen.



In this example, the following files will be created under weblogic700\samples\platform\e2eDomain\beaApps:

- `\e2eApp-project\application-sync\evaluatortest.portlet`
- `\e2eApp\b2cPortal\portlets\evaluatortest\content.jsp`
- `\e2eApp\b2cPortal\portlets\evaluatortest\evaluatortest_include.jsp`
- `\e2eApp\b2cPortal\portlets\evaluatortest\images\pt_image.gif`
- `\e2eApp\b2cPortal\WEB-INF\lib\sdl_wsdl_evalproduct.jar`

In addition, Portlet Wizard will update the following XML file that lists the available portlets (but not yet enabled on a Products page) for the b2cPortal Web application:

- `\e2eApp-project\application-sync\webapps\b2cPortal\b2cPortal.portal`

If you want to proceed, click the Create button. Or click the Cancel button if you simply wanted to see the process in the Portlet Wizard.

Behind the scenes, Portlet Wizard executes the following build command to generate the Web service interfaces for you:

```
weblogic700\server\bin\ant.bat -f buildfile client-gen.xml
```

This command creates a JAR file containing .class, .WSDL, and other files that comprise the Web service's interfaces that can be used by the portlet. Also included in the JAR are the .java source files.

On the final Next Steps screen, you can uncheck the edit options and click the Close button if you are satisfied with the settings for the initial portlet. Note that you would need to make important enhancements to the portlet's JSP code and associated files before using the EBCC to synchronize the project (placing the portal definitions on the server) and before making the draft portlet visible and available on the Product page. The next section highlights the packaging differences. For information about EBCC server synchronization and making portlets available and visible to a portal page, see the *WebLogic Portal Administration Guide*.

Next Step

Now that you understand the general process of using the Portlet Wizard screens to generate the initial portlet's JSP and the interfaces (including `.class`, `.wsdl`, `.java` files in a JAR), we should note again: the packaging and the variables used in the installed Product Evaluator portlet and the `EvalProd` Web service are different from what you may have just created using Portlet Wizard. When we created the JAR file, we extracted the Java sources, significantly modified the portlet JSP (and added two include JSP files), and deployed the resulting files as follows. Note: the Java sources did not require any modifications; we extracted them simply to provide them with the installed sample directories.

Product Evaluator Portlet JSP files

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal
\portlets\evaluator\*.jsp
```

In the main `evaluator.jsp` portlet file and the included `step1.jsp` and `step2.jsp` files, we enhanced the presentation graphics that comprised the portlet, and added scriptlets to work with the Webflow for the portlet and the b2cPortal.

Web service client interfaces, CLASS files, used by WebLogic Workshop runtime

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\W
EB-INF\classes\productEvalWSC\*.class
```

Web service client interfaces, JAVA files

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\W
EB-INF\src\productEvalWSC\*.java
```

Server-side implementation

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eWebServicesApp\
workshop\productEvalWSC\*
```

These files include the deployed `EvalProduct.jws` and `ProductEvalDBControl.ctrl` copied from the WebLogic Workshop area and deployed as part of the workshop Web application, which was deployed as part of the `e2eWebServicesApp` enterprise application. Of course, the **`workshop`** Web application has a `WEB-INF` subdirectory that contains the configuration settings in `*.xml` files.

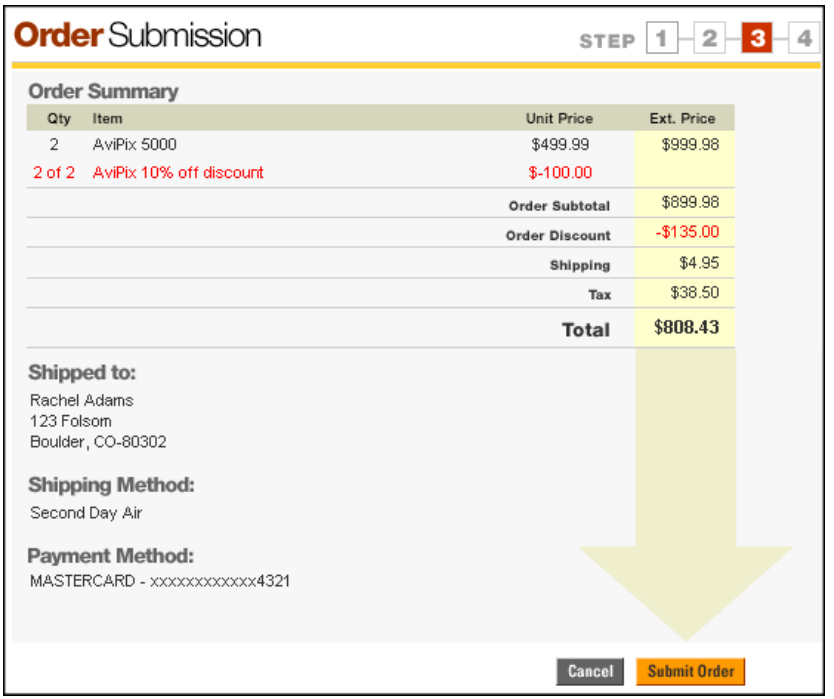
SOAP Conversation, JAVA files, generated by the client-gen task

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\b2cPortal\W
EB-INF\src\org\openuri\www\x2002\x04\soap\conversation
```

For information about the Payment Web service that is used in the b2cPortal, refer to the next section.

Defining the Payment Authorization Web Service

The B2C portal for Avitek Digital Imaging includes an Order Submission page as part of the checkout process.



When the user clicks the Submit Order button, the credit card payment authorization processing is handled by a pipeline component named `CajunBasedPaymentPC`, which calls a Java proxy that lets the pipeline component call the Payment Web service. We created the Web service in WebLogic Workshop.

The Payment Web service uses the conversational aspect of WebLogic Workshop framework. The first call is to authorize the credit card; we pass in the credit card number and the amount to be authorized as arguments. After the credit card authorization is complete, a call is made to capture an amount. Eventually a request is made to settle the amount.

The codes returned by the Payment Web service to the CajunBasedPaymentPC are as follows:

- -1 = Error (Bad argument passed)
- -2 = Error (When amount to capture is more than the amount the credit is authorized for)

- -3 = Error (When amount to settle is more than the amount for which the credit card was captured for)
- > 0 = Success

Note: The Payment Web service always sends payment information through without any errors, as if it were connected to and approved by a third-party payment service. The processing of payment via the Payment is not designed for production use. You must integrate with your third-party vendor's payment service to process payment correctly. Note, however, that the code shown in the sample pipeline component is set up to appropriately handle error conditions.

The Role of the CajunBasedPaymentPC Pipeline Component

Before we look at the definition of the Payment Web service in WebLogic Workshop, let's examine portions of the `CajunBasedPaymentPC.java` pipeline component. You can find the source file in the following directory under your `BEA_HOME`:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src\example
s\e2e\b2c\payment\pipeline
```

The sample pipeline component contains many import statements, including:

```
import paymentWS.Payment_Impl;
import paymentWS.PaymentSoap;
import examples.e2e.b2c.util.B2CPortalConstants;

import com.bea.p13n.util.debug.Debug;
import org.openuri.www.*;
import org.openuri.www.x2002.x04.soap.conversation.*;

import com.beasys.commerce.axiom.units.Money;
import com.beasys.commerce.ebusiness.shoppingcart.ShoppingCart;
```

These statements import packages for the Payment proxy, the generated SOAP interfaces, and a set of sample constant values (such as the payment type) to simplify the application. Also imported is a `ShoppingCart` EJB that processes shopping cart data for commerce applications.

Part of the `CajunBasedPaymentPC` pipeline component's work in the b2cPortal application is to get the current shopping cart data from the pipeline session.

```
CreditCard cc = (CreditCard)
getSessionAttribute(PipelineSessionConstants.PAYMENT_CREDIT_CAR
D, namespace, pipelineSession);

ShoppingCart sc = (ShoppingCart)
getSessionAttribute(PipelineSessionConstants.SHOPPING_CART,
namespace, pipelineSession);

Money amt = (Money) sc.getTotal();
```

We instantiate the proxy for the Web service:

```
try {
       proxy = new Payment_Impl(connectString +
          "/workshop/paymentWS/Payment.jws?WSDL");
       if (DEBUG.ON) {
           DEBUG.out("proxy instantiated");
       }
```

After setting up header objects in preparation for the conversation with the Web service, we start the conversation by first getting the credit card number and value (amount of the purchase) from the pipeline session. Then we pass authorize data to the Payment Web service, which returns a result. This entry point simply reserves credit on the supplied card for the amount specified.

```
Authorize auth = new Authorize(cc.getNumber(), amt.getValue());
    long result = proxySoap.authorize(auth, startHeader).getAuthorizeResult() ;
```

If no errors are returned in the authorization method, we continue with capture data. In payment transactions, the word "capture" refers to the amount of the credit card holder's remaining available credit. The total amount to settle must be less than the capture amount.

```
Capture capture = new Capture(amt.getValue());
result = proxySoap.capture(capture, continueHeader).getCaptureResult();
```

After getting the capture result from the Web service, we finish the conversation with the settlement data.

```
Settle settle = new Settle(amt.getValue());
result = proxySoap.settle(settle, continueHeader).getSettleResult();
```

See the `CajunBasedPaymentPC.java` source file for a closer look at the error handling. Again, the file is provided in:

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp\src
 \examples\e2e\b2c\payment\pipeline
```

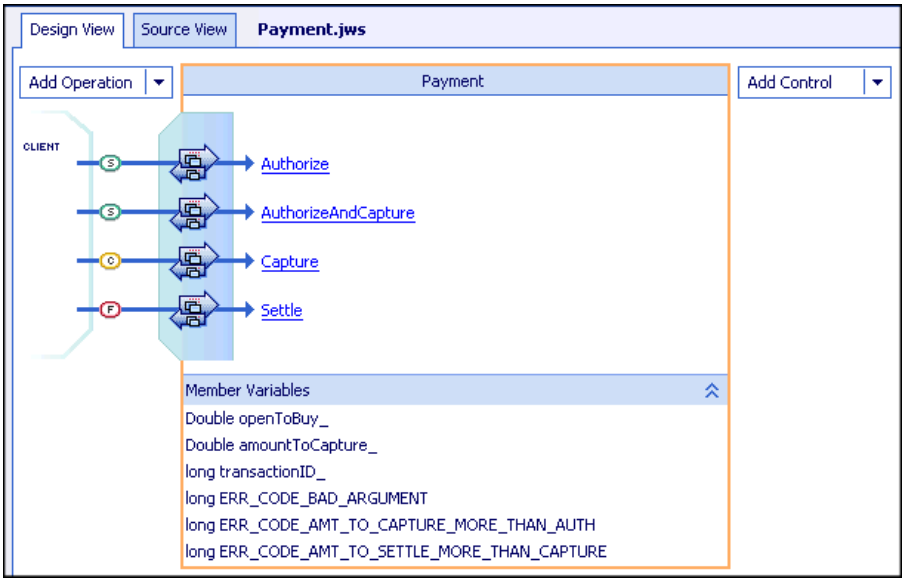
Viewing the Existing paymentWS Web Service

In the WebLogic Workshop project tree window, double-click on the `paymentWS` folder that you copied to:

`weblogic700\samples\workshop\applications\samples\paymentWS`

If you skipped this step in a prior tour page, see the section titled "About the Project Locations." Then return to this section.

In the expanded project folder for `paymentWS`, double-click on Java Web Service file named `Payment.jws`. The following screen shows the Design View display for this Web service:



We used the WebLogic Workshop to define the following methods for the Web service:

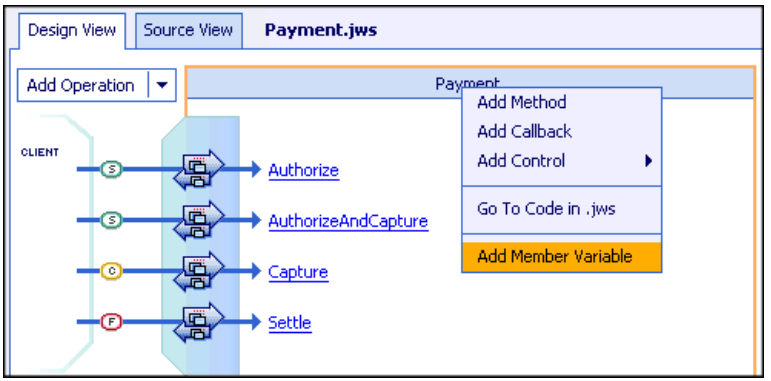
- `Authorize`
- `AuthorizeAndCapture`
- `Capture`
- `Settle`

In the Design View display, notice the small (s), (c), and (f) notations for the JWS methods. These notations indicate which methods can **start** the Web service's conversation (`Authorize` or `AuthorizeAndCapture`), which method(s) continue the conversation (`Capture`), and which method **finishes** the conversation (`Settle`).

If successful, the Web service methods return a unique `transactionID`, calculated in this sample as the time that the transaction occurred, in milliseconds. Returning an ID this way is typical for credit card authorization applications.

Defining methods in the WebLogic Workshop is simple. This tour does not show the steps taken to create the control and methods. These steps are documented in the WebLogic Workshop documentation, which is available from within the visual development environment, or on the BEA e-docs Web site's pages for WebLogic Workshop.

In the workshop, you can switch to Source View to see the definitions for the Web service's methods. After we added these four methods, and of course before we added the business logic you see in the visual development environment, we added **member variables** that we would need in the Web service. To do this, we right-clicked the Payment heading in the visual development environment and selected Add Member Variable from the pull-down menu:



The members variables defined for the Payment.jws are as follows. The underscore suffix is used to indicate instance variables.

- `openToBuy_`
- `amountToCapture_`
- `transactionID_`
- `ERR_CODE_BAD_ARGUMENT`

- ERR_CODE_AMT_TO_CAPTURE_MORE_THAN_AUTH
- ERR_CODE_AMT_TO_SETTLE_MORE_THAN_CAPTURE

Payment Models and the Methods Defined

In credit card transactions, there are two types of payment models: terminal-based and host-based. The difference between these payment models is where the transaction batch is stored. For a host-based model, the transaction batch is stored on the host network rather than on the local system at the merchant's site. Settlement typically occurs sometime at the end of the day, and the merchant is not required to do anything to initiate the settlement process.

For a terminal-based model, the transaction batch is stored as data files on the local system at the merchant's site. Merchants must initiate the settlement process at the end of each day in order for the funds to be transferred to the merchant's bank account.

The following list describes each of the terminal-based payment models that may be assigned by the financial institution.

- AUTO_MARK_AUTO_SETTLE

  This payment model is used for soft goods. Settlement occurs as soon as authorization is complete, because it is assumed that soft goods are shipped at the time of purchase.

- AUTO_MARK_MANUAL_SETTLE

  This payment model is used in cases where goods have been shipped at authorization but the merchant requests that funds should be transferred at a later date.

- MANUAL_MARK_AUTO_SETTLE

  This payment model allows merchants to indicate that the goods have been shipped, at which point settlement is done automatically.

- MANUAL_MARK_MANUAL_SETTLE

  This is the most flexible payment model in that it allows merchants to specify when goods are shipped and when funds should be transferred. The mark process allows the merchant to specify that the goods have been shipped. The settlement process allows the merchant to indicate that funds may be transferred.

This next list describes each of the **host-based payment models** that may be assigned:

- `HOST_AUTH_CAPTURE`

  This payment model is used for services, sale of digital goods, or physical goods shipped within 24 hours of when the order is placed. In this case, the merchant only needs to get an authorization for the purchase amount. The capture of the authorization into the batch and the settlement of the transaction are done for the merchant by the processor at the time of authorization.

- `HOST_POST_AUTH_CAPTURE`

  When the merchant fulfills orders more than one day after receiving them, the merchant must authorize and capture transactions separately. In this payment model, authorization is performed at the time the consumer wants to make the purchase. Capture is performed when the merchant ships the order. The processor handles settlement of the batched transactions at certain times of the day.

Again, the `Authorize` method is used for terminal-based payment models. This entry point validates the credit card number and reserves credit on the supplied card for the amount specified. When validated, it creates a new entry in a database table that records the incident and sets the state based on the payment model. The amount of the transaction is deducted from the `openToBuy` field in the customer's credit balance. However, the funds are not transferred to the merchant until settling.

Note: Merchants who are using a terminal-based processor must perform a capture and settlement procedure before the funds from the sale are transferred to their account. This is accomplished by a subsequent call to Capture and/or Settle, depending on the "Auto Mark/Auto Settle" processor configuration.

The `AuthorizeAndCapture` method is used for host-based payment models. This entry point validates the credit card number and reserves credit on the supplied card for the amount specified. When validated, it creates a new entry in a database table that records the incident and sets the state based on the payment model. The amount of the transaction is deducted from the `openToBuy` field in the customer's credit balance. However, the funds are not transferred to the merchant until settling.

Note: Merchants who are using a **host-based post-authorization capture processor** must perform a capture and settlement procedure before the funds from the sale are transferred to their account.

The `Capture` method in the Payment Web service determines the amount of the credit card holder's remaining available credit. The total amount to settle must be less than the capture amount, as checked in the Settle method:

```
* @jws:operation
* @jws:conversation phase="finish"
*/

public long Settle(Double amountToSettle)
{

if ((amountToSettle == null) ||
            (amountToSettle.compareTo(new Double(0)) < 0))
            return ERR_CODE_BAD_ARGUMENT;

if (amountToSettle.compareTo(amountToCapture_) > 0)
            return ERR_CODE_AMT_TO_SETTLE_MORE_THAN_CAPTURE;

return transactionID_;

}
```

If you need more information about defining methods in the visual development environment, please see the WebLogic Workshop documentation. It is available from within the visual development environment, or on the BEA e-docs Web site's pages for WebLogic Workshop.

Building and Testing the Payment Web Service

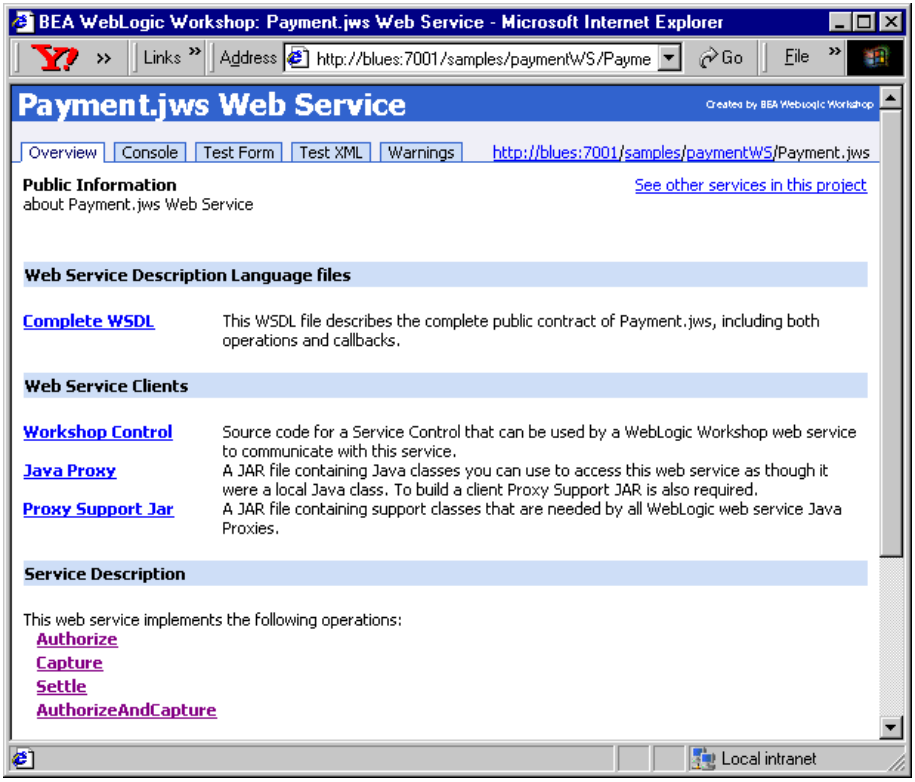
If the WebLogic Server instance for the `cgDomain` is not already running, start it. If you skipped this step in a prior tour page, see the section "Server Startup Options" on page 4-6. Then return to this section.

With the server running, click Debug →Build. Notice the Build Started message in the lower left corner of the visual development environment's screen. When the build completes for this already defined Web service, WebLogic Workshop will have placed compiled JWS classes in the following location:

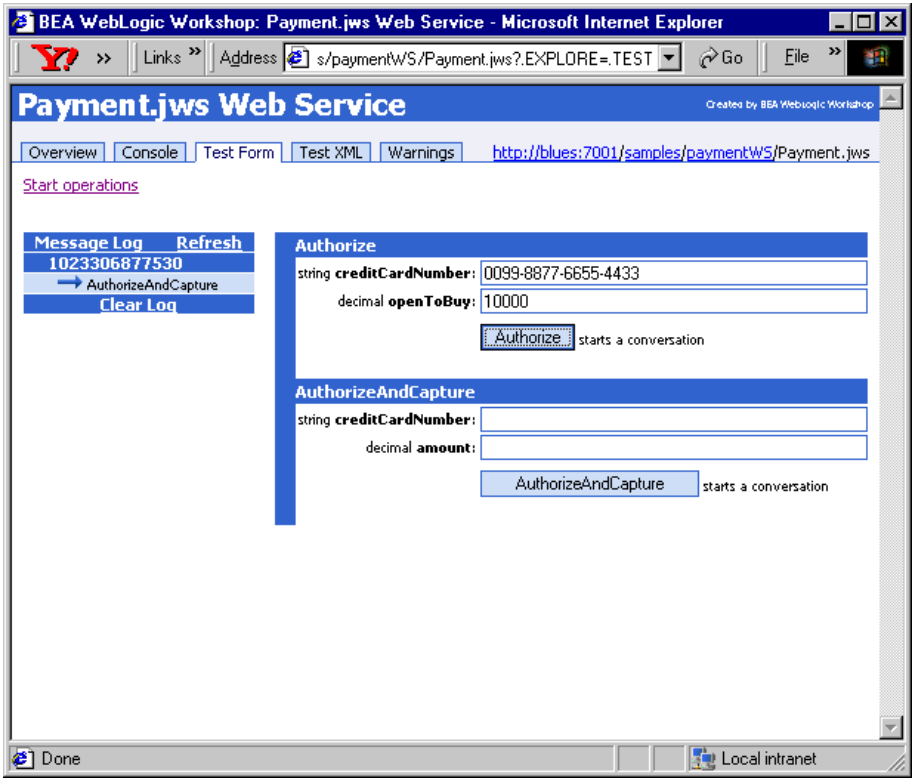
```
weblogic700\samples\workshop\cgServer
 \.jwscompile\_jwsdir_samples\classes\paymentWS\*.class
```

With the server running, click Debug →Start

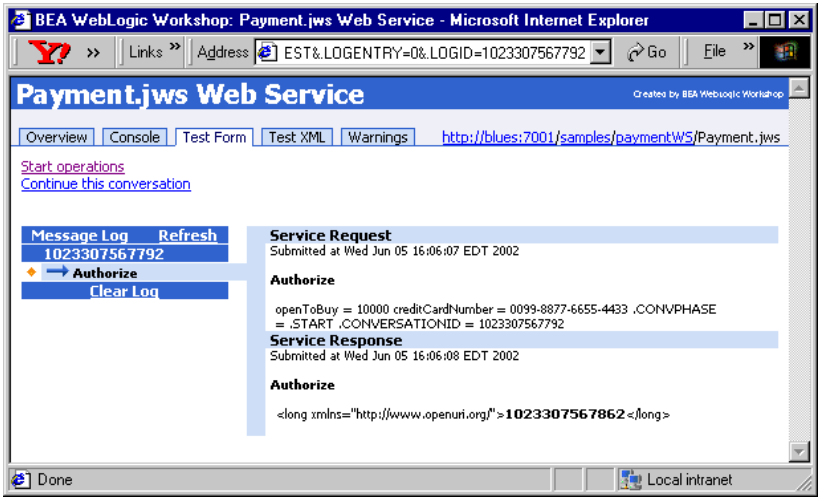
The test pages are browser-based. The initial tabbed page is the Test Form. For now, select the Overview tab. The following screen shows the Overview page with the server running on a remote machine named `blues`.



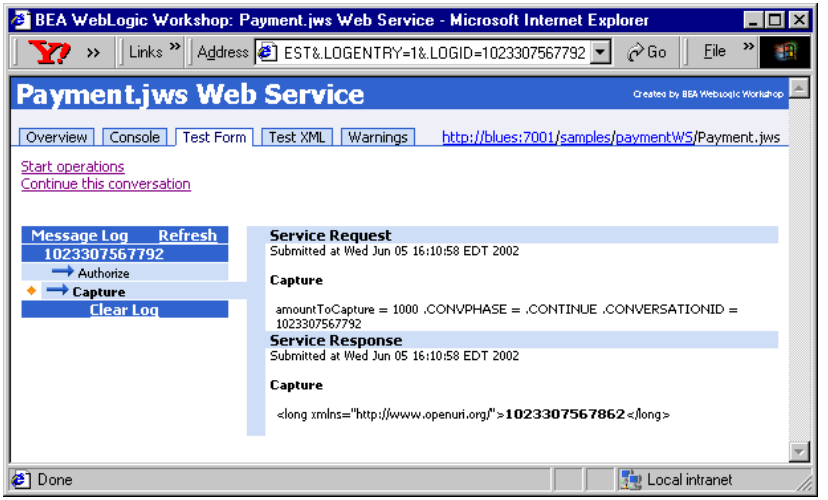
Select the Test Form tab. You can test the Payment Web service by entering sample data. For example, you can enter any set of numbers for the credit card, such as `0099-8877-6655-4433` (with or without hyphens) and an `openToBuy` authorization amount such as `10000`. For this test, we used the `Authorize` method:



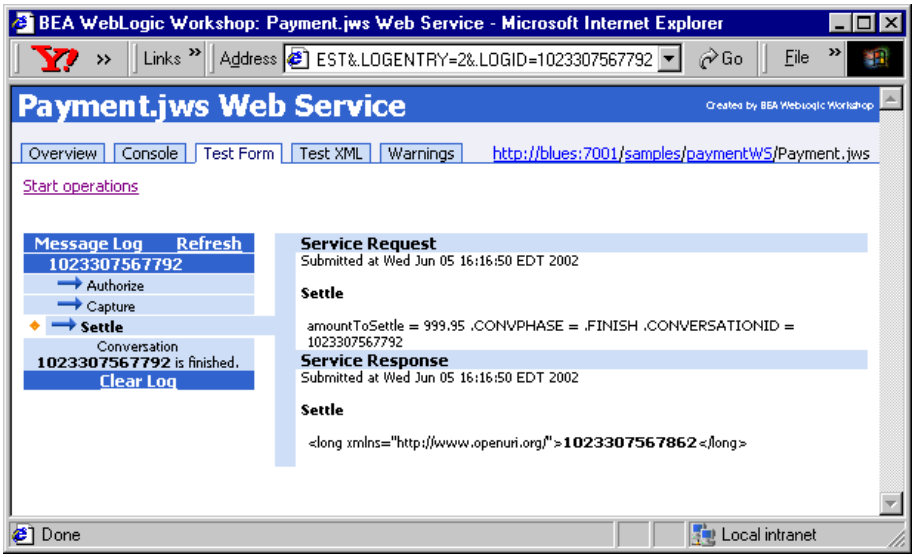
To start the conversation, click the Authorize button. This Web service method, if successful, returns a unique `transactionID`, computed as the time the transaction occurred in milliseconds, as shown in this example.



To continue the test, click the **Continue this conversation** link on the page. WebLogic Workshop returns a list of subsequent methods defined for the Payment Web service. In the `amountToCapture` variable for the Capture method, enter 1000. Then click the Capture button. If the amount to capture is less than the authorize amount (success), the Web service returns a unique `transactionID`:



Next, click the Continue this conversation link again. In the `amountToSettle` variable for the Settle method, enter a value such as 999.95. Then click the Settle button. If the amount to settle is less than the capture amount (success), the Web service returns a unique `transactionID`:



On the Test Form, you can try other values in the conversation to confirm the error handling.

Note that when you built the Web service in WebLogic Workshop, it automatically generated for you all the EJBs, with no action or work required by the developer. In our example, the first access resulted in the **`paymentWS.PaymentEJB.jar`** file creation in:

```
weblogic700\samples\workshop\cgServer
 \.jwscompile\_jwsdir_samples\EJB
```

The Web service class file was created in:

```
weblogic700\samples\workshop\cgServer
 \.jwscompile\_jwsdir_samples\classes\paymentWS\Payment.class
```

Save the Web Service's WSDL

At any time in the development process, the Web Service Description Language (WSDL) file describing your Web service is available from WebLogic Server. WSDL is a standard XML document type controlled by the World Wide Web Consortium (W3C, see `http://www.w3.org` for more information).

WSDL files describe all the methods a Web service exposes (in the form of XML messages it can accept and send), as well as the protocols over which the Web service is available. The WSDL file provides all the information a client application needs to use the web service.

There are several ways to obtain the WSDL file corresponding to a JWS file:

- In the WebLogic Workshop's Project tree, browse to the JWS file for which you would like to generate a WSDL file. In our example, browse to `Payment.jws`. Right-click on the JWS file in the Project tree and select Generate WSDL from JWS. A file with the name PaymentContract.wsdl will be created in the same directory. By default, the WSDL file is linked to the JWS file from which it was generated, meaning it will be regenerated whenever the JWS file is changed. For this reason, this is the recommended method of generating the WSDL.

- In a browser, browse to the URL of the Web service with `?WSDL` appended. For example, if the Web service is running on the default server on a machine named `blues`:

  ```
  http://blues:7001/samples/paymentWS/Payment.jws?WSDL
  ```

  Use your browser's File →Save As function to save the WSDL file to your local machine. Note that some browsers will include HTML tags at the top and bottom of the saved file. You must remove these tags to produce a valid WSDL file. Be sure to designate the `.wsdl` file type; we recommend that you follow the naming convention of `<WebServiceName>Contract.wsdl`. For example, name it `PaymentContract.wsdl`.

- In the WebLogic Workshop, you can view the WSDL by clicking the **Complete WSDL** link on the Overview page while testing your Web service. You can then use the browser's Save As function to write the Web service's file into the same directory as the JWS file. (Again, follow the naming convention shown above, and remember that some browsers will include HTML tags that you must remove from the top and bottom of the saved file.)

Packaging of PaymentWS Web Service for Use in e2eDomain

We packaged the files that comprise the Payment Web service for the installed `e2eDomain` as follows:

Web service client interfaces, CLASS files, used by WebLogic Workshop runtime

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp
 \b2cPortal\WEB-INF\classes\paymentWS\*.class
```

Web service client interfaces, JAVA files

```
weblogic700\samples\platform\e2eDomain\beaApps\e2eApp
 \src\paymentWS\*.java
```

Server-side implementation

```
weblogic700\samples\platform\e2eDomain\beaApps
 \e2eWebServicesApp\workshop\paymentWS\Payment.jws
```

This is the `Payment.jws` file copied from the WebLogic Workshop area and deployed as part of the **workshop** Web application, which was deployed as part of the `e2eWebServicesApp` enterprise application. Of course, the workshop Web application has a `WEB-INF` subdirectory that contains the configuration settings in `*.xml` files.

JAR file of the EJBs used by the paymentWS Web service

```
weblogic700\samples\platform\e2eDomain\e2eServer
 \.jwscompile\_jwsdir_workshop\EJB\paymentWS.PaymentEJB.jar
```

Webflow Input Processors, CLASS files, for Payment processing

```
weblogic700\samples\platform\e2eDomain\beaApps
 \e2eApp\b2cPortal\WEB-INF\classes\examples\e2e
 \b2c\payment\webflow\*IP.class
```

Final Step for the Web Services Tour

This concludes the Web Services Tour. In the online tour, please click the Back to Introduction button to return to the sample's Introduction page.