BEA WebLogic Portal®

Portlet Development Guide

Version 9.2
Revised: October 2006

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

Part II. Development

4. Understanding Portlet Development

5. Building Portlets

6. Optimizing Portlet Performance

7. Local Interportlet Communication

Part III. Staging

8. Assembling Portlets into Desktops

9. Deploying Portlets

Part IV. Production

10.Managing Portlets in Production

A. Portlet Database Data

CHAPTER 1

Introduction

This chapter introduces BEA WebLogic Portal® portlet concepts and describes the content of this guide.

This chapter includes the following sections:

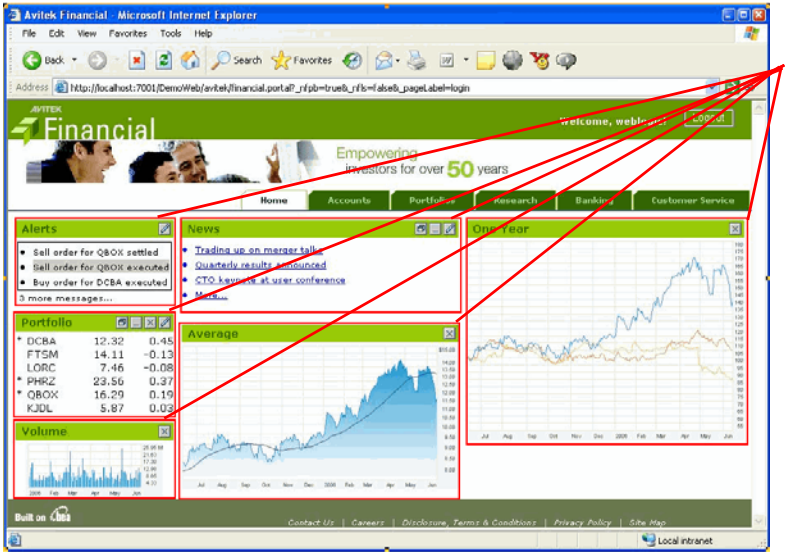
- Portlet Overview
- Portlet Development and the Portal Life Cycle

Portlet Overview

Portlets are modular panes within a web browser that surface applications, information, and business processes. Portlets can contain anything from static HTML content to Java controls to complex web services and process-heavy applications. Portlets can communicate with each other and take part in Java page flows that use events to determine a user's path through an application. A single portlet can also have multiple instances—in other words, it can appear on a variety of different pages within a single portal, or even across multiple portals if the portlet is enabled for Web Services for Remote Portlets (WSRP). You can customize portlets to meet the needs of specific users or groups.

Figure shows an example portal desktop with its associated portlets outlined in red.

Figure 1-1 Portal Desktop with Portlets



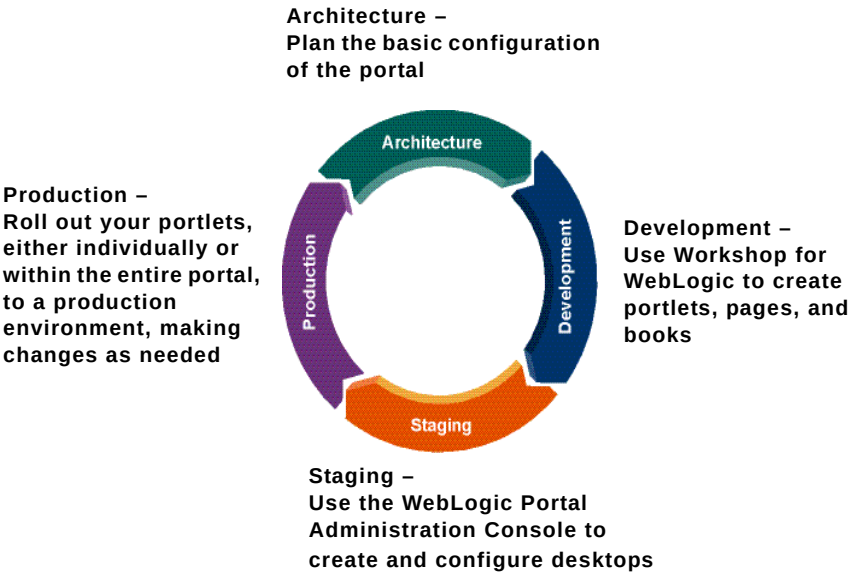
WebLogic Portal supports the development of portlets through BEA Workshop for WebLogic Platform (Workshop for WebLogic), which is a client-based tool. You can develop portals without Workshop for WebLogic through coding in any tool of choice such as JBuilder, VI or Emacs; portlets can be written in Java or JSP, and can include JavaScript for client-side operations. However, to realize the full development-time productivity gains afforded to the WebLogic Portal customer, you should use Workshop for WebLogic as your portal and portlet development platform.

For a description of each type of portlet that you can build using WebLogic Portal, refer to "Portlet Types" on page 3-1.

Portlet Development and the Portal Life Cycle

The tasks in this guide are organized according to the portal life cycle, which includes best practices and sequences for creating and updating portals. For more information about the portal life cycle, refer to the *BEA WebLogic Portal Overview*. The portal life cycle contains four phases: architecture, development, staging, and production. Figure 1-2 shows a sampling of portlet development tasks that occur at each phase.

Figure 1-2 Portlets and the Four Phases of the Portal Life Cycle



Architecture

During the architecture phase, you plan the configuration of your portal. For example, you can create a detailed specification outlining the requirements for your portal, the specific portlets you require, where those portlets will be hosted, and how they will communicate and interact with one another. You also consider the deployment strategy for your portal. Security architecture is another consideration that you must keep in mind at the portlet level.

The chapters describing tasks within the architecture phase include:

- Chapter 2, “Portlet Planning”
- Chapter 3, “Portlet Types”

Development

Developers use Workshop for WebLogic to create portlets, pages, and books. During development, you can implement data transfer and interportlet communication strategies.

In the development stage, careful attention to best practices is crucial. Wherever possible, this guide includes descriptions and instructions for adhering to these best practices.

The chapters describing tasks within the development phase include:

- Chapter 4, “Understanding Portlet Development”
- Chapter 5, “Building Portlets”
- Chapter 6, “Optimizing Portlet Performance”
- Chapter 7, “Local Interportlet Communication”

Staging

BEA recommends that you deploy your portal, including portlets, to a staging environment, where it can be assembled and tested before going live. In the staging environment, you use the WebLogic Portal Administration Console to assemble and configure desktops. You also test your portal in a staging environment before propagating it to a live production system. In the testing aspect of the staging phase, there is tight iteration between staging and development until the application is ready to be released.

The chapters describing tasks within the staging phase include:

- Chapter 8, “Assembling Portlets into Desktops”
- Chapter 9, “Deploying Portlets”

Production

A production portal is live and available to end users. A portal in production can be modified by administrators using the WebLogic Portal Administration Console and by users using Visitor Tools. For instance, an administrator might add additional portlets to a portal or reorganize the contents of a portal.

The chapter describing tasks within the production phase is:

- Chapter 10, “Managing Portlets in Production”

Getting Started

This section describes the basic prerequisites to using this guide and lists guides containing related information and topics.

Prerequisites

In general, this guide assumes that you have performed the following prerequisite tasks before you attempt to use this guide to develop portlets:

- Review the Related Guides and become familiar with the basic operation of the tools used to create portals, portlets, and desktops,
- Review the Workshop for WebLogic tutorials and documentation to become familiar with the Eclipse-based development environment and the recommended project hierarchy.
- Complete the tutorial *Getting Started with WebLogic Portal*.

Related Guides

BEA recommends that you review the following guides:

- *BEA WebLogic Portal Overview*
- *BEA WebLogic Portal Development Guide*

Whenever possible, this guide includes cross references to material in related guides.

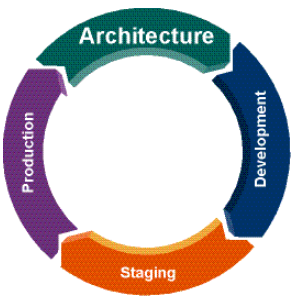
Part I Architecture

Part I includes the following chapters:

- Chapter 2, “Portlet Planning”
- Chapter 3, “Portlet Types”

During the architecture phase, you plan the configuration of the portlets that comprise your portal.

For a view of how the tasks in this section relate to the overall portal life cycle, refer to the *WebLogic Portal Overview*.



CHAPTER 2

Portlet Planning

Proper planning is essential to portlet development. A properly planned portlet structure and organizational model can provide a cohesive and consistent portal interface, flexible scalability, and high performance without requiring frequent adjustments within your production system.

This chapter focuses on planning considerations and decisions that should precede the development of your portlets. Global portal-wide planning information is provided in the *BEA WebLogic Portal Overview*, which summarizes the types of issues to consider in the architecture phase across your portal environment. The various WebLogic Portal feature guides, such as the *BEA WebLogic Portal Federated Portals Guide*, describe architectural issues in detail for each feature area.

This chapter includes the following sections:

Portlet Development in a Distributed Portal Team

If you will be creating portlets within an environment that includes a remote (distributed) development team, you must carefully plan your implementation. Considerations for team development include:

- **Using shared resources** – You can have common portlets, such as the login portlet.
- **Sharing a common domain** – You can have a common domain among team members with different BEA home directories.
- **Integrating remotely developed portlets into the portal** – You need to manage settings that are common to the portal application, which must match across the entire development project.

Team development of a WebLogic Portal web site revolves around well-designed source control and a correctly configured shared domain for development. For detailed instructions on setting up your development environment, refer to the Team Development chapter of the *Production Operations Guide*.

Portlets in a Non-Portal Environment

In some cases, you might want to expose portlets in a web page even though that web application is not based on WebLogic Portal. For example, you might want to expose portlets with WSRP from a producer environment that does not include any WebLogic Portal components. You might be running a Struts web application in a basic WebLogic Server domain, or a Java page flow application in a basic Workshop for WebLogic domain. In either case, WebLogic Portal is not part of the server configuration. The exposed portlets can then be consumed by remote portlets running in a regular WebLogic Portal domain.

For more information on developing portlets for a non-WebLogic Portal environment, refer to the *Federated Portals Guide*.

Planning Portlet Instances

In the Development phase, you use Workshop for WebLogic to create portlets and place them onto a portal. In the Staging phase, you use the Administration Console to add portlets to portal desktops. Each time you add a portlet to a desktop, you create an *instance* of that portlet. Portlet instances allow for multiple variations of the same portlet definition. By using portlet instances, portal users and administrators can configure multiple views of the same portlet through the use of portlet preferences, and reduce the overall number of distinct portlets; this portlet reuse

improves portal performance and management efficiency. A common example of portlet instances is a stock watch portlet in which there is a single or multi-valued preference for ticker symbols such as BEAS, which would configure the portlet to display BEA Systems stock information.

Try to plan your portal hierarchy to reuse portlets when practical. For more information about portlet instances and how portlet instances are related to portlets in the Administration Console's portlet library, refer to "Portlet Library" on page 8-1.

Security

You can control access to portlet resources for two categories of users:

- **Portal visitors** – You control access to portal resources using *visitor entitlements*. Visitor access is determined based on visitor entitlement roles.
- **Portal administrators** – You control portal resource management capabilities using *delegated administration*. Administrative access is determined based on delegated administration roles.

During the architecture phase, you plan how to organize security policies and roles, and how that fits into your system-wide security strategy. You implement your security plans by setting up delegated administration and visitor entitlements using the WebLogic Portal Administration Console.

For an overall look at managing security for your portal environment, refer to the *Security Guide*. Specific security considerations for feature areas are contained in those documents; for example, recommendations for security in WSRP-enabled environments are contained in the *Federated Portals Guide*.

Interportlet Communication

Interportlet communication (IPC) allows multiple portlets to use or react to data. You can use interportlet communication within a single portal web application, or within federated portal applications.

For more information on interportlet communication within a single portal web application, refer to Chapter 7, "Local Interportlet Communication." For more information on interportlet communication within federated portal applications, refer to the *Federated Portals Guide*.

Performance Planning

Try to plan for good performance within your portlet architecture to minimize the fine-tuning that is required in a production environment.

Here are some examples of performance optimizations that you can plan into your overall portal strategy:

- **Portlet caching –** You can cache the portlet within a session instead of retrieving it each time it recurs during a session (on different pages, for example).
- **Remote portlets –** With remote portlets, any portal controls within the application (portlet) that you are retrieving are rendered by the producer and not by your portal. The expense of calling the control life cycle methods is borne by resources not associated with your portal. You must balance this advantage against the delay that might be caused by network latency issues.
- **Customized portlet properties –** Customizing your portlet settings can help you improve performance; for example, you can set process-expensive portlets to be processed in a multi-threaded (forkable) environment.
- **Asynchronous portlet rendering** - Asynchronous portlet rendering allows you to render the content of a portlet independently from the surrounding portal page. You can use either AJAX technology or IFRAME technology to implement asynchronous rendering.

Plan your performance optimizations before you begin developing portlets so that you can implement any pre-requisites that are required. For detailed instructions on developing high-performance portlets, refer to Chapter 6, “Optimizing Portlet Performance.” For post-development WebLogic Portal performance recommendations, refer to the *Performance Tuning Guide*.

CHAPTER 3

Portlet Types

As part of your portlet implementation plan, BEA recommends that you examine the different types of portlets that are available in WebLogic Portal and decide which types are best suited for the tasks that you want to accomplish. For example, if you are looking for a way to interface with Java controls, use Struts-based infrastructure, and deliver rich navigation elements, then you might choose to implement Java Page Flow or Struts portlets. If you are looking for a simple portlet or you want to convert an existing JSP page into a portlet, you might consider using a JSP portlet. If you work for an independent software company or other enterprise that is concerned with portability across multiple portal vendors, then you might choose to use JSR 168-compliant Java portlets whenever possible. If you want to implement asynchronous portlet rendering in your portal, you can use nearly any of the portlet types described in this chapter.

This chapter differentiates the various portlet types to help you in your decision-making process. This chapter contains the following sections:

Java Server Page (JSP) and HTML Portlets

JSP portlets and HTML portlets point to JSP or HTML files for their content. These portlets can be simple to implement and deploy, and they provide basic functionality quickly. However, this type of portlet does not enforce separation of business logic and the presentation layer. As the application grows, the portlet often becomes harder to maintain as you try to update the web application and share code. JSP portlets are not well-suited for advanced portlet navigation.

When using JSP pages as part of a page flow portlet, you must make sure that requests adhere to WebLogic Portal scoping requirements. For more information about JSP portlets and page flow scoping, refer to the *Portal Development Guide*.

For instructions on building JSP portlets, see "JSP and HTML Portlets" on page 5-11.

Java Portlets (JSR 168)

JSR 168 (Java Portlet) is a Java specification that aims at establishing portability between portlets and portals. One of the main goals of the specification is to define a set of standard Java APIs for portal and portlet vendors. These APIs cover areas such as presentation, aggregation, security, and portlet life cycle.

A Java portlet is expressed as a Java class. This type of portlet accommodates portability across platforms, and does not require the use of portal server-specific JSP tags. The behavior is similar to a servlet. Java portlets produced using WebLogic Portal can be used universally by any other vendor's application server container that supports JSR 168.

For instructions on building Java portlets, refer to "Java Portlets" on page 5-12.

Java Page Flow Portlets

A Java page flow portlet uses Apache Beehive page flows to retrieve its content. This portlet type allows you to separate the user interface code from navigation control and other business logic, and provides the ability to implement both simple and advanced portlet navigation.

The Page Flow framework that is recommended for portlet application development is built on top of the Struts application framework. The Struts framework is a popular, reliable standard that is widely used to quickly create robust and navigable web applications. The page flow framework

adds valuable data binding facilities to the Struts standard, and the portal framework provides a scoping capability for page flow portlets so that multiple page flows can be supported in a single portal. You can use resources such as Java controls and web services.

Java page flow portlets are best suited for an environment where more advanced features are required—not for static, single-view portlets.

For instructions on building Java page flow portlets, refer to "Java Page Flow Portlets" on page 5-17.

Struts Portlets

Struts portlets are based on the Struts framework, which is an implementation of the Model-View-Controller (MVC) architecture. The MVC architecture provides a model for separating the different components and roles of the application logic. This development framework helps you create portlets that are easier to maintain over time.

Typically, native Struts development requires management and synchronization of multiple files for each action, form bean, as well as the Struts configuration file. Even in the presence of tools that help edit these files, developers are still exposed to all the underlying plumbing, objects, and configuration details. The Page Flow implementation provides a simpler, single-file programming model that allows developers to focus on the code they care about, see a visual representation of the overall application flow, and navigate between pages, actions, and form beans.

If you are developing a portal application from scratch, BEA recommends using a Page Flow implementation; if your goal is to aggregate an existing Struts application, then using Struts portlets can meet your needs.

For instructions on building Struts portlets, refer to "Struts Portlets" on page 5-27.

Java Server Faces (JSF) Portlets

The Java Server Faces (JSF) specification, JSR 127, defines a user interface framework that simplifies development and maintenance of Java applications that run on a server and are displayed and used from a client.

According to the Java Server Faces Specification, available from the Java Community Process web site:

JSF's core architecture is designed to be independent of specific protocols and markup. However it is also aimed directly at solving many of the common problems encountered when writing

applications for HTML clients that communicate via HTTP to a Java application server that supports servlets and JavaServer Pages (JSP) based applications. These applications are typically form-based, and are comprised of one or more HTML pages with which the user interacts to complete a task or set of tasks. JSF tackles the following challenges associated with these applications:

- Managing UI component state across requests
- Supporting encapsulation of the differences in markup across different browsers and clients
- Supporting form processing (single multi-page form, or more than one form per page)
- Providing a strongly typed event model that allows the application to write server-side handlers (independent of HTTP) for client generated events
- Validating request data and providing appropriate error reporting
- Enabling type conversion when migrating markup values (Strings) to and from application data objects (which are often not Strings)
- Handling error and exceptions, and reporting errors in human-readable form back to the application user
- Handling page-to-page navigation in response to UI events and model interactions.

For instructions on building Java Server Faces portlets, refer to "JSF Portlets" on page 5-20.

Browser (URL) Portlets

Browser portlets display HTML content from an external URL. Unlike other portlet types that are limited to displaying data contained within the portal project, browser portlets display URL content that is external from the portal project.

An advantage of browser portlets is that no development tasks are required to implement it, either from the Workshop for WebLogic workbench or from the WebLogic Portal Administration Console. However, keep in mind that WebLogic Portal does not provide a mechanism to develop content for this type of portlet; the definition of the portlet merely contains the external URL to display. For example, no mechanisms exist to dynamically influence the external content's URL; no support exists for portlet preferences, portlet modes, and so on. Browser portlets do not track the URL through the user's interaction with remote content – page refreshes cause the content of the URL specified in the portlet definition to be displayed.

WebLogic Portal implements a browser portlet using an IFRAME. You can override the default implementation mechanism using more advanced development techniques; more detailed documentation about these techniques will be provided in a future dev2dev article.

The content of the browser portlet is completely disconnected from the portal. The embedded application must manage the navigational state of the portlet.

For instructions on building Browser portlets, refer to "Browser Portlets" on page 5-24.

Remote Portlets

WebLogic Portal supports the Web Services for Remote Portlets (WSRP) standard, a product of the OASIS standards body. Portlets that are written to meet this standard, which includes a WSDL portlet description, can be hosted within a producer application, and surfaced in a consumer application. Moreover, the WebLogic Portal Administration Console facilitates access to WSRP producer applications in a local portal.

WebLogic Portal can act as either a WSRP remote producer or as a consumer. When acting as a consumer, WebLogic Portal's remote—or proxy—portlets are WSRP-compliant. These portlets present content that is collected from WSRP-compliant producers, allowing you to use external sources for portlet content, rather than having to create its content or its structure yourself.

Because setting up a remote portlet is a fundamental task in creating a federated portlet environment, the task of creating a remote portlet is described in detail within the *Federated Portals Guide*.

Portlet Type Summary Table

Table 3-1 summarizes the characteristics of each portlet type so that you can quickly determine the advantages and disadvantages of each type.

Table 3-1 Portlet Type Summary Table

Type	Advantages	Disadvantages
JSP/HTML	Simple to implement and deploy. Provides basic functionality without complexity.	Does not enforce separation of business logic and presentation layer. Not well-suited for advanced portlet navigation.
Java (JSR 168)	Accommodates portability across platforms. Does not require the use of portal server-specific JSP tags. Behavior is similar to a servlet	Lack of advanced portlet features that are available with some other portlet types. Requires a deeper understanding of the J2EE programming model.
Java Page Flow	Allows separation of the user interface code from navigation control and other business logic. Provides the ability to implement both simple and advanced portlet navigation. Allow you to quickly leverage Java controls, web services, and business processes. Provides a visual environment to build rich applications based on struts.	Implementation is more complex. Advanced page flow features are not necessary for static or simple, one view portlets.
JSF	Allows component-based development of pages that can handle their own intra-page events. Simplifies separation of the user interface code from navigation control and other business logic. Provides the ability to implement both simple and advanced portlet navigation. Allow you to quickly leverage Java controls, web services, and business processes.	All postbacks to a JSF application are expected to be done using a POST; the GET method is not supported.

Table 3-1 Portlet Type Summary Table (Continued)

Type	Advantages	Disadvantages
Browser	Allows a portlet to display content from a URL that is outside the portal project. Provides a "no development needed" portlet for quick implementation.	Less control over formatting. Lacks certain features of other portlet types, such as Content Path and Error Path. No interportlet communication support.
Struts	Provides a flexible control layer based on standard technologies like Java Servlets, JavaBeans, ResourceBundles, and XML. Provides a more structured approach for creating and maintaining complex applications. Useful for importing existing applications.	Not quite as robust as page flow portlets, which are based on Beehive. For new development, page flow portlets provide a better solution.
Remote	Allows you to functionally and operationally de-couple applications within your portal. Allows you to leverage external sources for portlet content. Depending on the environment, might improve performance.	Implementation is more complex. Your application's features might not be able to be as robust; for example, some Javascript might not perform correctly. Depending on the environment, might have a performance cost. For more about performance with remote portlets, refer to "Remote Portlets" on page 6-2.

Part II Development

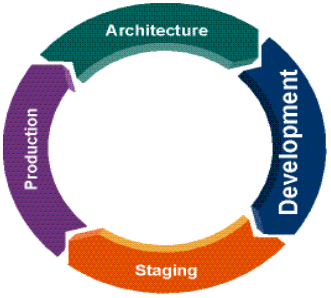
Part II includes the following chapters:

- Chapter 4, “Understanding Portlet Development”
- Chapter 5, “Building Portlets”
- Chapter 6, “Optimizing Portlet Performance”
- Chapter 7, “Local Interportlet Communication”

During the development phase, you use Workshop for WebLogic to create portlets, pages, and books. During development, you can implement federation and interportlet communication strategies. In the development stage, careful attention to best practices is crucial.

For a view of how the tasks in this section relate to the overall portal life cycle, refer to the *WebLogic Portal Overview*.

.



CHAPTER 4

Understanding Portlet Development

This chapter provides conceptual and reference information that you might find useful as you begin to develop portlets with WebLogic Portal. For a detailed description of the components that are involved in portlet design, refer to the *Portal Development Guide*. For instructions on how to create each type of portlet, refer to "Building Portlets" on page 5-1.

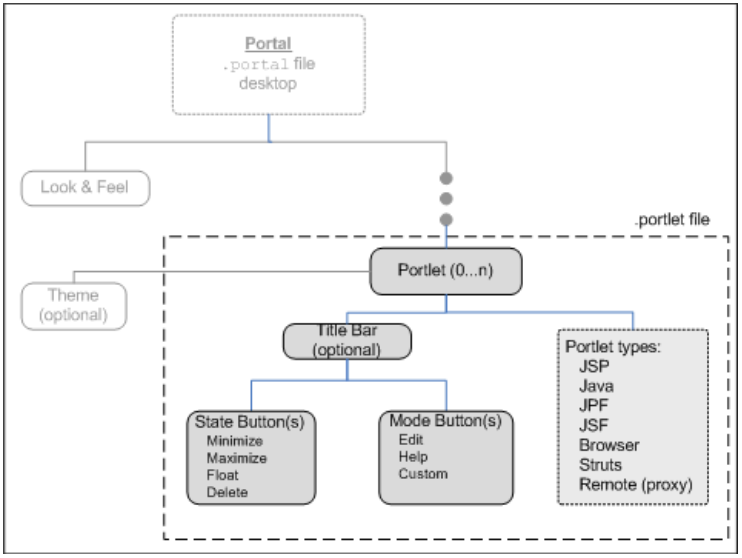
This chapter contains the following sections:

Portlet Components

Portlets are modular panes within a web browser that surface applications, information, and business processes. Portlets can contain anything from static HTML content to Java controls to complex web services and process-heavy applications. Within a portal application, a portlet is represented as an XML file with a `.portlet` file extension. As you build portlets using Workshop for WebLogic, the XML elements and attributes are automatically built.

Figure 4-1 shows the components that make up a portlet, which are located in the `.portlet` file. Objects shown in gray text are described in more detail within the *Portal Development Guide*.

Figure 4-1 Portlet Components



This section includes the following topics:

- Portlet Properties
- Portlet Look & Feel Components

  For details about Look & Feel components, refer to the *Portal Development Guide*.
- Portlet Title Bar, Mode, and State
- Portlet Preferences

Portlet Properties

Portlet properties are named attributes of the portlet that uniquely identify it and define its characteristics. Some properties—such as title, definition label, and Content URI—are required; many other optional properties allow you to enable specific functions for that portlet such as scrolling, presentation properties, pre-processing (such as for authorization) and multi-threaded rendering. The specific properties that you use for a portlet vary depending on your expected use for that portlet.

For detailed information on portlet properties and how to set them, refer to "Portlet Properties" on page 5-32.

Portlet Title Bar, Mode, and State

When you create a portlet, you can choose whether or not it should have a title bar. Also, all portlets created with WebLogic Portal support modes and states. Modes affect the portlet's content; edit, help, float, and custom modes are available. States affect the rendering of the portlet; minimize, maximize, normal, float, and delete states are available.

You must enable the title bar on a portlet if you want to set modes and states for that portlet.

In certain situations your selection of a mode and state for a portlet might affect your ability to set up other portlet features, such as interportlet communication. For example, if you are setting up an event handler that listens to a portlet, you can select to execute the event handler only if the portlet to which it is listening is in a window that is *not minimized*, and is in *view mode*.

For detailed instructions on setting portlet modes and states, refer to "Portlet Appearance and Features" on page 5-69.

Portlet Preferences

Portlets are distinct applications that you can reuse in a given portal. Once you create a portlet, you can instantiate it several times.

Along with the ability to create multiple instances of portlets, WebLogic Portal allows you to specify preferences for portlets. You use preferences to cause each portlet instance to behave differently yet use the same code and user interface. Portlet preferences provide the primary means of associating application data with portlets; this feature is key to personalizing portlets based on their usage.

Plan a portlet implementation that allows portlets to be as reusable as possible; planning for reuse simplifies your development and testing efforts because you can differentiate generic portlets by setting unique preferences.

For detailed instructions on setting portlet preferences, refer to "Portlet Preferences" on page 5-49.

Resources for Creating Portlets

Although the Portlet Wizard provides an easy way to create portlets, you might find that it is not your primary means of creating them. You can create a portlet in many ways, such as duplicating existing portlets or generating a portlet based on an existing JSP or struts module. Many resources can provide the raw material for a portlet, including the following:

- **Portlets in Library Modules** - Portlets provided with WebLogic Portal, which you can copy into your project and modify for your use. For example, you can add the Collaboration Portlets (pre-built portlets that are supplied with WebLogic Portal) to your Portal Web Project, and have access to Calendar, Task, Address Book, Discussion, and Mail portlets. These portlets are located in the library module `wlp-collab-portlets-web-lib` in the path `WebLogic_HOME/portal/lib/modules/`. For step-by-step instructions on adding the Collaboration portlets, refer to the *Communities Guide*. For a complete list of library modules and their locations, refer to the *Portal Development Guide*.

  Caution: Portlets that are part of the GroupSpace sample application cannot be used outside of the GroupSpace application.

- **Third-party portlets** - Special-purpose portlets provided as separate products by partner companies.

- **Existing JSPs, Struts modules, and Page Flows** – Existing resources that you can drag onto a portal page to automatically generate a portlet.

You can find detailed instructions on how to use these resources as the basis for a portlet in Chapter 5, "Building Portlets."

Portlet Rendering

Portlet rendering consists of two processes:

- **Pre-rendering** – The background work to obtain necessary data or to perform pre-processing

- **Rendering** – The actual drawing of the portlet onto the portal page

General rendering topics are covered in the *Portal Development Guide*. This section contains the following portlet-specific rendering topics:

- Render and Pre-Render Forking

- Asynchronous Portlet Content Rendering

Render and Pre-Render Forking

By default, pre-rendering and rendering for each portlet on a page is performed in sequence, and the portal page is not displayed until processing is complete for every portlet. This sequence can cause a noticeable delay in displaying the web page and might cause a user to think there is a

problem with the web site. To prevent this situation, you can set up your portlets so that they perform pre-rendering and rendering tasks in parallel using multi-threaded *forked* processing.

Forking portlets at the rendering stage is supported for all portlet types. Pre-render forking is supported for the following portlet types:

- JSP
- Page flow
- Java (JSR168)
- WSRP (consumer portlets only)

For detailed instructions on implementing forked portlets, refer to "Portlet Forking" on page 6-3.

Asynchronous Portlet Content Rendering

Asynchronous portlet rendering allows the content of a portlet to be rendered independently of the surrounding portal page. When using asynchronous portlet rendering, a portlet is rendered in two phases. The first phase is the normal portal page request during which the portlet's non-content areas, such as the title bar, are rendered; a second request causes the portlet's content to render in place.

For detailed instructions on implementing asynchronous content rendering, refer to "Asynchronous Portlet Content Rendering" on page 6-13.

Portlets as Popups (Detached Portlets)

WebLogic Portal supports the use of detached portlets. Detached portlets provide popup-style behavior. You can see examples of detached portlets within WebLogic Portal in the GroupSpace Message Center and in the Administration Console wizards.

For detailed instructions on using detached portlets, refer to "Detached Portlets" on page 5-30.

JSP Tags and Controls in Portlets

WebLogic Portal provides JSP tags that you can use within JSPs. Portlets can use JSPs as their content nodes, enabling reuse and facilitating personalization and other programmatic functionality. When you use the JSP Design Palette view in Workshop for WebLogic, you can view available JSP tags and then drag them into the Source View of your JSP, and use the Properties view to edit elements of the code.

To view the JSP tags available as you develop a portal, select **Window > Show View > JSP Design Palette**.

WebLogic Portal also provides custom Java controls that make it easy for you to quickly add pre-built modules to your portal; custom Java controls exist for event management, Visitor Tools, Community management, and so on. For example, most user management functionality can be easily exposed with a User Manager Control on a page flow.

Note: The term control is also used to refer to the portal (netuix) framework controls, such as desktop, book, page, and so on. These controls are referred to in the text as *portal framework controls*.

For information about the classes associated with WebLogic Portal's JSP tags, refer to the *Javadoc*.

For more information about using controls within portlets, see "JSP Tags and Controls in Portlets" on page 5-81.

Backing Files

The most common means of influencing portlet behavior within the control life cycle is to use a portlet backing file. A portlet backing file is a Java class that can contain methods corresponding to Portal control life cycle stages, such as init() and preRender(). You can use a portlet's backing context, an abstraction of the portlet control itself, to query and alter the portlet's characteristics. For example, in the init() life cycle method, a request parameter might be evaluated, and depending on the parameter's value, the portlet backing context can be used to specify whether the portlet is visible or hidden.

Backing files can be attached to portals either by using Workshop for WebLogic or coding them directly into a `.portlet` file.

For detailed instructions on implementing backing files, refer to "Backing Files" on page 5-64.

CHAPTER 5

Building Portlets

This chapter describes the most common ways to create portlets, including the Portlet Wizard and the use of out-of-the-box portlets. This chapter also contains instructions for building each type of portlet that is supported by WebLogic Portal.

Before you begin, be sure you are familiar with the concepts associated with creating portlets, as described in Chapter 4, "Understanding Portlet Development."

This chapter contains the following sections:

- Portlet State Persistence
- Adding a Portlet to a Portal
- Deleting Portlets
- Third-Party Portlets
- Advanced Portlet Development with Tag Libraries

Supported Portlet Types

The following portlet types are supported by WebLogic Portal:

- Java Server Page (JSP) and HTML Portlets - JSP portlets and HTML portlets point to JSP or HTML files for their content.
- Java Portlets (JSR 168) - Java portlets produced using WebLogic Portal can be used universally by any vendor's application server container that supports JSR 168.
- Java Page Flow Portlets - Java page flow portlets use Apache Beehive page flows to retrieve their content.
- Java Server Faces (JSF) Portlets - JSF portlets produced using WebLogic Portal conform to the JSR 127 specification.
- Browser (URL) Portlets - Browser portlets display HTML content from an external URL; no development tasks are required to implement them.
- Struts Portlets - Struts portlets are based on the Struts framework, which is an implementation of the Model-View-Controller (MVC) architecture.
- Remote Portlets - WebLogic Portal's remote portlets conform to the WSRP standard; they can be hosted within a producer application, and surfaced in a consumer application.

For a detailed discussion of each portlet type, refer to Chapter 3, "Portlet Types."

Portlets in Library Modules

You can copy portlets or other resources from a library module into your portal application and modify them as needed. A portlet existing in your project will supersede a portlet of the same name in a library module. To see a list of available portlets, you can use the Merged Projects View of the workbench; resources contained in library modules are shown in italic print. You can expand the tree to see the resources that are stored in the various modules. For a reference list of

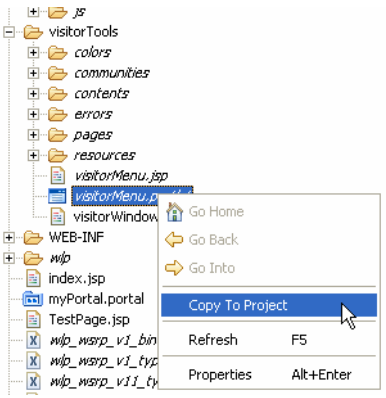
all the library modules and their locations on your file system, you can select **Window > Preferences > WebLogic > Library Modules**.

After you locate a portlet that you want to use, you can right-click the portlet in the Merged Projects View and select the **Copy to Project** option. Figure 5-1 shows an example of a library module portlet in the Merged Projects view with the Copy to Project option selected.

Caution: Portlets that are part of the GroupSpace sample application cannot be used in a non-GroupSpace-enabled application.

If you copy J2EE library resources into your project, keep in mind that with future updates to the WebLogic Portal product, you might have to perform manual steps in order to incorporate product changes that affect those resources. *With any future patch installations, WebLogic Portal supports only configurations that do not have copied J2EE library resources in the project.*

Figure 5-1 Portlet Being Copied to a Project from Merged Projects View



For more information about library modules, refer to the *Portal Development Guide*.

Portlet Wizard Reference

An important tool that you can use to create portlets from scratch is the WebLogic Portal Portlet Wizard. The following sections describe the Portlet Wizard in detail:

- Order of Creation - Resource or Portlet First
- Starting the Portlet Wizard

- Select Portlet Type Dialog
- Portlet Details Dialogs

In general, you choose the portlet type on the first dialog of the wizard; when generating a portlet based on an existing resource, the Portlet Wizard automatically detects the portlet type whenever possible.

Order of Creation - Resource or Portlet First

This section provides an overview of the two methods you can use to begin creating a portlet—creating the portlet resource information/file first or creating the portlet itself first.

Creating the Resource First

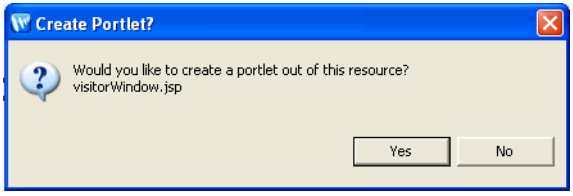
You might already have a JSP file, for example, that you want to use as the basis for a portlet. (In addition to JSP files, you can drag other resources onto the portal (such as content selectors) to automatically start the portlet wizard.)

If you have an existing resource that you want to use as the basis of a portlet, follow these steps:

1. Create or open a portal's `.portal` file in Workshop for WebLogic.
2. Drag the resource, such as a JSP file, into one of the portal's placeholder areas in the design view in the editor.

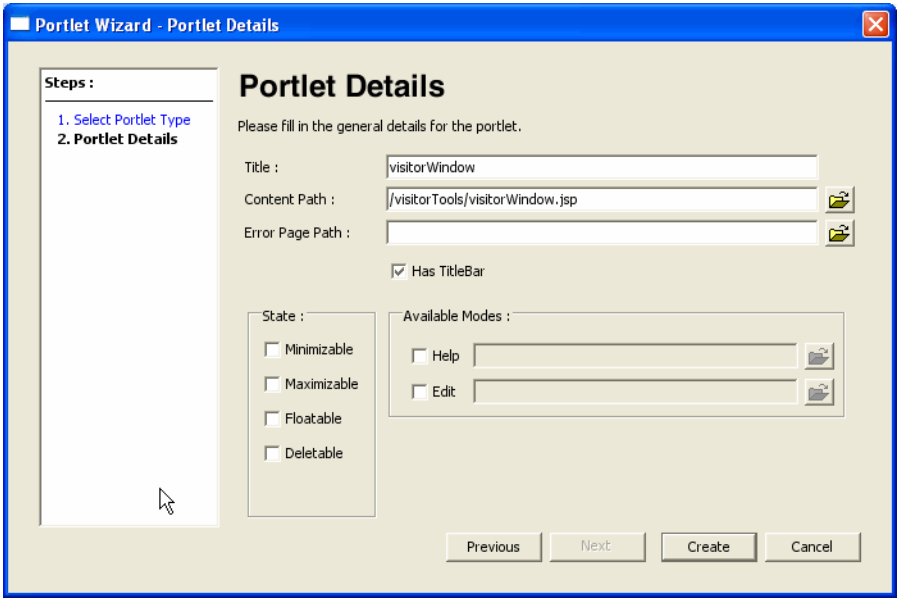
   Workshop for WebLogic prompts you with a dialog similar to the example in Figure 5-2.

Figure 5-2 Portlet Wizard Prompt Following Drag and Drop of a Resource



If you click **Yes**, the Portlet Wizard uses information from the resource file to begin the process of creating a portlet, and displays the Portlet Details dialog. Figure 5-3 shows an example:

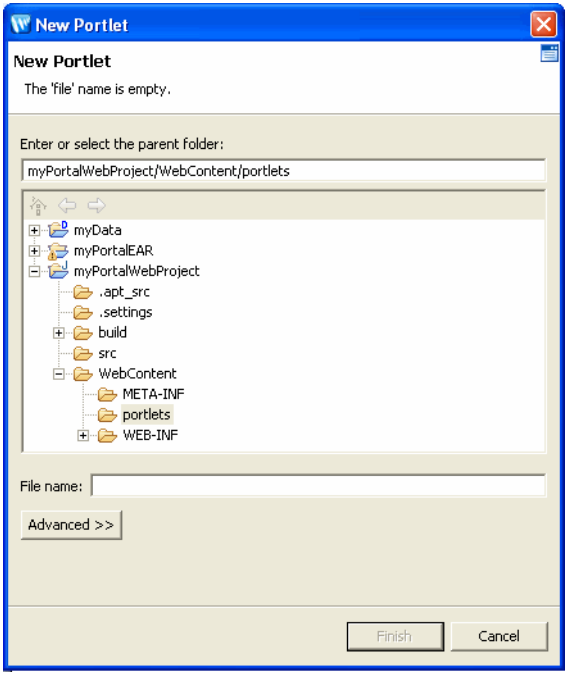
Figure 5-3 Example Portlet Wizard Details Dialog Following Drag and Drop of a Resource



Create the Portlet First

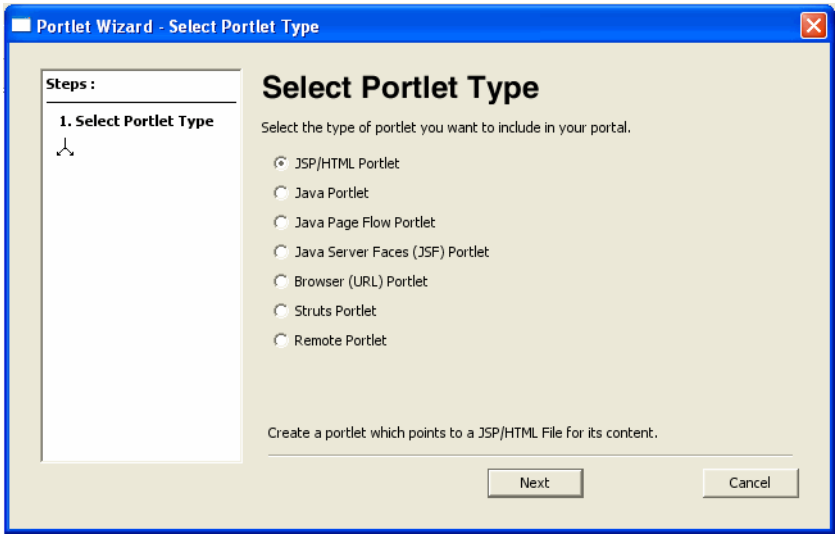
If you do not have an existing source file to start with, you can create the portlet using the New Portlet dialog and the Portlet Wizard. To do so, right-click a folder in your portal web project and select **New > Portlet.** Figure 5-4 shows an example of the New Portlet dialog.

Figure 5-4 Portlet Wizard New File Dialog



After you confirm or change the parent folder, name the portlet, and click **Finish**, the Portlet Wizard begins and displays the Select Portlet Type dialog. Figure 5-5 shows an example dialog.

Figure 5-5 Portlet Wizard - Select Portlet Type Dialog



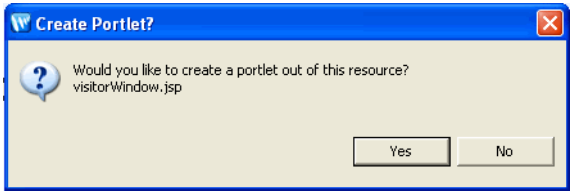
Detailed instructions for creating each type of portlet are contained in "How to Build Each Type of Portlet" on page 5-10.

Starting the Portlet Wizard

Workshop for WebLogic invokes the Portlet Wizard any time you perform one of these operations:

- Select **File > New > Portlet** from Workshop for WebLogic's top-level menu, or right-click a folder in your web application, and select **New > Portlet**. After you name the portlet and click **Create**, the Portlet Wizard starts.
- Drag and drop a resource such as a JSP from the Package Explorer view onto a placeholder area of an open portal (in other words, a *portal_name*.portal file is open in the editor view of the workbench.) Workshop for WebLogic prompts you with a dialog similar to the example in Figure 5-6.

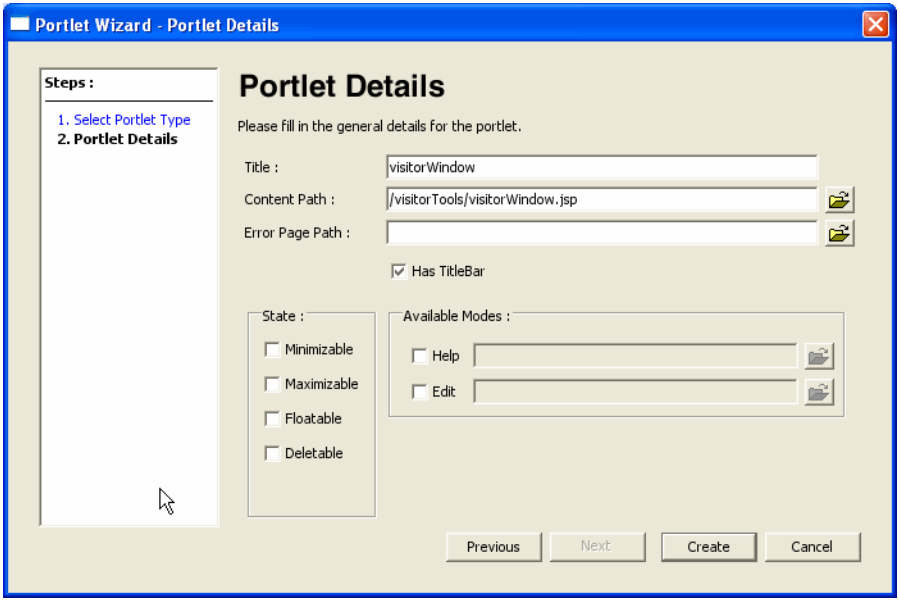
Figure 5-6 Portlet Wizard Prompt Following Drag and Drop of a Resource



If you click **Yes**, the Portlet Wizard uses information from the resource file to begin the process of creating a portlet.

- Right-click an existing resource such as a JSP file, a page flow, a portal placeholder, or a portal content selector; then select **Generate Portlet** from the context menu. The Portlet Wizard displays the **Portlet Details** dialog. Figure 5-7 shows an example of a dialog after right-clicking a JSP file.

Figure 5-7 Portlet Wizard - Portlet Details Example for JSP Resource



New Portlet Dialog

When you use **File > New > Portlet** to create a new portlet, a New Portlet dialog displays before the Portlet Wizard begins. Figure 5-4 shows an example of the New Portlet dialog.

The parent folder defaults to the location from which you selected to add the portlet.

This dialog requires that you select a project and directory for the new portlet, and provide a portlet file name. (The file name appears in the Package Explorer view after you create the portlet.) The **Finish** button is initially disabled; the button enables when you select a valid project/directory and portlet name. If you select an invalid portal project in the folder tree on this dialog, an error message appears in the status area near the top of the dialog explaining that the project is not a valid portal project. You cannot continue until you have selected a valid project (if one is available).

Note: With WebLogic Portal Version 9.2, the option to convert a non-portal project to a portal project is not offered. For information on how to integrate portal library modules into an already existing project, see the *Portal Development Guide*.

Select Portlet Type Dialog

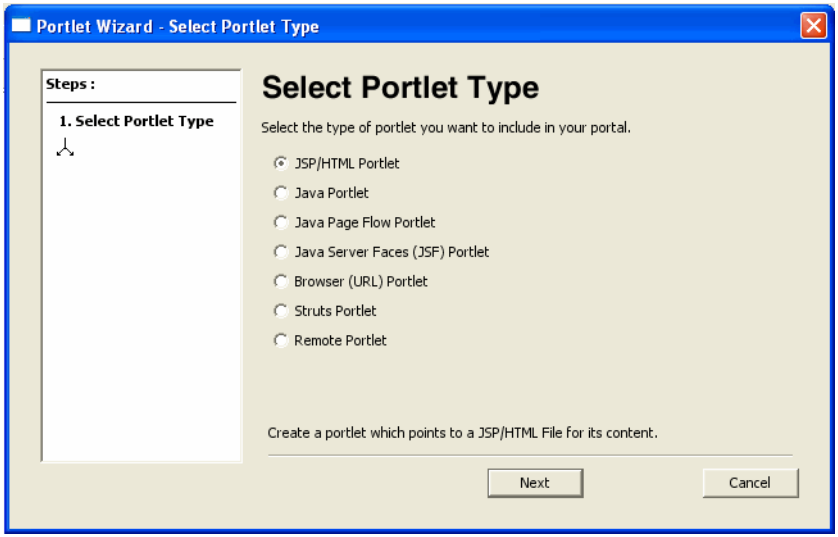
When the Portlet Wizard starts, it determines the valid portlet types to offer on the Select Portlet Type dialog, based on the type of project that you are working in.

For example, if you are working within a Portal Web Project that has only the WSRP-Producer feature (and its required accompanying features) installed, it does not have the full set of portal libraries. In this case, only the JPF, JSF, Browser, and Struts portlet types are valid selections; the other portlet types are not included in the Select Portlet Type dialog.

If no valid portlet types exist based on the project type, an informational message displays.

Figure 5-8 shows an example of the Select Portlet Type dialog.

Figure 5-8 Portlet Wizard - Select Portlet Type Dialog



Portlet Details Dialogs

The Portlet Details dialogs that display after you select a portlet type vary according to the type of portlet you are creating. The portlet-building tasks that are described in "How to Build Each Type of Portlet" on page 5-10 contain detailed information about each data entry field of the portlet details dialogs.

How to Build Each Type of Portlet

The following sections describe how to create each type of portlet supported by WebLogic Portal:

- JSP and HTML Portlets
- Java Portlets
- Java Page Flow Portlets
- JSF Portlets
- Browser Portlets

- Struts Portlets
- Remote Portlets
- Web Service Portlets

JSP and HTML Portlets

JSP portlets are very similar to simple JSP files. In most cases you can use existing JSP files to build portlets from them. JSP portlets are recommended when the portlet is simple and doesn't require the implementation of complex business logic. Also, JSP portlets are ideally suited for single page portlets.

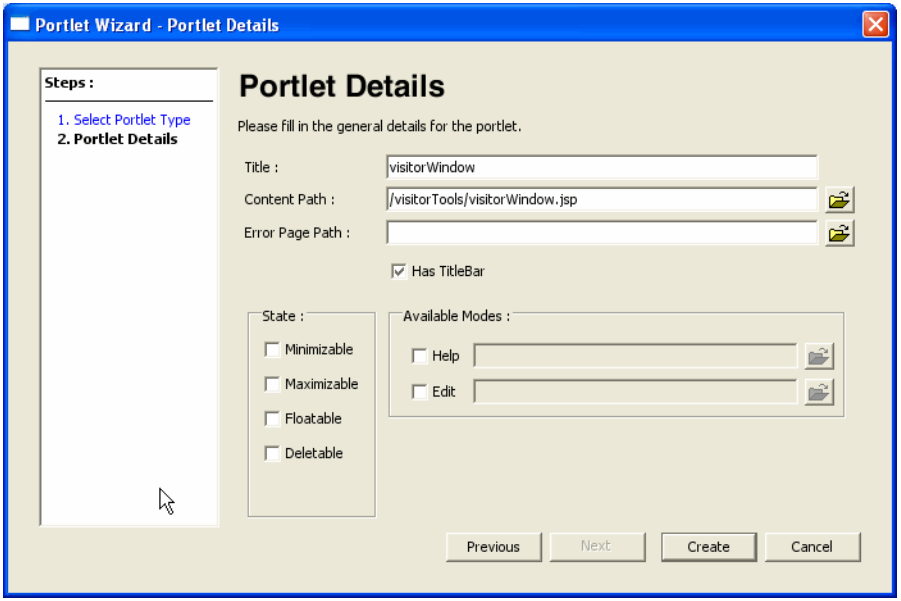
There are several ways to invoke the Portlet Wizard, as explained in the section "Starting the Portlet Wizard" on page 5-7. This description assumes that you create a portlet based on an existing JSP file.

To create a JSP portlet, follow these steps:

1. Right-click a JSP file and select **Generate Portlet** from the menu.

   The Portlet Wizard displays the Portlet Details dialog; Figure 5-9 shows an example.

Figure 5-9 Portlet Wizard - JSP Portlet Details Dialog



2. Specify the values you want for this portlet, following the guidelines shown in Table 5-1.

Table 5-1 Portlet Wizard - JSP Portlet Data Entry Fields

Field	Description
Title	The value for the Title might already be filled in.You can change the value if you wish.
Content URI	The value for the Content URI (location of the JSP) is probably already filled in. You can change this value if you wish.
Error Page URI	To designate a default error page to appear in case of an error, check the box and indicate the path to the desired URI.
Has Titlebar	If you want your portlet to have a title bar, check this box. The displayed title matches the value in the Title field. In order for a portlet to have changeable states or modes, the portlet must have a title bar.
State	Select the desired check boxes to allow the user to minimize, maximize, float, or delete the portlet. For a more detailed description of portlet states, refer to "Portlet States" on page 5-76.
Available Modes	You can enable access to Help from the portlet or you can allow the user to edit the portlet. To enable an option, select the desired check box and provide the path to the JSP page that will provide the appropriate function. For a more detailed description of portlet modes, refer to "Portlet Modes" on page 5-73.

3. Click **Create**.

   The Workshop for WebLogic window updates, adding the *Portlet_Name*.portlet file to the display tree; by default, Workshop for WebLogic places the portlet file in the same directory as the content file.

Java Portlets

Java portlets are based on the JSR 168 specification that establishes rules for portlet portability. Java portlets are intended for software companies and other enterprises that are concerned with portability across multiple portlet containers.

WebLogic Portal provides capabilities for Java portlets beyond those listed in the JSR168 spec. For example, you can set threading options, use a backing file, and so on. To implement these additional features, WebLogic Portal uses a combination of the typical `.portlet` file—which you create in the same way that you create other portlet types—as well as the standard `portlet.xml` file and the `weblogic-portlet.xml` file.

Building a Java Portlet

To create a Java portlet, follow these steps:

1. Right-click the folder where you want to store the portlet and select **New > Portlet.**

   The **New Portlet** dialog displays.

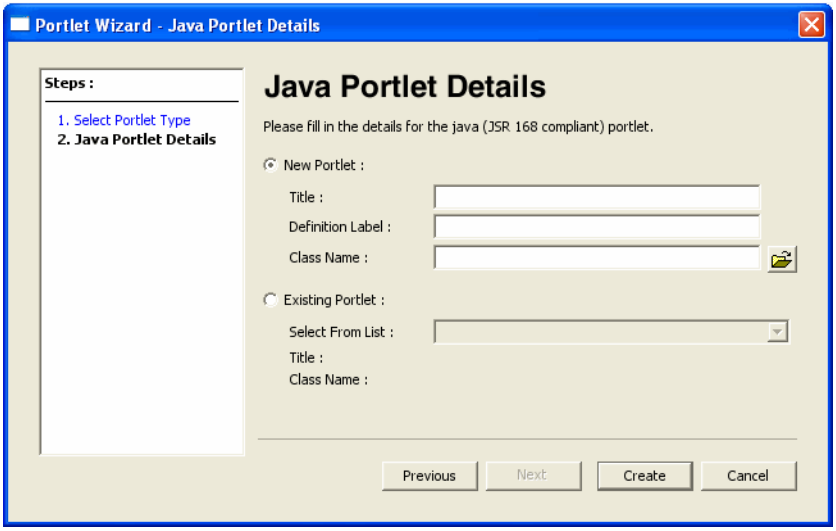
2. Enter a name for the portlet and click **Create**.

   The Portlet Wizard displays the Select Portlet Type dialog.

3. Select the Java Portlet radio button and click **Next**.

   The Java Portlet Details dialog displays. Figure 5-10 shows an example.

Figure 5-10 Portlet Wizard - Java Portlet Details Dialog



4. Identify whether you want to create a new portlet or update an existing portlet (as an entry in the `portlet.xml` file) by selecting the appropriate radio button.

If you are creating a new portlet, WebLogic Portal uses the information that you enter in the wizard to perform these two tasks:

- Create a new `.portlet` file
- Either create a new `portlet.xml` file (if this is the first Java portlet that you are creating in the project), or add an entry in the `portlet.xml` file, which is located in the WEB-INF directory.

If you choose to refer to an existing portlet in the wizard, the wizard displays a selection for every entry in the portlet.xml file, allowing you to create a new `.portlet` file and associate it with an existing entry in the portlet.xml file.

5. Specify the values you want for this portlet, following the guidelines shown in Table 5-2. All fields are required.

Table 5-2 Portlet Wizard - Java Portlet Data Entry Fields

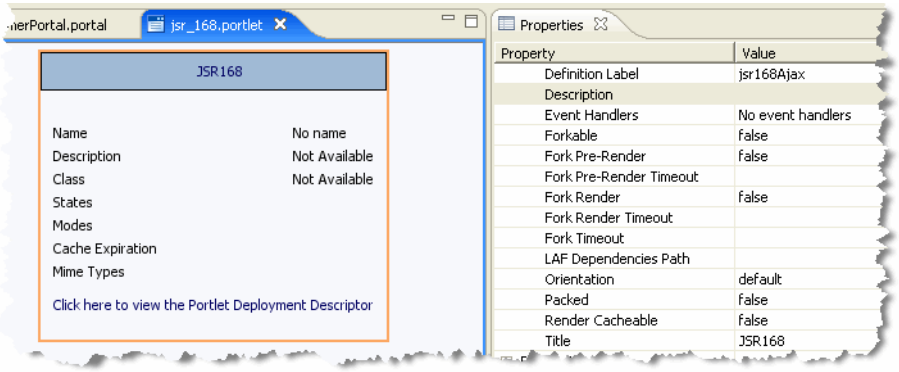
Field	Description
New Portlet – Title	The value for the Title maps to the `<title>` element in the file `portlet.xml`. The title in the `.portlet` file takes priority over the one in the `portlet.xml` file.
New Portlet – Definition Label	This value acts as the definition label for any portlet; more importantly, the value maps to the `<portlet-name>` element in the portlet.xml deployment descriptor. This value must be unique.
New Portlet – Class Name	Enter a valid class name or click **Browse** to navigate to the location of a Java class. This value maps to the `<portlet-class>` element. If you enter a javax.portlet.Portlet class that does not currently exist, the wizard will create the corresponding .java file when you click Create.
Existing Portlet – Select From List	The dropdown menu is populated from the `portlet.xml` file and contains the values from the `<portlet-name>` elements. When you select an existing portlet, the Title and Class Name display in read-only fields. **Note:** If you import an existing Java portlet into Workshop for WebLogic, you do not need to add an entry in the `web.xml` file for the WebLogic Portal implementation of the JSR-168 portlet taglib; this taglib is declared implicitly. Be sure to use `http://java.sun.com/portlet` as the taglib URI in your JSPs.

6. Click **Create**.

   Based on these values that you entered, the Wizard creates a `.portlet` file, and adds an entry to `/WEB-INF/portlet.xml`, if it already exists, or creates the file if needed.

   Workshop for WebLogic displays the newly created portlet and its current properties. Figure 5-11 shows an example of a Java portlet's appearance and properties.

Figure 5-11 Java Portlet Appearance and Properties



The `portlet-name` attribute in the `portlet.xml` file matches the `definitionLabel` property in the `.portlet` file.

After you create the portlet, you can modify its properties in the Properties view, or double-click the portlet in the editor to view and edit the generated Java class.

Note: If you delete a `.portlet` file, the corresponding entry remains in the `portlet.xml` file. You might want to clean up the `portlet.xml` file periodically; these extra entries do not cause problems when running the portal but do result in error messages in the log file.

Java Portlet Deployment Descriptor

The separate `portlet.xml` deployment descriptor file for Java portlets is located in the `WEB-INF` directory. In addition, the `weblogic-portlet.xml` file is an optional BEA-specific file that you can use to inject some additional features.

Listing 5-1 shows an example of how entries might look in the `portlet.xml` file:

Listing 5-1 Example of a portlet.xml file for a Simple Hello World Java Portlet

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

```
<portlet-app version="1.0"
   xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <portlet>
      <description>Description goes here</description>
      <portlet-name>helloWorld</portlet-name>
      <portlet-class>aJavaPortlet.HelloWorld</portlet-class>
      <portlet-info><title>Hello World!</title></portlet-info>
      <supports>
         <mime-type>text/html</mime-type>
         <portlet-mode>view</portlet-mode>
      </supports>
      <portlet-info><title>new Java Portlet</title></portlet-info>
   </portlet>
</portlet-app>
```

Packaging Java Portlets for Use on Other Systems

WebLogic Portal produces Java portlets that conform to the JSR 168 specification and can be used universally across operating systems. To package a Java portlet that you created using WebLogic Portal, use your desired packaging/archiving tool (such as the File > Export > WAR file feature in Workshop for WebLogic) to create a standard WAR file that contains the `portlet.xml` file, portlet `.class` file, and any other files the portlet needs to function. Keep in mind that these required files might include Java classes from non-WebLogic Portal JARs, any non-BEA EJBs from the application, JSPs or HTML files to handle rendering, and so on.

Customizing Java Portlets Using weblogic-portlet.xml

WebLogic Portal allows you to add more functionality to java portlets than you can obtain using the standard JSR 168 specification. You can use the optional `weblogic-portlet.xml` file to inject some additional features. The following sections provide some examples.

Floatable Java Portlets

If you want to create a floatable Java portlet, you can do so by declaring a custom state in `weblogic-portlet.xml` as shown in the following example code:

```
    <portlet>
         <portlet-name>fooPortlet</portlet-name>
         <supports>
```

```
            <mime-type>text/html</mime-type>
            <window-state>
                <name>float</name>
            </window-state>
        </supports>
    </portlet>
```

Adding an Icon to a Java Portlet

To add an icon to a Java portlet, you need to edit the `weblogic-portlet.xml` file, as described in this section.

1. Place the icon in the images directory of the skin that the portal is using. For example, if the skin name is avitek, icons must be placed in:

   ```
   myPortal/skins/avitek/images
   ```

2. In the Application panel, locate and double-click the `weblogic-portlet.xml` file to open it. This file is located in the portal's WEB-INF folder, for example:

   ```
   myPortal/WEB-INF/weblogic-portlet.xml
   ```

3. Add the following lines to the `weblogic-portlet.xml` file:

   ```
   <portlet>
     <portlet-name>myPortlet</portlet-name>
     <supports>
       <mime-type>text/html</mime-type>
       <titlebar-presentation>
         <icon-url>myIcon.gif</icon-url>
       </titlebar-presentation>
     </supports>
   </portlet>
   ```

4. Make these substitutions:
   - Change *myPortlet* to the name of the portlet that is specified in WEB-INF/portlet.xml
   - Be sure the mime-type also matches the mime-type found in WEB-INF/portlet.xml
   - Change *myIcon.gif* to the name of the icon you wish to add

Java Page Flow Portlets

You can use the Portlet Wizard to built a portlet that uses Apache Beehive Page Flows to retrieve its content.

To create a page flow portlet, follow these steps:

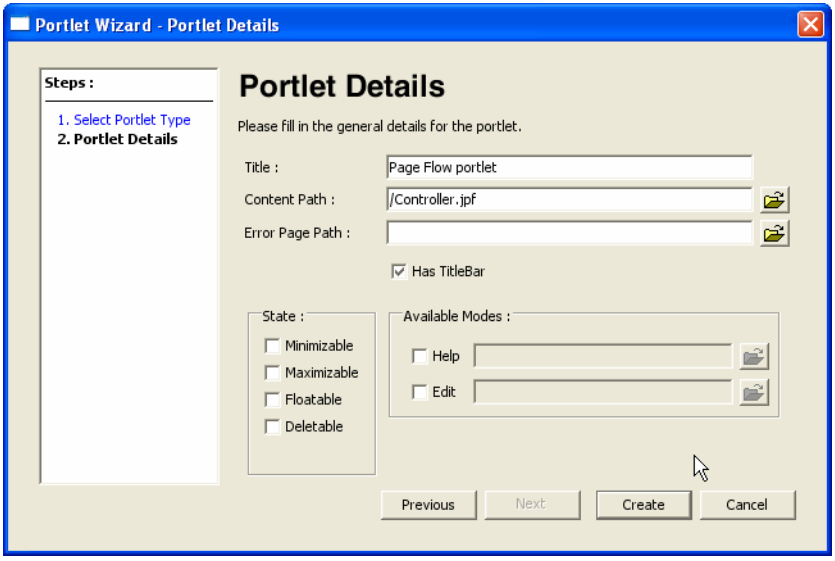
1. Right-click the folder where you want to store the page flow portlet. (The folder must be within the WebContent directory.)
2. Select **New > Portlet**.

   The **New Portlet** dialog displays.
3. Enter a name for the portlet and click **Create**.

   The Portlet Wizard displays the Select Portlet Type dialog.
4. Select the Java Page Flow Portlet radio button and click **Next**.

   The Portlet Wizard displays the Portlet Details dialog; Figure 5-12 shows an example.

Figure 5-12 Portlet Wizard - JPF Portlet Details Dialog



5. Specify the values you want for this portlet, following the guidelines shown in Table 5-3.

Table 5-3 Portlet Wizard - JPF Portlet Data Entry Fields

Field	Description
Title	The title for this portlet, which displays in the title bar if you select to include one.
Content Path	The Page Flow Request URI. You can type a value here, or click the Browse button  to open a class picker and select the appropriate class. If you use the class picker to choose a page flow class, this fully-qualified class name is converted to a URI path of a JPF. The JPF does not reside in the project, but is referred to by the `.portlet` file when the portlet is created. If you enter or navigate to a `.jpf` that has no corresponding class in the project or library modules, the Portlet Wizard creates the `.java` file for the page flow. If multiple project source directories exist, then the wizard prompts you to store the new `.java` file in the source directory of your choice. The `.java` template refers to a `.jpf` that is also created as part of this operation. The `.jpf` is created in the web content directory using the same directory structure as the package name of the new page flow class.
Error Page Path	To designate a default error page to appear in case of an error, check the box and indicate the path to the desired URI.
Has Titlebar	If you want your portlet to have a title bar, check this box. The displayed title matches the value in the Title field. In order for a portlet to have changeable states or modes, the portlet must have a title bar.
State	Select the desired check boxes to allow the user to minimize, maximize, float, or delete the portlet. For a more detailed description of portlet states, refer to "Portlet States" on page 5-76.
Available Modes	You can enable access to Help from the portlet or you can allow the user to edit the portlet. To enable an option, select the desired check box and provide the path to the JSP page or page flow that will provide the appropriate function. For a more detailed description of portlet modes, refer to "Portlet Modes" on page 5-73.

6. Click **Create**.

The Workshop for WebLogic window updates, adding the *Portlet_Name*.portlet file to the display tree; by default, Workshop for WebLogic places the portlet file in the same directory as the content file.

In order to fully understand the process of creating a page flow portlet, you should be familiar with the concept of Page Flows. For more information on using page flows with WebLogic Portal, refer to the *Portal Development Guide*.

If you want to create a page flow portlet that calls a web service, refer to "Web Service Portlets" on page 5-30.

JSF Portlets

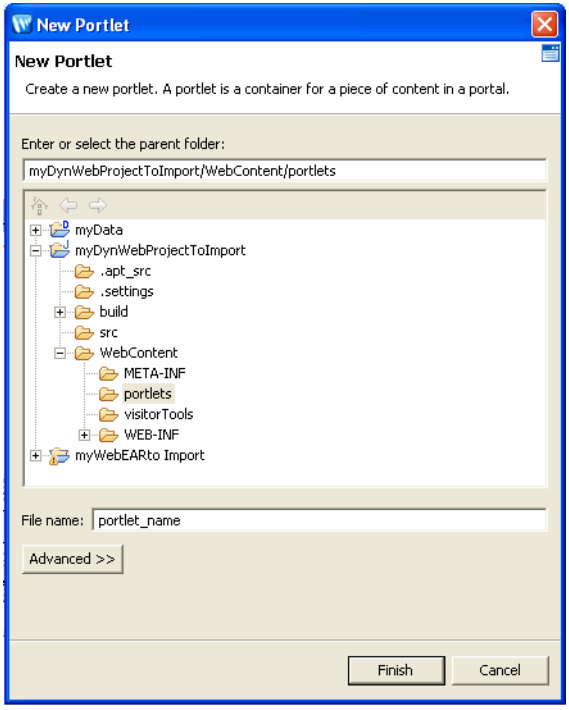
You can create JSF portlets for a WSRP producer or a framework web application that has the JSF facet installed (that is, you selected the JSF facet when you created the portal web project).

To create a JSF portlet, follow these steps:

1. Right-click in the Package Explorer view, within the web content directory, and select **New > Portlet** from the menu.

   The New Portlet dialog displays. Figure 5-15 shows an example of the New Portlet dialog.

Figure 5-13 Portlet Wizard - New Portlet Dialog



The parent folder defaults to the location from which you selected to add the portlet.

2. Edit the parent folder field if needed to indicate the project and directory for the new portlet.

   The **Finish** button is initially disabled; the button enables when you select a valid parent folder and portlet name. If you select an invalid portal project in the folder tree on this dialog, an error message appears in the status area near the top of the dialog explaining that the project is not a valid portal project.

3. Type a file name for the new portlet.

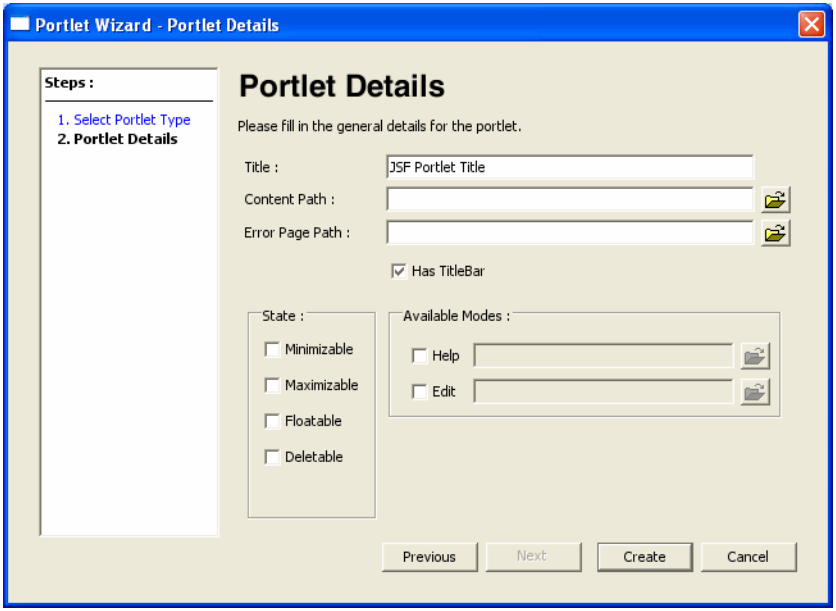
4. Click **Finish** to continue.

   The Portlet Wizard displays the Select Portlet Type dialog.

5. Click **Java Server Faces (JSF) Portlet** and then click **Next**.

   The Portlet Wizard displays the Portlet Details dialog; Figure 5-14 shows an example.

Figure 5-14 Portlet Wizard - JSF Portlet Details Dialog



6. Specify the values you want for this portlet, following the guidelines shown in Table 5-4.

Table 5-4 Portlet Wizard - JSF Portlet Data Entry Fields

Field	Description
Title	The value for the portlet title, which displays in the title bar if enabled.
Content Path	The value for the Content URI; this value should point to a JSF-enabled `.jsp` file.
Error Page Path	**Note:** Error pages are not supported with JSF portlets.
Has Titlebar	If you want your portlet to have a title bar, check this box. The displayed title matches the value in the Title field. In order for a portlet to have changeable states or modes, the portlet must have a title bar.

Table 5-4 Portlet Wizard - JSF Portlet Data Entry Fields (Continued)

Field	Description
State	Select the desired check boxes to allow the user to minimize, maximize, float, or delete the portlet. For a more detailed description of portlet states, refer to "Portlet States" on page 5-76.
Available Modes	You can enable access to Help from the portlet or you can allow the user to edit the portlet. To enable an option, select the desired check box and provide the path to the file that will provide the appropriate function. For a more detailed description of portlet modes, refer to "Portlet Modes" on page 5-73.

7. Click **Create**.

   The Workshop for WebLogic window updates, adding the *`Portlet_Name`*.portlet file to the display tree.

Placing Multiple JSF Portlets on a Portal Page

If you want to have more than one JSF portlet on a portal page, use the namingContainer JSP tag immediately after a JSF view tag, in order to provide component naming in the generated component tree. See Supporting Unique JSF Component Identifiers for an example.

Using JSPs in JSF Portlets

If you are using JSPs in your JSF portlets, be aware that you will only see your JSP edits when you view the portlet in a new session. A simple page refresh is not sufficient. This behavior differs from typical JSP development behavior, where changes are compiled and made available after a page refresh. Normally, JSPs are handled by the servlet container, which checks for updated JSPs. JSF, on the other hand, uses JSPs as a source for the component tree, which typically is loaded only once per session, depending on how the JSF implementation handles or does not handle changed JSP source. To see your JSP changes reflected in a JSF portlet, you must view the portlet in a new session. Typically, you can do this by opening a new browser to view the portal.

Supporting Unique JSF Component Identifiers

JSF applications associate a unique identifier with each JSF component in the component tree. When multiple JSF applications appear on a portal page, it becomes necessary to further scope these unique identifiers.

WLP provides the following features to support scoping JSF component identifiers on a portal page:

- The <namingContainer> tag
- The NamingContainerComponent component
- The ScopedIdBuilder class

These features are discussed in the WLP Javadoc for the com.bea.portlet.adapter.faces package.

Listing 5-2 is an example that demonstrates how to use the <namingContainer> tag. The <namingContainer> tag is described in detail in the JSP Tag Javadoc.

Listing 5-2 Using the <namingContainer> Tag

```
<%@ page language='java' contentType='text/html;charset=UTF-8'%>
<%@ taglib uri='http://java.sun.com/jsf/core' prefix='f' %>
<%@ taglib uri='http://bea.com/faces/adapter/tags-naming'
prefix='jsf-naming' %>
<%@ taglib uri='http://java.sun.com/jsf/html' prefix='h' %>
<f:view>
       <jsf-naming:namingContainer id='myPortlet'>
              <h:outputText value='Hello World'/>
</jsf-naming:namingContainer>
</f:view>
```

Browser Portlets

Browser portlets, also called Content URI portlets, are basically HTML portlets that use URLs to retrieve their content. Unlike other portlet types that are limited to displaying data contained within the portal project, browser portlets can display URL content that is outside from the portal project.

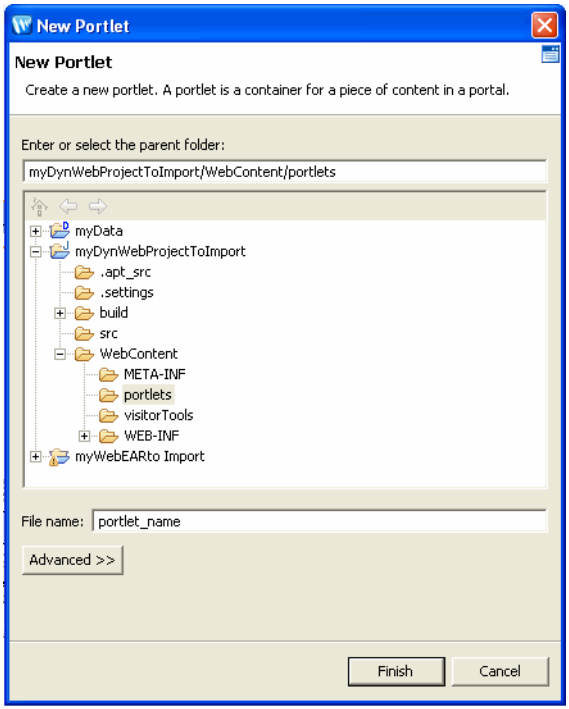
There are several ways to invoke the Portlet Wizard, as explained in the section "Starting the Portlet Wizard" on page 5-7. This description assumes that you right-click in the Package Explorer view tree within a portal project and select **New > Portlet** from the menu.

To create a browser portlet, follow these steps:

1. Right-click in the Navigation tree within a portal project and select **New > Portlet** from the menu.

The New Portlet dialog displays. Figure 5-15 shows an example of the New Portlet dialog.

Figure 5-15 Portlet Wizard - New Portlet Dialog



The parent folder defaults to the location from which you selected to add the portlet.

2. Edit the parent folder field if needed to indicate the project and directory for the new portlet.

   The **Finish** button is initially disabled; the button enables when you select a valid parent folder and portlet name. If you select an invalid portal project in the folder tree on this dialog, an error message appears in the status area near the top of the dialog explaining that the project is not a valid portal project.

3. Type a file name for the new portlet.

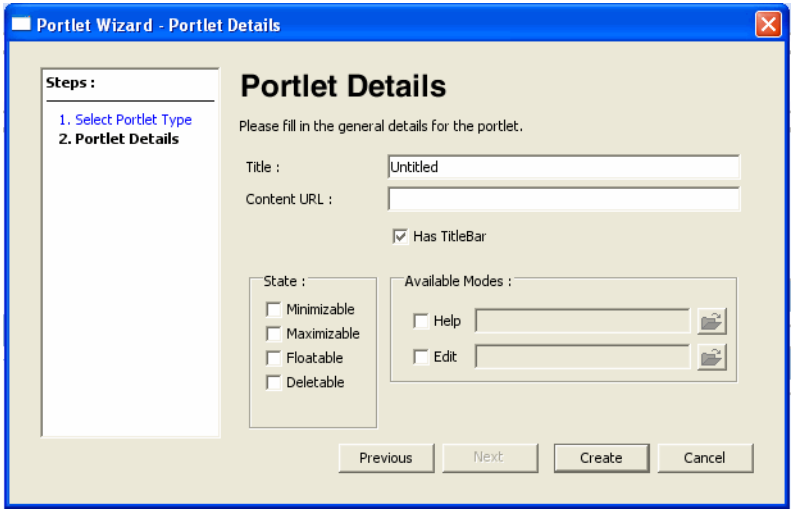
4. Click **Finish** to continue.

   The Portlet Wizard displays the Select Portlet Type dialog.

5. Click **Browser (URL) Portlet** and then click **Next**.

   The Portlet Wizard displays the Portlet Details dialog; Figure 5-16 shows an example.

Figure 5-16 Portlet Wizard - Browser Portlet Details Dialog



6. Specify the values you want for this portlet, following the guidelines shown in Table 5-5.

Table 5-5 Portlet Wizard - Browser Portlet Data Entry Fields

Field	Description
Title	The title for the portlet. This value appears in the title bar of the portlet in the editor view of the Workshop for WebLogic workbench.
Content URL	The value for the Content URL (external URL) that the portlet should use to retrieve its information. A validator checks the format of the URL that you enter, and a message notifies you if the URL is not properly formatted. You can either change the URL or ignore the warning and continue with the URL as is.
Has Titlebar	If you want your portlet to have a title bar, check this box. The displayed title matches the value in the Title field. In order for a portlet to have changeable states or modes, the portlet must have a title bar.

Table 5-5 Portlet Wizard - Browser Portlet Data Entry Fields (Continued)

Field	Description
State	Select the desired check boxes to allow the user to minimize, maximize, float, or delete the portlet. For a more detailed description of portlet states, refer to "Portlet States" on page 5-76.
Available Modes	You can enable access to Help from the portlet or you can allow the user to edit the portlet. To enable an option, select the desired check box and provide the path to the JSP page that will provide the appropriate function. For a more detailed description of portlet modes, refer to "Portlet Modes" on page 5-73.

7. Click **Create**.

   The Workshop for WebLogic window updates, adding the *Portlet_Name*.portlet file to the display tree; by default, Workshop for WebLogic places the portlet file in the same directory as the content file.

Note: The internal implementation of Browser portlets depends on asynchronous portlet content rendering; because of this, the portlet attribute Async Content displayed in the Properties view is set to none and is read-only. For more information about asynchronous content rendering, refer to "Asynchronous Portlet Content Rendering" on page 6-13.

Struts Portlets

You can use the Portlet Wizard to generate a portlet based on a Struts module.

Before you can create a Struts portlet, you must first integrate your existing Struts application into your portal web application. For detailed information on integrating Struts applications into WebLogic Portal, refer to the *Portal Development Guide*.

To create a Struts portlet, follow these steps:

1. Create or navigate to the folder that will contain the portlet file that you want to generate.
2. From the Workshop for WebLogic top-level menu, select **File > New Portlet.**

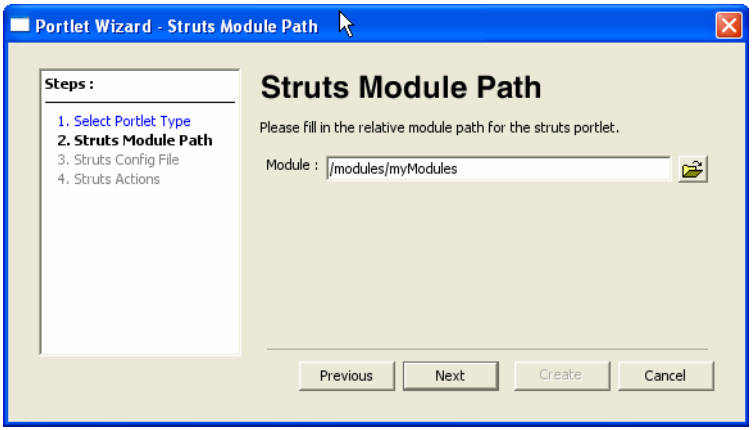
   The New Portlet Dialog displays.
3. Enter a name for the portlet, then click **Create**.

   The Portlet Wizard displays the **Select Portlet Type** dialog.

4. Select the **Struts Portlet** radio button, and click **Next**.

   The Portlet Wizard displays the Struts Module Path dialog, as shown in Figure 5-17.

Figure 5-17 Portlet Wizard - Struts Module Path Dialog

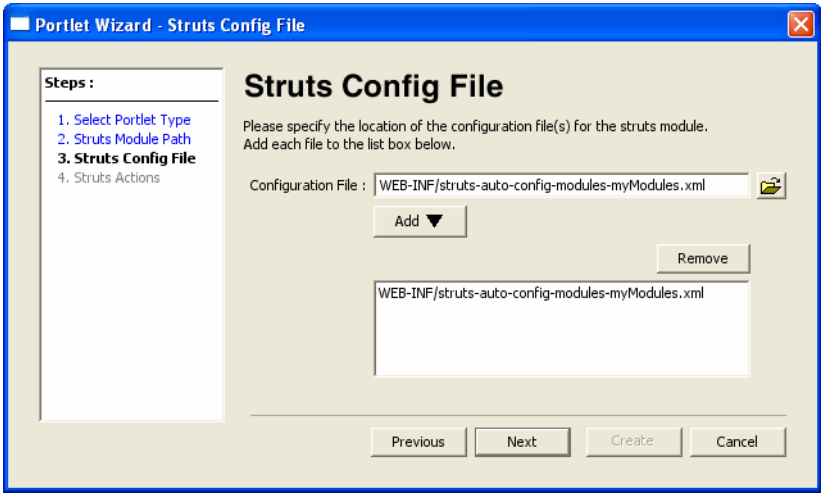


5. Specify the relative path to the struts module.

   Click **Next**.

   The Struts Config File dialog displays; an example is shown in Figure 5-18.

Figure 5-18 Portlet Wizard - Struts Config File Dialog



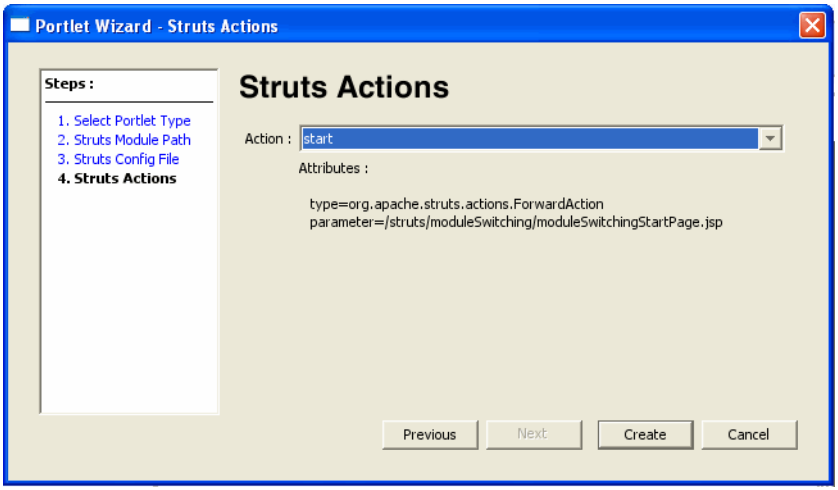
6. Type the path to, or browse to, the Struts module's XML configuration file(s); click **Add** to add each applicable configuration file.

   As a best practice, BEA recommends that you locate your configuration file(s) in the WEB-INF directory of your portal web project.

7. Click **Next**:

   The Struts Actions dialog displays, as shown in Figure 5-19.

Figure 5-19 Portlet Wizard - Struts Actions Dialog



8. Specify an action for the Struts portlet.

   The actions that appear in the drop-down menu are based on entries in the configuration file(s) that you selected in a previous step.

9. Click **Create**.

The Workshop for WebLogic window updates, adding the *`Portlet_Name`*`.portlet` file to the display tree; by default, Workshop for WebLogic places the portlet file in the directory that you specified in the Struts Module Path dialog of the wizard.

Remote Portlets

Because remote portlet development is a fundamental task in a federated portlet environment, the task of creating remote portlets is described in detail within the *BEA WebLogic Portal Federated Portals Guide*.

The following types of portlets can be exposed with WSRP inside a WebLogic portal:

- Page flow portlets
- JavaServer Pages (JSP) portlets
- Struts portlets
- Java portlets (JSR168; supported only for complex producers)
- JavaServer Faces (JSF) portlets

Web Service Portlets

A web service portlet is a special type of page flow portlet, allowing you to call a web service. You create web service portlets using the features of Workshop for WebLogic and WebLogic Portal.

Before you can create a portlet that calls a web service, you must perform the following prerequisite tasks:

1. Create a Java control from a web service.
2. Call the Java control from a page flow.

Instructions on performing these tasks are contained in the *BEA Workshop for WebLogic Programmer's Guide*.

After you have performed the setup tasks, you can create a web service portlet by following these steps:

1. In Workshop for WebLogic, navigate to the page flow that you want to use as the basis for the portlet.
2. Follow the instructions for creating a Java Page Flow portlet, as described in "Java Page Flow Portlets" on page 5-17.

Detached Portlets

WebLogic Portal supports the use of detached portlets, which provide popup-style behavior. Technically, a detached portlet is defined as anything outside of the calling portal context. Any portlet type supported by WebLogic Portal can be rendered as a detached portlet.

Considerations for Using Detached Portlets

Keep the following considerations in mind as you implement detached portlets:

- Detached portlets are never referenced from within a portal; there is no portlet instance in the portal associated with a detached portlet.
- Detached portlets can be streamed but generally cannot be entitled or customized; the library instance can be entitled, but portlet instances that are de-coupled from the portlet library cannot. For more information about library portlet instances and de-coupling, refer to the *Production Operations Guide*.
- Detached portlet are not visible or accessible from the WebLogic Portal Administration Console portlet library.
- In a streamed portal, the primary instance of the portal is used. In some cases, the primary instance cannot be determined; for example, you might have set entitlements on the primary instance to make it not viewable, or you could have set up a configuration that excludes portlets from the scanner and poller so that they are not streamed into the database. If the primary instance cannot be determined, a static version of the portlet is used (the portlet will be served in file mode). In these cases, some features related to a streamed portal (such as a community context) will not be available, and applications might be required to implement workarounds.
- Although technically a detached portlet can be implemented to use asynchronous rendering, this is not a best practice and is not recommended.
- No presentation mechanism is provided as part of the detached portlet feature; the application must define how to actually present the portlet. For example, a floated portlet will automatically be popped up in a separate window; detached portlets have no such mechanism, so your application must handle popping up the window.
- When developing detached portlets, you can place them anywhere in the hierarchy of your portal web application; the portal references the absolute path to the portlet. A good example of a detached portlet is the GroupSpace member list portlet.
- The framework for standalone portlets creates a “dummy” control tree above the portlet, including desktop, book, and page controls. The context objects associated with such controls reflect the state of the dummy controls, and not of the main control tree; for example, if a portlet tries to get information about its current book or page, the Book/Page Presentation/Backing Context objects will not reflect the actual structure of the portal. There might also be cases where the dummy control tree does not support certain backing

context APIs. When developing your portal, you need to keep this artificial control tree structure in mind.

Building Detached Portlets

You use the `standalonePortletUrl` class or associated JSP tag to create URLs to detached portlets.

To create a detached portlet URL from a JSP page, you use the render:standalonePortletUrl JSP tag or class; the following example shows the syntax of the JSP tag:

```
<render:standalonePortletUrl
portletUri="/absolute_path/detached_portlet_name.portlet" …/>
```

To create a detached portlet URL from Java code, use the following example as a guide:

```
StandalonePortletURL detachedURL =
StandalonePortletURL.createStandalonePortletURL(request, response);
detachedURL.setPortletUri(“/path/to/detached.portlet”);
```

Portlet Properties

Portlet properties are named attributes of the portlet that uniquely identify it and define its characteristics. Some properties—such as title, definition label, and content URI—are required; many optional properties allow you to enable specific functions for the portlet such as scrolling, presentation properties, pre-processing (such as for authorization) and multi-threaded rendering. The specific properties that you use for a portlet vary depending on your expected use for that portlet.

During the development phase of the portal life cycle, you generally edit portlet properties using the Workshop for WebLogic interface; this section describes properties that you can edit using Workshop for WebLogic.

During staging and production phases, you typically use the WebLogic Portal Administration Console to edit portlet properties; only a subset of properties are editable at that point. For instructions on editing portlet properties from the WebLogic Portal Administration Console, refer to “Modifying Library Portlet Properties” on page 8-3 and “Modifying Desktop Portlet Properties” on page 8-4.

For a detailed description of all portlet properties, refer to “Portlet Properties in the Portal Properties View” on page 5-35 and “Portlet Properties in the Portlet Properties View” on page 5-36.

This section contains the following topics:

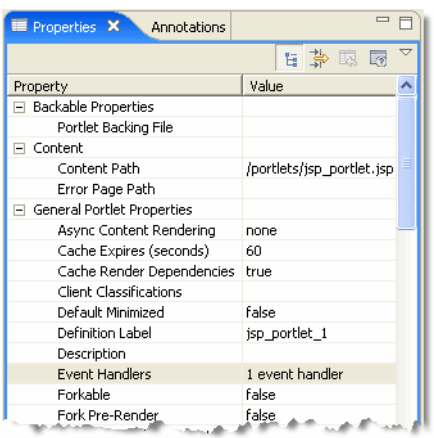
Editing Portlet Properties

To edit portlet properties, follow these steps:

1. Navigate to the location of the portlet whose properties you want to edit, and double-click the `.portlet` file to open it in the workbench editor.
2. Click the border of the desired portlet component to display the properties for that component in the Properties view.

   The displayed properties vary according to the active area that you select. If you click the outer border, properties for the entire portlet appear; if you click the inner border, properties for the content of the portlet appear, and so on.
3. Navigate to the Properties view to view the current values for the portlet properties. Figure 5-20 shows a segment of a JSP portlet's Properties view:

Figure 5-20 Editing Portlet Properties - JSP Portlet Properties View Example



4. Double-click the field that you want to change.

   If you click on a property field, a description of that field displays in the status bar.

Values for some portlet properties are not editable after you create the portlet.

In some cases, from the property field you can view associated information pertaining to that portlet property; for example, the Java portlet Class Name property contains a read-only value with an **Open** button to view the associated Java file. For more information about options available in the Properties view, refer to "Tips for Using the Properties View" on page 5-34.

Tips for Using the Properties View

The behavior of the Properties view varies depending on the type of field you are editing. The following tips might help you as you manipulate the content of the data fields in the Properties view.

- If a file is associated with a portlet property, the Properties view includes an **Open** button in addition to a **Browse** button; you can click **Open** to display the appropriate Eclipse editor/view for the file type.
- If you want to edit the XML source for a portlet, you can right-click the `.portlet` file in the Package Explorer view and choose **Edit with > XML Editor** to open the file using the basic XML editor that Eclipse provides.

  Caution: The Eclipse XML editor has limited validation capabilities. BEA recommends the use of a robust validation tool to ensure that your hand-edited XML is valid.
- The book, page, and portlet actions in the palette display properties in the Properties view when you select them in the palette. The cell editor for the content file property is read only, and includes an **Open** button; clicking **Open** displays the Eclipse editor/view for the applicable file type.
- For page flow portlets, a property editor is available for page flow content paths when displaying a page flow portlet in the editor. The property editor is a dialog cell editor that allows you to type in the URI of the page flow directly, or you can click the ellipses button  to launch the page flow class picker dialog. If you open the dialog, the page flow class name is converted to a URI when you leave the dialog. WebLogic Portal stores the URI in the `.portlet` file when you save the portlet. The property editor validates the page flow URI specified and warns you if you choose a URI that has no corresponding page flow class. You can choose to proceed anyway and store an invalid URI; you should create a valid class later so that the portlet works correctly.
- For page flow portlets, while in the portlet editor you can double-click the portlet content view to launch the corresponding Java element specified in the portlet content path. This consists of the page flow source if the source is available in the project or attached to the

JAR containing the class. If the source cannot be located, then the disassembled class browser is displayed showing the contents of the class.

- Due to a limitation in Eclipse, some long property descriptions are truncated in the Status bar. To display the entire description, while the property is highlighted click the Show Property Description button in the menu. A popup window displays the full text of the property's description. Click outside the window to close it.

Portlet Properties in the Portal Properties View

The properties described in this section are contained within the `.portal` file and are editable using the Workshop for WebLogic workbench. The values you enter here override the corresponding value in the `.portal` file, if a value exists there.

To display the portlet properties that display in the Properties view for a portal, follow these steps:

Note: These steps assume that you have an existing portal that contains portlets.

1. Double-click the `.portal` file of the portal for which you want to view portlet instance properties.

   A WYSIWYG view of the portal appears in the editor.

2. Click a portlet to highlight it.

   An orange border appears around the outside edge of the portlet.

The Properties view displays the properties of the portlet instance; Figure 5-21 shows an example.

Figure 5-21 Portlet Instance Properties in the Portal Properties View

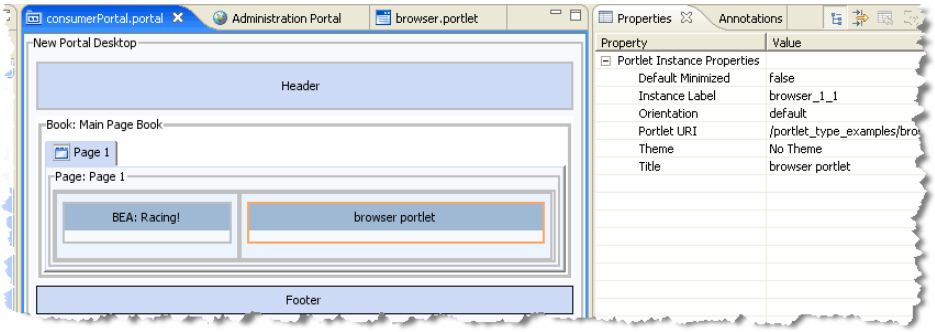


Table 5-6 describes these properties and their values.

Table 5-6 Portlet Instance Properties in the Properties View

Property	Value
Default Minimized	Optional. Select `true` for the portlet to be minimized when it is rendered. The default value is `false`. Change the value for this property only if you want to override the default value provided by the `.portlet` file.
Instance Label	Required. A single portlet, represented by a `.portlet` file, can be used multiple times in a portal. Each use of that portlet is a portlet instance, and each portlet instance must have a unique ID, or Instance Label. A default value is entered automatically, but you can change the value. Instance labels help WebLogic Portal manage the runtime state of multiple instances of portlets independently of each other on the server. WebLogic Portal also uses instance labels during URL rewriting and scoping of various HTML controls such as names of forms, and ID attributes.
Orientation	Optional. Hint to the skeleton to position the portlet title bar on the top, bottom, left, or right side of the portlet. You must build your own skeleton to support this property. The allowable values are: default, top=0, left,=1 right=2, bottom=3. Enter a value for this property only if you want to override the orientation specified in the `.portlet` file. Selecting `default` removes the orientation attribute from the portlet, book, and/or portlet instance; use this value if you want to revert to the framework default setting for this attribute.
Portlet URI	Required. The path (relative to the project) of the parent `.portlet` file. For example, if the file is stored in `Project\myportlets\my.portlet`, the Portlet URI is `/myportlets/my.portlet`.
Theme	Optional. Select a theme to give the portlet a different Look & Feel than the rest of the desktop.
Title	Enter a title if you want to override the default title specified in the `.portlet` file. The title is used in the portlet title bar.

Portlet Properties in the Portlet Properties View

The properties described in this section are contained within the `.portlet` file and are editable using the Workshop for WebLogic workbench. The values you enter here override the corresponding value in the `.portlet` file, if a value exists there.

When you select the outer border of a portlet instance in the editor, a related set of properties appears in the Properties view. The displayed properties vary according to the type of portlet that you are viewing. Figure 5-22 shows a portion of the Properties view for a portlet.

Figure 5-22 Properties View Example Showing Portlet Properties

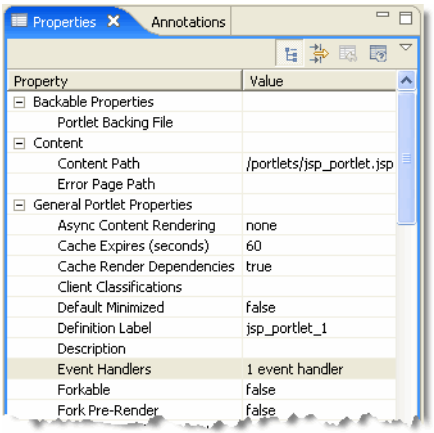


Table 5-7 describes these properties and their values.

Table 5-7 Properties in the Portlet Properties View

Property	Value
Backable Properties	
Portlet Backing File	Optional. If you want to use a class for preprocessing (for example, authentication) prior to rendering the portlet, enter the fully qualified name of that class. That class should implement the interface com.bea.netuix.servlets.controls.content.backing.JspBacking or extend com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking. From the data field you can choose to browse to a class or open the currently displayed class.
Content	
Content Path	Required. The path (relative to the project) to the file/class to be used for the portlet's content. From the data field you can choose to browse to a file (or class for page flow portlets) or open the currently displayed file/class. For example, if the content is stored in `Project/myportlets/my.jsp`, the Content URI is `/myportlets/my.jsp`.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Error Page Path	Optional. The path (relative to the project) to the JSP or HTML file to be used for the portlet's error message if the main content cannot be rendered. For example, if the error page is stored in `Project/myportlets/error.jsp`, the Content URI is `/myportlets/error.jsp`.
General Portlet Properties	
Async Content Rendering (new in Version 9.2)	Allows you to specify whether to use asynchronous content for a given portlet and the implementation to use. An editable dropdown menu provides the selections `none`, `ajax`, and `iframe`. Portlet files that do not contain the `asyncContent` attribute appear with the initial value `none` displayed. For more information, refer to "Asynchronous Portlet Content Rendering" on page 6-13.
Cache Expires (seconds)	Optional. When the `Render Cacheable` property is set to `true`, enter the number of seconds after which the portlet cache expires.
Cache Render Dependencies (new in Version 9.2)	This instance-scoped boolean property appears in the Properties view whenever a window portlet or proxy portlet is loaded, allowing render dependencies to be cached. The value defaults to `true` if the attribute is not already included in the `.portlet` file. The value is read-only for proxy portlets and editable for all other portlet types. For proxy portlets, the value is initialized from the producer whenever a proxy portlet is generated from the portlet wizard. This property does not affect posts targeted to the portlet.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Client Classifications	Optional. Select the multichannel devices on which the portlet can be viewed. The list of displayed devices is obtained from the file *Project_Path*\WEB-INF\client-classifications.xml. You must create this file to map clients to classifications in your portal web project. For more information about this task, refer to the *Portal Development Guide*. In the Manage Portlet Classifications dialog: 1. Select whether you want to enable or disable classifications for the portlet. 2. Move the client classifications you want to enable or disable from the Available Classifications to the Selected Classifications. 3. Click **OK**. When you disable classifications for a portlet, the portlet is automatically enabled for the classifications that you did not select for disabling.
Default Minimized	Required. Select `true` if you want the portlet to be minimized when it is rendered. The default value is `false`.
Definition Label	Required. Each portlet must have a unique value within the web project. For Java portlets, you type the desired value when creating the portlet; for the remaining portlet types, a value is generated automatically when you create the portlet. Definition labels can be used to navigate to portlets. Also, components must have Definition Labels for entitlements and delegated administration. As a best practice, you should edit this value in Workshop for WebLogic to create a meaningful value. This is especially true when offering portlets remotely, as it makes it easier to identify them from the producer list. **Note:** When you create a portlet instance on a desktop using the WebLogic Portal Administration Console, the generated definition label is not editable.
Description	Optional. A short text description of the portlet. The description is displayed in the Administration Console and Visitor Tools areas, and is sent from a WSRP producer where applicable.
Event Handlers	Optional. Use this value to configure interportlet communication using portlet events. The default is `No event handlers`. Click **Browse** if you want to select or add an event handler.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Forkable	Optional. Indicates whether or not the portlet can be multithread rendered. When set to `true`, a portal administrator can use the Fork Render property to make the portlet multithread rendered. The default is `false`. For more information, refer to "Portlet Forking" on page 6-3.
Fork Pre-Render	Enables forking (multi-threading) in the pre-render life cycle phase. (Refer to "How the Control Tree Affects Performance" in the *BEA WebLogic Portal Overview* for more information about the control tree life cycle.) Pre-render forking is supported by these portlet types: • JSP • Page Flow • JSR168 • WSRP (consumer portlets only) Setting Fork Pre-Render to `true` indicates that the portlet's pre-render phase should be forked.
Fork Pre-RenderTimeout (seconds)	Optional. If Fork Pre-Render is set to `true`, you can set an integer timeout value, in seconds, to indicate that the portal framework should wait only as long as the timeout value for each fork pre-render phase. The default value is -1 (no timeout). If the time to execute the forked pre-render phase exceeds the timeout value, the portlet itself times out (that is, the remaining life cycle phases for this portlet are cancelled), the portlet is removed from the page where it was to be displayed, and an error level message is logged that looks something like the following example. `<May 26, 2005 2:04:05 PM MDT> <Error> <netuix> <BEA-423350> <Forked render timed out for portlet with id [t_portlet_1_1]. Portlet will not be included in response.>`
Fork Render	Optional. Intended for use by a portal administrator when configuring or tuning a portal. Setting to `true` tells the framework that it should attempt to multithread render the portlet. This property can be set to true only if the `Forkable` property is set to `true`.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Fork Render Timeout (seconds)	Optional. If Fork Render is set to `true`, you can set an integer timeout value, in seconds, to indicate that the portal framework should wait only as long as the timeout value for each fork render portlet. The default value is -1 (no timeout). When a portlet rendering times out, an error is logged, but no markup is inserted into the response for the timed-out portlet. Selecting a value of `0` or `-1` removes the timeout attribute from the portlet; use this value if you want to revert to the framework default setting for this attribute.
Orientation	Optional. Hint to the skeleton to position the portlet title bar on the top, bottom, left, or right side of the portlet. You must build your own skeleton to support this property in the .portal file. Following are the numbers used in the .portal file for each orientation value: top=0, left=1, right=2, bottom=3. You can override the orientation in each instance of the portlet (in the Properties view).
Packed	Optional. Rendering hint that can be used by the skeleton to render the portlet in either expanded or packed mode. You must build your own skeleton to support this property. When packed=”false” (the default), the portlet takes up as much horizontal space as it can. When packed=”true,” the portlet takes up as little horizontal space as possible. From an HTML perspective, this property is most useful when the window is rendered using a table. When packed=”false,” the table's relative width would likely be set to “100%.” When packed=”true,” the table width would likely remain unset.
Render Cacheable	Optional. To enhance performance, set to `true` to cache the portlet. For example, portlets that call web services perform frequent, expensive processing. Caching web service portlets greatly enhances performance. Do not set this to true if you are doing your own caching. For more information, refer to “Portlet Caching” on page 6-2.
Required User Properties Mode	Optional. Possible values are `none`, `all`, or `specified`. If the value is `specified`, then you must enter a list of property names in the field Required User Properties Names field.
Required User Properties Names	Optional. Use this field if you entered a value of `specified` in the Required User Properties Mode field; enter a comma-delimited list of property names.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Title	Required. Enter the title for the portlet's title bar. You can override this title in each instance of the portlet (in the portal editor, as described in "Portlet Properties in the Portal Properties View" on page 5-35).
Page Flow Content	
Listen To	(Deprecated) The comma-separated list of instance labels of the portlets whose actions should also be called in the selected page flow portlet. This functionality has been replaced with the more complete interportlet communication mechanism.
Page Flow Action	Optional. The initial action to be executed in a page flow. If not specified, the `begin` action is used.
Page Flow Refresh Action	Optional. The action to be executed in the page flow when the page is refreshed but the portlet is not targeted. This is equivalent to using portlet event handlers configured on the `onRefresh` portal event to invoke the page flow action.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Request Attribute Persistence	Optional. Possible values are `none`, `session`, and `transient-session`. This attribute controls attribute persistence for Page Flow, JSF, and Struts portlets. The default is `session`, where request attributes populated by an action are persisted into a collection class that is placed into a session attribute so that the portal framework can safely include the forwarded JSP on subsequent requests without re-running the action. Using the value `session` can cause session memory consumption and replication that would not otherwise occur in a standalone Page Flow, JSF, or Struts application. The value `transient-session` places a serializable wrapper class around a HashMap into the session. The value `none` performs no persistence operation. JPF or Struts portlets that have the `transient-session` value applied generally have the same behavior as existing portlets; however, in failover cases, the persisted request attributes disappear on the failed-over-to server. In the failover case, you must write forward JSPs to handle this contingency gracefully by, at a minimum, not expecting any particular request attribute to be populated; ideally you should include the ability to either repopulate automatically or present the user with a link to re-run the last action to repopulate the request attributes. For non-failover cases, request attributes are persisted, providing a performance advantage for non-postback portlets identical to default `session` persistence portlets. Portlets that have the `none` value applied will never have request attributes available on refresh requests; you must write forward JSPs to assume that they will not be available. You can use this option to completely remove the framework-induced session memory loading for persisted request attributes.
Java Server Faces (JSF) Content	
Request Attribute Persistence	Refer to the description in the **Page Flow Content** section.
Portlet Properties	

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Content Presentation Class	A CSS class that overrides any default CSS class used by the component's skeleton. For proper rendering, the class must exist in a cascading style sheet (CSS) file in the Look and Feel's selected skin, and the skin's skin.xml file must reference the CSS file. **Sample:** If you enter "my-custom-class", the rendered HTML from the default skeletons looks like this: `<div class="my-custom-class">` The properties you enter are added to the component's parent tag. This property also applies to books and pages. For more information, refer to the *Portal Development Guide*.
Content Presentation Style	Optional. The primary uses are to allow content scrolling and content height-setting. For scrolling, enter the following attributes: • overflow:auto – Enables vertical and horizontal scrolling For setting height, enter the following attribute: • height:*200px* where *200px* is any valid HTML height setting. You can also set other style properties for the content as you would using the Presentation Style property. The properties are applied to the component's content/child tag.
Offer as Remote	Optional. Defines whether the portlet is accessible using the WSRP producer. The default is `true`, which allows the portlet to be accessed. For more information about entitling remote portlets, refer to the *Federated Portals Guide*.
JSP Content	
Content Backing File	Optional. If you want to use a backing file for content prior to rendering the portlet, enter the fully qualified name of the appropriate class. That class should implement the interface com.bea.netuix.servlets.controls.content.backing.JspBacking or extend com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Thread Safe	Optional. Performance setting for books, pages, and portlets that use backing files. When Thread Safe is set to `true`, an instance of a backing file is shared among all books, pages, or portlets that request the backing file. You must synchronize any instance variables that are not thread safe. When Thread Safe is set to `false`, a new instance of a backing file is created each time the backing file is requested by a different book, page, or portlet.
Portlet Title Bar	
Can Delete	Optional. If set to `true` the portlet can be deleted from a page.
Can Float	Optional. If set to `true` the portlet can be floated into a separate window. For instructions on creating a floatable Java portlet, which requires editing the weblogic-portlet.xml file, in "Customizing Java Portlets Using weblogic-portlet.xml" on page 5-16.
Can Maximize	Optional. If set to `true` the portlet can be maximized.
Can Minimize	Optional. If set to `true` the portlet can be minimized.
Edit Path	Optional. The path (relative to the project) to the portlet's edit page.
Help Path	Optional. The path (relative to the project) to the portlet's help file.
Icon Path	Optional. The path (relative to the project) to the graphic to be used in the portlet title bar. You must create a skeleton to support this property.
Mode Properties (available when you add a mode to a portlet)	
Content Path	Required. The path (relative to the project) to the JSP, HTML, or .jpf file to be used for portlet's mode content, such as the edit page. For example, if the content is stored in Project/myportlets/editPortlet.jsp, the Content URI is /myportlets/editPortlet.jsp. Although a Browse button appears for this property, if you want to point to a page flow you must manually enter the path to the `.jpf`.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Error Path	Optional. The path (relative to the project) to the JSP, HTML, or .jpf file to be used for the error message if the portlet's mode page cannot be rendered. For example, if the error page is stored in Project/myportlets/errorPortletEdit.jsp, the Content URI is /myportlets/errorPortletEdit.jsp. Although a Browse button appears for this property, if you want to point to a page flow you must manually enter the path to the `.jpf`.
Portlet Backing File	Optional. If you want to use a class for preprocessing (for example, authentication) prior to rendering the portlet's mode page (such as the edit page), enter the fully qualified name of that class. That class should implement the interface com.bea.netuix.servlets.controls.content.backing.JspBacking or extend com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking.
Visible	Optional. Makes the mode icon (such as the edit icon) in the title bar or menu invisible (`false`) or visible (`true`). Set Visible to `false` when, for example, you want to provide an edit URL in a desktop header.
Mode Toggle Button Properties	
Name	Optional. Displayed when you select an individual mode. An optional name for the mode, such as `Edit`.
Presentation Properties	
Presentation Class	This property is described in the *Portal Development Guide*.
Presentation ID	This property is described in the *Portal Development Guide*.
Presentation Style	This property is described in the *Portal Development Guide*.
Properties	Optional. A comma-delimited list of name-value pairs to associate with the object. This information can be used by skeletons to affect rendering.
Skeleton URI	This property is described in the *Portal Development Guide*.
Proxy Portlet Properties	
Connection Establishment Timeout	Optional. The number of milliseconds after which this portlet will time out when establishing an initial connection with its producer.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
Connection Timeout	Optional. The number of milliseconds after which this portlet will time out when communicating with its producer, after the physical connection has been established. If not specified here, the default value contained in the file `WEB-INF/wsrp-producer-registry.xml` is used.
Group ID	Optional. This value is assigned by the producer and is not editable. Portlets with the same Group ID from the same producer can share sessions. The Group ID value is meaningful only to the producer and not manipulated by WebLogic Portal.
Invoke Render Dependencies (new in version 9.2)	This boolean property allows the consumer to obtain render dependencies from the producer during the pre-render life cycle of a proxy portlet. When a portlet on a producer has a `lafDependenciesUri` value, the producer exposes the `invokeRenderDependencies` boolean in the portlet description. For more information on this attribute, refer to "Portlet Dependencies" on page 5-70. **Note:** Provide an absolute path for the `lafDependenciesUri` attribute, rather than a relative path. The value defaults to `false` if the attribute is not included in the `.portlet` file. The value is read-only, and is initialized from the producer whenever a proxy portlet is generated from the portlet wizard.
Portlet Handle	Required. The producer's unique identifier for the portlet that this proxy references. The value is not editable.
Producer Handle	Required. The producer's unique identifier.
Templates Stored in Session	Indicates whether the producer stores URL templates in the user's session on the producer side. This boolean is meaningful only when URL Template Processing boolean is set to `true`.
URL Template Processing	Indicates whether the producer uses URL templates to create URLs. If true, the consumer supplies URL templates. If false, the producer rewrites URLs using special rewrite tokens.

Table 5-7 Properties in the Portlet Properties View (Continued)

Property	Value
User Context Stored In Session (new in version 9.2)	Required. The purpose of this value is to cut down on network traffic by sending the user's context (including the profile) only once per session. For WebLogic Portal producers it will always be `true`. For third party producers it can be `true` or `false`, depending on the response from GetServiceDescription. If it is `false`, the entire user context will be sent on every getMarkup and performBlockingInteraction request. If `true` it will be sent only once per producer session. This boolean value defaults to `false` if the attribute is not included in the `.portlet` file. The value is read-only, and is initialized from the producer whenever a proxy portlet is generated from the portlet wizard.
Struts Content	
Listen To	(Deprecated) Allows this portlet to invoke an action when another portlet invokes the same action. This functionality has been replaced with the more complete interportlet communication mechanism. For more information on interportlet communication, refer to Chapter 7, "Local Interportlet Communication."
Request Attribute Persistence	Refer to the description in the **Page Flow Content** section.
Struts Action	The begin action that this struts portlet should invoke on the first request to the portlet.
Struts Module	The struts module that is associated with this struts portlet. A "struts module" is a means of scoping a particular set of struts actions to a group called a module, which generally maps to a single subdirectory of web resources and a separate `struts-config.xml file`.
Struts Refresh Action	Optional. The action to be performed in the struts module when the page is refreshed but the portlet itself is not targeted.
Uri Content (Browser portlet properties)	
Content Url	Required. The content control takes a URI that is expected to be a URL for a standalone application or web page, and embeds the URL as portlet content.

Portlet Preferences

Portlet preferences provide the primary means of associating application data with portlets. This feature is key to personalizing portlets based on their usage. This section describes portlet preferences in detail.

After you create a portlet, you can instantiate it several times. Because you can create several instances of a portlet, it is natural to expect each instance to behave differently yet use the same code and user interface. For instance, consider a typical portlet to display a Stock Portfolio. Given a list of stock symbols, this portlet retrieves quotes from a stock quote web service periodically, and displays the quotes in the portlet window. By letting each user change the list of stock symbols and a time interval to reload the quote data, you can let each user customize this portlet.

The portlet needs to be able to store the list of stock symbols and the retrieval interval persistently, and update these values whenever a user customizes these values. In particular, the following data must be persistently managed:

- Default Values – Your portlet may specify a default list of stock symbols and a reasonable retrieval interval. These values are applicable to all usages of the portlet no matter who the user is. The user could even be anonymous.
- Customized Values – Your portlet also needs to be able to store these values when a user updates the values for a given portlet instance. Note that your portlet should also scope this data to an instance, such that other instances of this portlet are not affected by this customization.

Technically, a portlet preference is a named piece of string data. For example, a Stock Portfolio portlet could have the following portlet preferences:

- A preference with name “stockSymbols” and value “BEAS, MSFT”
- Another preference with name “refreshInterval” and value “600” (in seconds).

You can associate several such preferences with a portlet. WebLogic Portal provides the following means to manage portlet preferences:

- Specify portlet preferences during the development phase

  When you are building a portlet using the Workshop for WebLogic workbench, you can specify the names and default values of preferences for each portlet. All portlet instances derived from this portlet will, by default, assume the values specified during development.

- Let administrators modify portlet preferences

WebLogic Portal allows portal administrators to modify preferences for a given portlet instance.This task occurs during the staging phase and uses the WebLogic Portal Administration Console.

- Let portlets access and modify preferences at request time

  At request time, your portlets can programmatically access and update preferences using a `javax.portlet.PortletPreferences` object. You can create an edit page for your portlet to let users update preferences, or you can automatically update preferences as part of your normal portlet application flow.

This section contains the following topics:

Specifying Portlet Preferences

The steps to associate preferences with a portlet depend on the type of portlet you are building. If you are using the Java Portlet API, described in "Getting and Setting Preferences for Java Portlets Using the Preferences API" on page 5-56, the steps follow those specified in the Java Portlet Specification. For other kinds of portlets, such as those using Java Page Flows, Struts, or JSPs, you can use the Workshop for WebLogic workbench to add preferences to a portlet.

You can also allow the administrator to create new preferences using the Administration Console. However, because the portlet developer is more likely to be aware of how portlet preferences are used by the portlet, it is generally better to create portlet preferences during the development phase.

Specifying Preferences for Java Portlets in the Deployment Descriptor

For portlets using Java Portlet API, you can specify preferences in the portlet deployment descriptor according to the specification. For all portlets in a web project, the deployment descriptor is `portlet.xml`, found in the `WEB-INF` directory of the web project. Listing 5-3 provides an example.

Listing 5-3 Specifying Portlet Preferences in portlet.xml with a Single Value

```
<portlet>
   <description>This portlet displays a stock portfolio.</description>
   <portlet-name>portfolioPortlet</portlet-name>
   <portlet-class>portlets.stock.PortfolioPortlet </portlet-class>
      <supports>
         <mime-type>text/html</mime-type>
         <portlet-mode>edit</portlet-mode>
      </supports>
      <portlet-info>
         <title>My Portfolio</title>
      </portlet-info>
      <portlet-preferences>
         <preference>
            <name>stockSymbols</name>
               <value>BEAS, MSFT</value>
         </preference>
         <preference>
            <name>refreshInterval</name>
            <value>600</value>
         </preference>
   </portlet-preferences>
</portlet>
```

This snippet deploys the portfolio portlet with two preferences: a preference with name stockSymbols and value BEAS, MSFT, and another preference refreshInterval with value 600.

Instead of specifying a single value for the stockSymbols preference, you can declare each symbol as a separate value as shown in Listing 5-4 below, with the value elements shown in bold.

Listing 5-4 Specifying Portlet Preferences with Values Specified Separately

```
<portlet>
   <description>
      This portlet displays a stock portfolio.
```

```
   </description>
   <portlet-name>portfolioPortlet</portlet-name>
   <portlet-class>portlets.stock.PortfolioPortlet </portlet-class>
   <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>edit</portlet-mode>
   </supports>
   <portlet-info>
      <title>My Portfolio</title>
   </portlet-info>
   <portlet-preferences>
      <preference>
         <name>stockSymbols</name>
         <value>BEAS</value>
         <value>MSFT</value>
         </preference>
      <preference>
         <name>refreshInterval</name>
         <value>600</value>
      </preference>
   /portlet-preferences>
</portlet>
```

If you prefer that portlets should not be allowed to programmatically update any given preference, you can mark the preference as read-only. Listing 5-5 shows an example of preventing a portlet from changing the refreshInterval.

Listing 5-5 Specifying a Read-Only Portlet Preference Value

```
<portlet>
   <description>
      This portlet displays a stock portfolio.
   </description>
   <portlet-name>portfolioPortlet
   <portlet-class>portlets.stock.PortfolioPortlet
   <supports>
```

```
      <mime-type>text/html</mime-type>
      <portlet-mode>edit</portlet-mode>
   </supports>
   <portlet-info>
      <title>My Portfolio</title>
   </portlet-info>
   <portlet-preferences>
      <preference>
         <name>stockSymbols</name>
         <value>BEAS</value>
         <value>MSFT</value>
      /preference>
      <preference>
         <name>refreshInterval</name>
         <value>600</value>
         <read-only>true</read-only>
      </preference>
   </portlet-preferences>
</portlet>
```

Note that by marking a preference read-only, you are preventing the portlet from changing the current value only at request time. Portal administrators can always change the value(s) of a preference using the Administration Console.

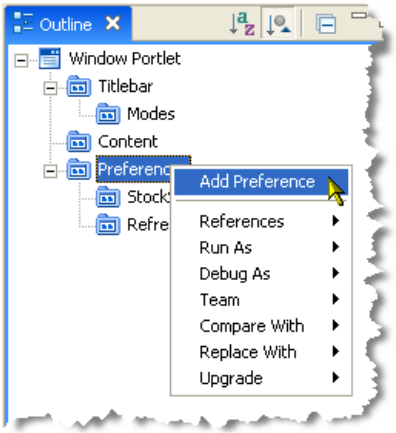
Specifying Preferences for Other Types of Portlets using Workshop for WebLogic

If you are building other kinds of portlets (such as those using Java Page Flows, Struts, or simple JSPs), you can add preferences using Workshop for WebLogic.

To add a preference, follow these steps:

1. Click to select the portlet for which you want to add a preference.
2. In the Outline view for the portlet, right-click **Preferences** and in the context menu click **Add Preference**. Figure 5-23 shows an example of the preferences context menu.

Figure 5-23 Portlet Preferences Context Menu



A new preference is added to the tree hierarchy with the name **New Preference** Preference.

3. Click the new item to display its properties in the Properties view.
4. Edit the values in the Properties view. Figure 5-24 shows an example of the fields in the Properties view.

Figure 5-24 Portlet Preferences Properties View

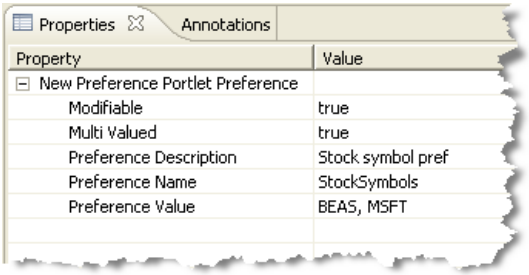


Table 5-8 describes the attributes for portlet preferences as shown in the Properties view.

Table 5-8 Portlet Preference Properties

Field	Value
Modifiable	Indicates whether the preference is read-only or can be modified by the user. The default is `true`.
Multi Valued	Indicates whether the preference can have multiple values. The default is `true`. To specify multiple values for a preference, create multiple preferences with the same name.
Description	A brief description of the preference.
Name	Name of the preference.
Value	Each preference can have one or more values. Each value is of type java.lang.String.

Using the Preferences API to Access or Modify Preferences

At request time, portlet preferences for a given portlet are represented as instances of the `javax.portlet.PortletPreferences` interface. This interface is part of the Java Portlet API. This interface specifies methods to access and modify portlet preferences.

Getting Preferences Using the Preferences API

Table 5-9 describes methods that allow a portlet to access its preferences.

Table 5-9 Methods that Allow a Portlet to Access its Preference Values

Method	Purpose
`String getValue(String name, String default)`	Use this method to get the first value of a preference.
`String[] getValues(String name, String[] defaults)`	Use this method to get all the values of a preference.
`boolean isReadOnly(String name)`	Use this method to determine whether a given preference is read-only.

Table 5-9 Methods that Allow a Portlet to Access its Preference Values (Continued)

Method	Purpose
`Enumeration getNames()`	Use this method to get an enumeration of the names of all preferences.
`Map getMap()`	Use this method to get a map of preferences. The keys in this map are the names of all the preferences, and the values are the same as those returned by getValues(String name, String[] defaults)

Setting Preferences Using the Preferences API

Table 5-10 describes methods that allow a portlet to change preference values.

Table 5-10 Methods that Allow a Portlet to Change Preference Values

Method	Purpose
`void setValue(String name, String value)`	Use this method to set the value of a preference
`void setValues(String name, String[] values)`	Use this method to set several values for a preference
`void store()`	Use this method to commit the changes made to preferences for a portlet.
`void reset(String name)`	Use this method to reset the value of a preference to its default, or remove the preference if there is no default

After modifying preferences by calling setValue(), setValues() and reset() methods, you must call store() explicitly to make the changes permanent; otherwise, changes will not be made permanent.

Getting and Setting Preferences for Java Portlets Using the Preferences API

For portlets written using the Java Portlet API, you can obtain an instance of `javax.portlet.PortletPreferences` object from the incoming portlet request – `javax.portlet.RenderRequest` within the `processAction()` method, or `javax.portlet.ActionRequest` within the `render()` method.

In Listing 5-6, the portlet displays a form to edit the current values of portlet preferences in a JSP page included from the `doEdit()` method of the portfolio portlet.

Listing 5-6

```
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet"%>
<%@ page import="javax.portlet.PortletPreferences" %>

<portlet:defineObjects/>

<%
   PortletPreferences prefs = renderRequest.getPreferences();
   String refreshInterval = prefs.getValue("refreshInterval", "600");
   String symbols = prefs.getValue("stockSymbols", "BEAS, MSFT");
%>

<form method="POST" action="">
   <table>
      <tr>
         <td>Symbols</td><td><input name="symbols"
         value="<%=symbols>"/></td>
      </tr>
      <tr>
         <td>Refresh Interval</td><td><input name="refreshInterval"
         value="<%=refreshInterval>"/></td>
      </tr>
      <tr>
         <td></td>
         <td><input type="submit" value="Submit"/></td>
      </tr>
   </table>
</form>
```

The portlet updates the preferences in its `processAction()` method, as shown in Listing 5-7.

Listing 5-7 Portlet Updates the Preferences in the processAction() Method

```
public class PortfolioPortlet extends GenericPortlet
{
   {
      public void doEdit(RenderRequest renderRequest, RenderResponse
      renderResponse)
      throws IOException, PortletException
      {
         ...
      }
      public void processAction(ActionRequest actionRequest, ActionResponse
      actionResponse)
      throws PortletException
      {
         String refreshInterval =
         actionRequest.getParameter(“refreshInterval”);
         String symbols = actionRequest.getParameter(“stockSymbols”);

         PortletPreferences prefs = actionRequest.getPreferences();
         prefs.setValue(“refreshInterval”, refreshInterval);
         prefs.setValue(“stockSymbols”, symbols);
         try
         {
            prefs.store();
         }
         catch(SecurityException se) {
         // Thrown when the user does not have enough privileges to store
         // preferences. Make sure that the user logged into the portal.
         ...
         }
      catch(catch(IOException ioe) {
      // There is an error storing preferences
      ...
      }
   }
}
```

During `processAction()`, this portlet uses the `javax.portlet.ActionRequest` object to obtain preferences.

Getting and Setting Portlet Preferences Using the API for Other Portlet Types

Portlet preferences can be accessed and updated from other kinds of portlets too. The main difference is in the way your portlets obtain an instance of the `javax.portlet.PortletPreferences` object.

- Before rendering, portlets can use `com.bea.netuix.servlets.controls.portlet.PortletBackingContext` to obtain portlet preferences; for example, in a page flow action, or in the `handlePostbackData()` method of the backing file associated with the portlet.
- During the render phase portlets can use `com.bea.netuix.servlets.controls.portlet.PortletPresentationContext` to obtain portlet preferences; for example, in a JSP associated with a page flow.

Both these classes provide a method `getPreferences()` that takes `javax.servlet.HttpServletRequest` as an argument and return an object of type `javax.portlet.PortletPreferences`.

JSP Tags for Getting Portlet Preferences

WebLogic Portal provides a JSP tag library for setting up portlet preferences. Table 5-11 describes the applicable JSP tags.

Table 5-11 JSP Tags for Getting Portlet Preferences

Method	Purpose
`getPreference`	Use this tag to get the value of a portlet preference.
`getPreferences`	Use this tag to get all the values of a portlet preference. This tag can also used to write multiple values to the output separated by a separator.
`forEachPreference`	Use this tag to iterate through all the preferences of a portlet. You can nest other tags (getPreference, getPreferences, ifModifiable and Else) inside this tag.

Table 5-11 JSP Tags for Getting Portlet Preferences

Method	Purpose
`ifModifible`	Use this tag to include the body of this tag if the given portlet preference is not read-only.
`else`	Use this tag in conjunction with the ifModifiable tag to include the body of this tag if the given portlet preference is read-only

For more information on the Java classes associated with these tags, refer to the *Javadoc*.

Portlet Preferences SPI

In WebLogic Portal, the framework includes a default implementation that manages portlet preferences in the built-in PF_PORTLET_PREFERENCE and PF_PORTLET_PREFERENCE_VALUE database tables. If desired, you can replace this implementation with your own.

You can use the Portlet Preferences SPI to allow portal applications to manage portlet preferences outside framework-managed database tables. For example, you can store preferences along with other application data in another back-end system or a different set of database tables.

When propagating a portal, the preferences SPI participates in the propagation process. When you exporting data for the propagation, the SPI is called to obtain the preferences, and when you are importing data, the SPI is called to store the preferences.

The following sections describe how to use the Portlet Preferences SPI.

Implement the SPI

You specify the SPI using the interface `com.bea.portlet.prefs.IPreferenceAppStore`. An implementation of this class must be deployed as a EJB jar file.

Listing 5-8 provides an example.

Listing 5-8 Implementing the SPI Using the Interface IPreferencesAppStore

```
public interface IPreferenceAppStore extends EJBObject
{
   /**
```

```
    * Returns preferences for a portlet entity with the given uniqueId.
    *
    * The returned java.util.Map contains
    * com.bea.netuix.application.prefs.Preference
    * objects keyed against their names.</p>
    *
    * @param uniqueId unique ID
    * @return preferences
    */
    public Map getPreferences(PortletPreferencesId uniqueId) throws
    RemoteException, PreferenceAppStoreException;

    /**
    * Writes the preferences to the underlying persistence.
    *
    * This method should be implemented to be atomic. That is, the
    * implemenation should guarantee that either all preference
    * values are persisted or none at all.
    *
    * The java.util.Map argument should contain
    * com.bea.netuix.application.prefs.Preference
    * objects keyed against their names.
    *
    * @param uniqueId unique ID
    * @param preferences preferences
    */
    public void storePreferences(PortletPreferencesId uniqueId,
    Map preferences) throws RemoteException, PreferenceAppStoreException;

    /**
    * Clear all preferences for the given unique ID from the
    * underlying persistence store.
    *
    * @param uniqueIds unique IDs
    */
    public void removePreferences(PortletPreferencesId[] uniqueIds) throws
    RemoteException, PreferenceAppStoreException;
}
```

Using the SPI

To cause the framework to use a new SPI in place of the default SPI, you must update the EJB named `PreferencePersistenceManager` in the `ejb-jar.xml` file within `netuix.jar`. The

value `BEA_netuix.DefaultStore` must be changed to the name of the SPI EJB as specified in its deployment descriptor (`ejb-jar.xml`). The value `com.bea.portlet.prefs.provider.DefaultStoreHome` must be changed to the home interface of the SPI implementation.

Caution: To edit the `ejb-jar.xml` file you need to copy the J2EE library resources into your project. Keep in mind that with future updates to the WebLogic Portal product, you might have to perform manual steps in order to incorporate product changes that affect those resources.

The code segment in Listing 5-9 shows the default entries, which you must change to use the SPI.

Listing 5-9 Example Code Showing Default Entries that Must be Changed

```
<session>
   <ejb-name>PreferencePersistenceManager</ejb-name>
   <home>com.bea.portlet.prefs.PreferencePersistenceManagerHome</home>
   <remote>com.bea.portlet.prefs.PreferencePersistenceManager</remote>
   <ejb-class>com.bea.portlet.prefs.PreferencePersistenceManagerImpl
   </ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>
   <env-entry>
      <env-entry-name>prefs-spi-jndi-name</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>BEA_netuix.DefaultStore</env-entry-value>
   </env-entry>
   <env-entry>
      <env-entry-name>prefs-spi-home-class-name</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>com.bea.portlet.prefs.provider.DefaultStoreHome
      </env-entry-value>
   </env-entry>
<!-- Snip -->
</session>
```

Best Practices in Using Portlet Preferences

Desktop Testing of Portlet Preferences

In order to view and test the preferences that you have created, you must use a desktop view from the WebLogic Portal Administration Console rather than Workshop for WebLogic's **Open on Server** view.

Portlets accessed from `.portal` files cannot store preferences. If you update a preference using a `.portal` file, your portlet encounters a `java.lang.UnsupportedOperationException` error.

Users Must be Authenticated

You must provide a means for users to log in before they can update preferences; users who are updating portlet preferences must first be authenticated. If an anonymous user attempts to update a portlet, a `java.lang.SecurityException` error occurs.

Note that portlets can always get portlet preferences whether or not the user is anonymous or whether the portlet is accessed via a `.portal` file.

Do Not Store Arbitrary Data as Preferences

It is tempting to store arbitrary application data as portlet preferences. For example, if you have a portlet that allows users to upload and store documents on the server, it might seem appropriate to store those documents as portlet preferences. This is not a good practice. The purpose of portlet preferences is to associate some *properties* for a portlet instance without having to be aware of any implementation-specific portlet instance IDs. These properties allow customization of the portlet's behavior. The underlying implementation of portlet preferences is not designed for storing arbitrary application data.

The following steps outline an alternative implementation that can meet the needs of the portlet:

Perform setup steps:

1. Add a preference to your portlet. This preference acts as the primary key to your portlet's application data. Assign a default value for this preference.
2. Create tables in your database to store application data with the value of the preference as the primary key.

Set up preferences in your portlet:

1. When you want to associate application data with the current portlet instance, check the value of the preference. If the value is the default, generate a new value (for example, using a sequence number generator), and set this as the value of the preference, and store the preference.
2. If the value of the preference is not the default, you do not need to generate a new value.
3. Store your application data using the value of the preference as the primary key.

This procedure ensures that your application data is always scoped to portlet instances.

Do Not Use Instance IDs Instead of Preferences

The portal framework maintains instance identity using internally generated instance IDs. Portlets can access their instance IDs using `getInstanceId()` methods on `com.bea.netuix.servlets.controls.portlet.PortletPresentationContext` and `com.bea.netuix.servlets.controls.portlet.PortletBackingContext`.

Storing data directly in the database using portlet instance IDs does not work, for the following reasons:

- The portal framework generates instance IDs, and portlets have no control over when and how those instance IDs are generated.
- Instance IDs might change at any time without the portlet's knowledge. For example, as the user or administrator customizes a desktop using Visitor Tools or the Administration Console, the framework can create new instances or change the instance ID of a portlet. If the instance ID changes, your portlet cannot load the data from your database; the primary key has changed without your portlet being aware of it.

Backing Files

The most common means of influencing portlet behavior within the control life cycle is to use a portlet backing file. A portlet backing file is a Java class that can contain methods corresponding to portal control life cycle stages, such as init() and preRender(). A portlet's backing context, an abstraction of the portlet control itself, can be used to query and alter the portlet's characteristics. For example, in the init() life cycle method, a request parameter might be evaluated, and depending on the parameter's value, the portlet backing context can be used to specify whether the portlet is visible or hidden. For more information about backing contexts, refer to the *Portal Development Guide*.

Backing files can be attached to portals either by using Workshop for WebLogic or coding them directly into a `.portlet` file.

Backing files are simple Java classes that implement the `com.bea.netuix.servlets.controls.content.backing.JspBacking` interface or extend the `com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking interface` abstract class. The methods on the interface mimic the controls life cycle methods (refer to "How Backing Files are Executed" on page 5-65) and are invoked at the same time the controls life cycle methods are invoked.

The following portal controls support backing files:

- Desktops
- Books
- Pages
- Portlets
- JspContent controls

The interportlet communication example in Chapter 7, "Local Interportlet Communication" uses backing files.

This section contains the following topics:

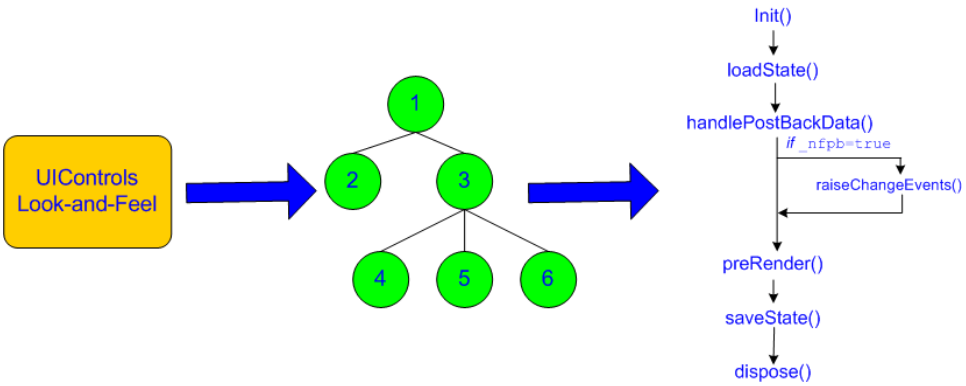
How Backing Files are Executed

All backing files are executed before and after the JSP is called. In its life cycle, each backing file calls these methods:

- `init()`
- `handlePostBackData()`
- `preRender()`
- `dispose()`

Figure 5-25 illustrates the life cycle of a backing file.

Figure 5-25 Backing File Life Cycle



On every request, the following sequence occurs:

Note: In the following steps, the methods are called unless items on inactive pages have been "optimized away" if tree optimization is enabled. For example, if tree optimization is enabled and items on an inactive page are not included on the resulting partial control tree, then the method is not called.

1. All `init()` methods are called on all backing files in depth-first order (that is, in the order they appear in the tree). This method is called whether or not the control (the portal, page, book, or desktop) is on an active page.
2. If the `_nfpb` parameter is set to true, all `handlePostbackData()` methods are called.
   - If the `_nfpb` parameter is set to true in the request parameter of any called `handlePostbackData()` methods, `raiseChangeEvents()` is called. This method causes events to fire, which is necessary if the backing file tries to make any state or mode changes.

Tip: You can use the method `AbstractJspBacking.isRequestTargeted(request)` to determine if a request is for a particular portlet.

   - If the backing file's `handlePostbackData()` method returns true, the `raiseChangeEvents()` method is called.
3. All `preRender()` methods are called for all portal framework controls on an active (visible) page.

4. The JSPs are called and rendered on the active page.
5. The `dispose()` method is called on each backing file.

Thread Safety and Backing Files

A new instance of a backing file is created per request, so you do not have to worry about thread safety issues. New Java VMs are specially tuned for short-lived objects, so this is not the performance issue it was in the past. Also, `JspContent` controls support a special type of backing file that allows you to specify whether or not the backing file is thread safe. If this value is set to `true`, only one instance of the backing file is created and shared across all requests.

Scoping and Backing Files

The difference between having a backing file as part of `<netuix: portlet backingfile =some_value>` or part of `<netuix: jspContent backingfile=some_value>` is related to scoping.

For example, if you have the backing file on the portlet itself, you can actually stop the portlet from rendering. If the backing file is at the jspContent level, the portlet portion of the control tree has already run; you use this implementation to run processes that are specifically for the JSP in the portlet.

Backing File Guidelines

Follow these guidelines when creating a backing file:

- Ensure `netuix_servlet.jar` is included in the in the project classpath; otherwise, compilation errors occur.
- When implementing the `init()` method, avoid any heavy processing.

Listing 5-10 shows an example backing file.In this example, the `AbstractJspBacking` class is extended to provide the backing functionality required by the portlet. The example uses a session attribute because of the volatility of the HTTPRequest object; BEA recommends that you pass data between life cycle methods using the session rather than the request object.

Listing 5-10 Backing File Example

```
package backing;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import com.bea.netuix.events.Event;
import com.bea.netuix.events.CustomEvent;
import
com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking;

public class ListenCustomerName extends AbstractJspBacking
{
   public void listenCustomerName(HttpServletRequest request,
   HttpServletResponse response, Event event)
   {
      CustomEvent customEvent = (CustomEvent) event;
      String message = (String) customEvent.getPayload();
      HttpSession mySession = request.getSession();
      mySession.setAttribute("customerName", message);
   }
}
```

Adding a Backing File Using Workshop for WebLogic

You can add a backing file to a portlet either from within Workshop for WebLogic or by coding it directly into the file to which you are attaching it. Simply specify the backing file in the **Backing File** field of the Properties view, as shown in Figure 5-26. You need to specify the backing directory and, following a dot-separator, *only* the backing file name. Do not include the backing file extension; for example enter this:

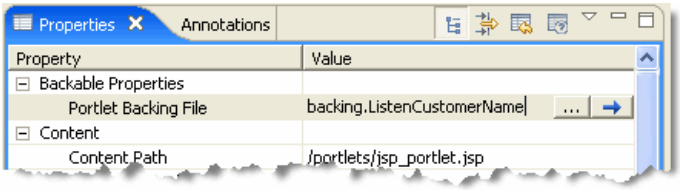
```
backing.ListenCustomerName
```

Not this:

```
backing.ListenCustomerName.java
```

For the preceding example, if you include the file extension, the application interprets it as the file name—because the file path is specified by a dot-separator—and looks for a non-existent file called `java` in a non-existent directory called `ListenCustomerName`.

Figure 5-26 Adding a Backing File Using Workshop for WebLogic



Adding the Backing File Directly to the .portlet File

To add the backing file by coding it into a `.portlet` file, use the `backingFile` parameter within the `<netuix:jspContent>` element, as shown in Listing 5-11.

Listing 5-11 Adding a Backing File to a .portlet File

```
<netuix:content>
   <netuix:jspContent
   backingFile="portletToPortlet.pageFlowSelectionDisplayOnly.menu.
      backing.MenuBacking"
   contentUri="/portletToPortlet/pageFlowSelectionDisplayOnly/menu/
      menu.jsp"/>
</netuix:content>
```

Portlet Appearance and Features

Some aspects of portlet appearance are controlled by default at the portal level, such as colors, layouts, and themes. Appearance/rendering characteristics and portlet-specific features include the use of title bars and associated states (minimize, maximize, float, and delete) and modes that affect portlet content (edit mode, help mode, and custom modes).

The following sections describe how to work with portlet-specific appearance/content features and modes:

- Portlet Dependencies
- Portlet Modes
- Portlet States

- Portlet Title Bar Icons
- Portlet Height and Scrolling

Portlet Dependencies

In a rendered HTML page, the proper place to include most types of resources, such as script files or style sheet references, is in the header of the document. Portlets sometimes need to specify resources that are required for rendering the portlet in the page. In the past, methods for making required elements available on the page included placing elements into the skeleton, which is not recommended because this creates a coupling between the skeleton and the portlet; or putting references directly in the portlet content, leading to the possibility of creating invalid HTML.

The problem was exacerbated in a federated (WSRP) environment because remote portlets are potentially included in several places and there was no way for one of these portlets to indicate that it relies on, for example, a piece of a CSS that resides in an external file.

WebLogic Portal now provides an explicit way to handle this requirement, using the portlet dependencies feature.

The configuration of a Look & Feel has significantly changed in WebLogic Portal Version 9.2. The concepts related to skin and skeleton resource dependencies are now more formally known as *render dependencies* and *script dependencies*. Typical examples of such dependencies are CSS files and JavaScript files.

Both skins and skeletons can now specify such dependencies as well as associated search paths to be used for resolving these dependencies. Additionally, mechanisms exist to eliminate redundancy and to provide a reliable ordering for dependencies related to skins, skeletons, and theme skin and skeletons. These same capabilities are now available for portlets as well as portals, so that a portlet can specify necessary dependencies in a standards-compliant way; you identify these dependencies using appropriate elements located in the head section of the rendered page. The other advantages of the Look & Feel dependencies framework are also realized at a portlet level, such as reliable ordering and redundancy elimination.

This section contains the following topics:

- Identifying Portlet Dependencies
- Considerations and Limitations
- Creating a Dependency File

Identifying Portlet Dependencies

The configuration of portlet dependencies shares the same mechanisms as the standard Look & Feel—you use an XML configuration document conforming to a standard Look & Feel schema. This XML document is referenced from a `.portlet` file using an attribute on the portlet element.

As with a Look & Feel's render dependencies, you can resolve a portlet's render dependencies utilizing a set of application search paths. Additionally, the search paths of the Look & Feel skin, or any appropriate Theme skin, are used before the portlet's own search paths to resolve a portlet's render dependencies.

You can specify a portlet's dependencies configuration file in the Workshop for WebLogic Properties view by entering the value in LAF Dependencies Path field. Alternatively, you can add the attribute `lafDependenciesUri` to the portlet element in a `.portlet` file, as shown in the following example:

```
<netuix:portlet definitionLabel="myPortlet" title="My Portlet"
lafDependenciesUri="/portlets/example/myPortlet.dependencies">
```

By convention, you should adhere to the following guidelines when setting up a portlet's dependencies configuration file:

- Give the file the same name as the `.portlet` file.
- Assign the file a `.dependencies` extension.
- Locate the file at the same level in the file hierarchy as the `.portlet` file.

Although the guidelines listed here are not required, deviating from them can lead to unexpected behavior. For more information, refer to "Considerations and Limitations" on page 5-72.

The portlet dependencies configuration file uses standard types from the standard Look & Feel schemas and looks similar to the example shown in Listing 5-12.

Listing 5-12 Portlet Dependencies Configuration File Example

```
<?xml version="1.0" encoding="UTF-8"?>
<p:window
xmlns:p="http://www.bea.com/servers/portal/framework/laf/1.0.0";
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance";
xsi:schemaLocation="http://www.bea.com/servers/portal/framework/laf/1.0.0
laf-window-1_0_0.xsd ">
   <p:render-dependencies>
```

```
      <p:html>
        <p:links>
          <p:search-path>
            <p:path-element>.</p:path-element>
          </p:search-path>
          <p:link rel="stylesheet" type="text/css" href="my.css"/>
        </p:links>
      </p:html>
    </p:render-dependencies>
</p:window>
```

The configuration file shown in Listing 5-12 causes a CSS file to be included in the rendered page output (as a link element in the HTML head section). First, the search occurs for the CSS file relative to the Look & Feel or Theme skin search paths for the links element. If the CSS file is not found, then the search path in the configuration file is used. Relative search paths use the directory of the configuration file as a base.

The default behavior is to look first in the Look & Feel or Theme–specified search paths. This behavior allows a Look & Feel/Theme the ability to properly skin portlet resources. However, portlet-level resources should not be placed in the Look & Feel/Theme directories. If a situation arises when you do not want to use this behavior, you can disable it by specifying a value of `false` for the `use-skin-paths` attribute on the `render-dependencies` element.

Considerations and Limitations

At this time, Workshop for WebLogic does not providing editing capabilities for portlet render dependencies configuration files; you can use the included Eclipse-based XML file editor for this purpose.

BEA recommends that you not share a single `.dependencies` file across several portlets. Although WebLogic Portal does not prevent this usage, sharing a single file might lead to confusion when coordinating updates to the file later.

Creating a Dependency File

You can use Workshop for WebLogic to create a valid dependency file that you can then complete using Workshop's XML editor.

1. Select **File** > **New** > **Other**.

2. In the New dialog, open the XML folder and select **XML**. The New XML File wizard opens.
3. Choose **Create XML From XML Schema File** and click **Next**.
4. Enter a name for the XML file in the XML File Name dialog and click **Next**.
5. In the Select XML Schema File dialog, choose **Select XML Catalog Entry** and in the Key column select `laf-window-1_0_0.xsd` as the schema. Click **Next**.
6. In the Select Root Element dialog, choose the root element **window**.
7. Optionally check the boxes that add optional attributes/elements to your new XML file.
8. Click **Finish**.
9. Rename the generated file's extension from `.xml` to `.dependencies`.

Portlet Modes

All portlets created with WebLogic Portal support the use of modes. Modes allow you to affect the end user's ability to edit the portlet or display Help for the portlet. You add icon buttons to a portlet's title bar to indicate the availability of a mode.

The following pre-defined modes exist for WebLogic Portal:

- **Edit** – Lets you specify a custom file that lets users modify the portlet's content when they click the Edit button.
- **Help** – Lets you specify a custom file that shows users help content for the portlet when they click the Help button.

You can also create your own custom portlet modes using WebLogic Portal.

Buttons for the selected modes appear in the portlet's title bar. Figure 5-27 shows an example of the default buttons for the portlet modes when displayed in the editor; Figure 5-28 shows the appearance of the mode icons in a running portlet.

Figure 5-27 Portlet Mode and State Buttons in Editor

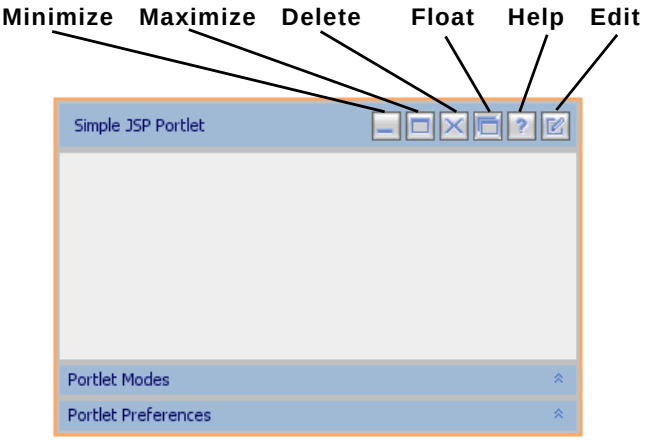
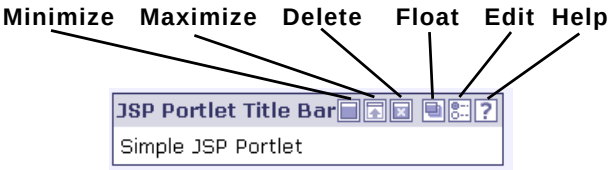


Figure 5-28 Portlet Mode and State Buttons in a Running Portlet



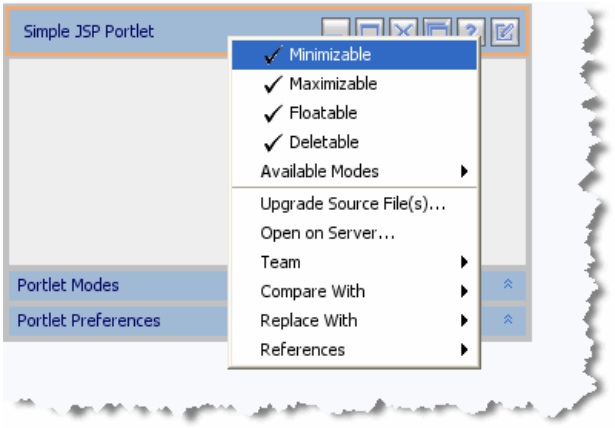
When you use the Portlet Wizard to create a portlet, mode and state settings are available on the Portlet Details dialog. These settings can also be edited in the portlet's Properties view: The following sections describe possible methods of performing these tasks.

Adding or Removing a Mode for an Existing Portlet

To add or remove the Help or Edit mode from the title bar, follow these steps:

1. Display the portlet for which you want to add or remove a mode.
2. Right-click the title bar of the displayed portlet to display the context menu. Figure 5-29 shows an example of the title bar context menu.

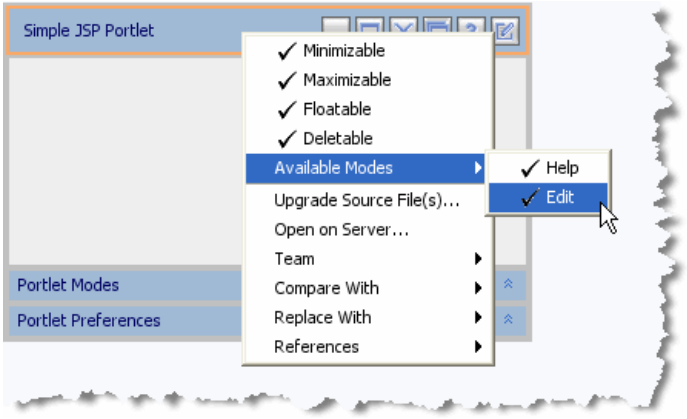
Figure 5-29 Available Portlet Modes - Title Bar Context Menu



3. Click **Available Modes**.

   Checkmarks on the submenu indicate the available modes for this portlet, which were determined when you created it. Figure 5-30 shows an example of the submenu.

Figure 5-30 Portlet Mode - Available Modes Submenu



4. Click the mode for which you want to change the availability status. For example, in Figure 5-30, the Help mode is checked (available); when you click Help, the Help button disappears from the title bar.

5. Select **File > Save** to save your changes.

Properties Related to Portlet Modes

You can view and edit the mode's property details in the Properties view. For example, you can edit the Portlet Backing File property if you want to perform preprocessing before rendering the portlet's mode page (such as the edit page).

To display the mode properties for the portlet, click the expand/contract toggle button in the Portlet Mode area of the portlet. Edit mode properties and Help mode properties display in the Properties view.

For descriptions of the mode properties, refer to Table 5-7.

Portlet States

States determine the end user's ability to affect the rendering of a portlet. WebLogic Portal supports these portlet states:

Normal – the typical rendered appearance of the portlet.

- **Minimize** – Collapses the portlet, leaving only the title bar, when the user clicks the **Minimize** button.
- **Maximize** – Makes the portlet take up the entire desktop area (not including the desktop header and footer) when the user clicks the **Maximize** button.
- **Float** – Displays the portlet in a popup window when the user clicks the Float button.
- **Delete** – Removes the portlet from the desktop when the user clicks the **Delete** button.

When you use the Portlet Wizard to create a portlet, state and mode settings are available on the Portlet Details dialog. These settings can also be edited in the portlet's Properties view: The following sections describe possible methods of performing these tasks.

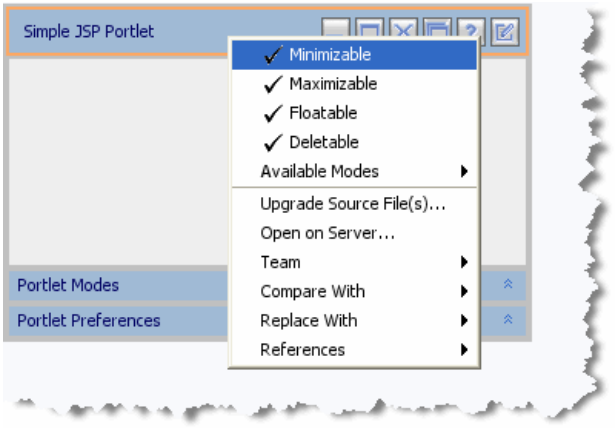
Modifying Portlet States in Workshop for WebLogic

You can select which of the states you want to include with the portlet by following these steps:

1. Right-click the portlet title bar.

   A context menu showing applicable states appears. Figure 5-31 shows an example of the title bar context menu showing all states as available.

Figure 5-31 Portlet State - Title Bar Context Menu



2. Click to select the state that you want to change.

   Selecting a state adds it to the portlet, while deselecting the state removes it from the portlet. For example, in Figure 5-31, all four states are selected, and appear in the title bar.

   If you click to deselect **Deletable**, the Delete button on the portlet disappears.

3. Select **File > Save** to save your changes.

Minimizing or Maximizing a Portlet Programmatically

You can minimize or maximize a portlet either in the portlet file or in a portlet's backing file. The actual code is the same for both. Here is an example of maximizing a (Java page flow) portlet:

```
PortletBackingContext context =
PortletBackingContext.getPortletBackingContext(request);
context.setupStateChangeEvent(WindowCapabilities.MAXIMIZED.getName());
```

You can put this code in an `action` method of the Java page flow or in the `handlePostbackData` method of the backing file. When using the backing file, in order to get the handlePostbackData method to be called, you must have `'_nfpb=true'` in the URL.

These mechanisms do not work if asynchronous content rendering is enabled for the portlet.

Portlet Title Bar Icons

The default state and mode icons used in portlet title bars are stored in the wlp-lookandfeel-web-lib library module; you can view them in Merged Projects view in the various subdirectories of `framework/skins`.

Portlet Height and Scrolling

All portlets created with WebLogic Portal support height and scrolling.

- **Height** affects the portlet's displayed height on the portlet page.
- **Scrolling** affects whether or not the portlet is scrollable.

You can control the height of portlets and determine whether or not their contents scroll.

Portlet height and scrolling is controlled by the following CSS style attributes:

- `overflow: auto` – Enables vertical and horizontal scrolling
- `height:` *`200px`* (where 200px is any valid HTML setting)

You can set these attributes on a portlet that is open in the workbench editor.

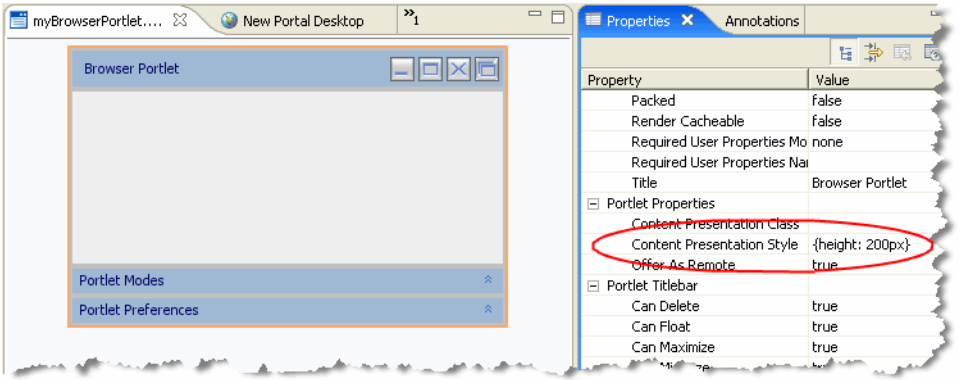
To set these properties, follow these steps:

1. Open a portlet in the workbench editor.
2. Click the outer border of the portlet to display the portlet properties in the Properties view.
3. In the Properties view, set one of the following properties:
   - **Presentation Style** - Enter any of the previously listed attributes for this property. You can use overflow and height. Separate the values with a semicolon.
   - **Presentation Class** - Enter the name of a style sheet class that contains the height or scrolling attributes that you want to use.
   - **Content Presentation Style** - Enter any of the previously listed attributes for this property. You can use overflow and height. Separate the values with a semicolon.
   - **Content Presentation Class** - Enter the name of a style sheet class that contains the height or scrolling attributes that you want to use.

   Note: The distinction between Presentation Style and Content Presentation Style, for example, is the location where the styling is applied (portlet or content). The use of one or the other depends on the specifics of what the specific styling is trying to accomplish.

   Figure 5-32 shows an example of a height property, set using Content Presentation Style.

Figure 5-32 Portlet Height and Scrolling Presentation Properties Example



Based on the entries shown in Figure 5-32, the result looks similar to the example in Figure 5-33.

Figure 5-33 Portlet Height and Scrolling—Portlet Appearance Results

If you use the Presentation Class property instead of the Presentation Style property, you must have the corresponding style class defined in a CSS file.

For example, if you use the value .**portlet-scroll** in the **Content Presentation Class** field, you must have the following style class definition already set up in your CSS file:

```
.portlet-scroll
{
   overflow:auto;
   height:250px;
}
```

4. Select **File** > **Save** to save your changes.

Making All Portlets Scroll

To provide portlet height and scrolling automatically, you can specify an additional rule for the standard portlet content CSS class. For example, you can do one of the following:

- Add a `<style>` element to the `skin.xml` file for your Look & Feel containing this rule:

```
.bea-portal-window-content
{
     height: 250px;
     overflow: auto;
}
```

- Alternatively, you can place the above rule in a custom CSS file and create a `<style>` or `<link>` element in the `skin.xml` file that references the custom CSS file.

For more information on portal skins, themes, and skeletons, refer to the *Portal Development Guide*.

Getting Request Data in Page Flow Portlets

A page flow stores information in the requests. If you have a portal page with multiple page flow portlets, you need a way for each page flow to individually store and retrieve that information. For example, the request object for a page might have a variable *car_type*, with a value of `x`. When the page flow runs, it obtains this value and uses it in some way. If you have another page flow portlet with a *car_type* value of `z`, and if only one request exists for the whole page, the two page flow portlets might interfere with each other. To prevent this problem, WebLogic Portal essentially makes a copy of the outer (portal) request to make separate *scoped* requests, one for each portlet. This gives each page flow portlet its own unique request to use to store its information.

In some cases, you might want to use information that is stored at the outer request rather than within the scoped request.

For example, if you use regular HTML tags within Netui form tags, you might have something similar to this:

```
<netui:form action="myAction">
   <input type="check box" name="test"/>
   <netui:button value="myAction"></netui:button>
</netui:form>
```

Based on the tags used above, you might typically use a regular `getParameter` request like this:

```
<%request.getParameter("test")%>
```

However, to get that HTML input value from the outer request, use the following:

```
<%@page import="org.apache.beehive.netui.pageflow.scoping.ScopedServletUtils"%>
   <%
   HttpServletRequest outerRequest = ScopedServletUtils.getOuterRequest
   ( request );
   %>
   test: <%=outerReq.getParameter("test")%>
```

JSP Tags and Controls in Portlets

WebLogic Portal provides JSP tags that you can use within JSPs. When you use the JSP Design Palette view in Workshop for WebLogic, you can view available JSP tags and then drag them into the Source View of your JSP, and use the Properties view to edit elements of the code.

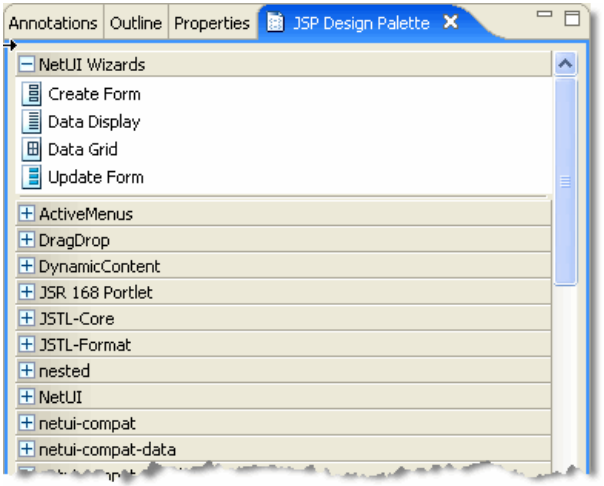
WebLogic Portal also provides custom Java controls that make it easy for you to quickly add pre-built modules to your portal; custom Java controls exist for event management, Visitor Tools, Community management, and so on. For example, most user management functionality can be easily exposed with a User Manager Control on a page flow.

Note: The term control is also used to refer to the portal (netuix) framework controls, such as desktop, book, page, and so on. These controls are referred to in the text as *portal framework controls.*

Viewing Available JSP Tags

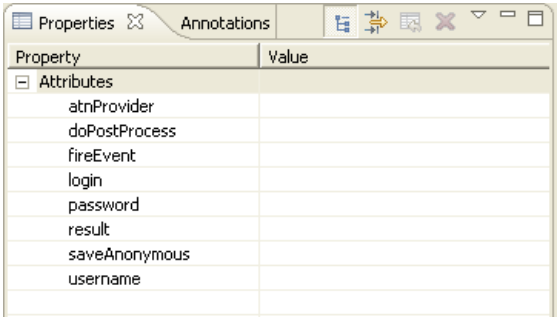
When you open a JSP in Workshop for WebLogic, you can use the JSP Design Palette (**Window > Show View > JSP Design Palette**) to display all the JSP tags currently loaded and available; Figure 5-34 shows a portion of the display.

Figure 5-34 JSP Design Palette Showing Available JSP Tags



To use a tag, drag it into the editor, use the Source View to edit the code directly, and use the Properties view to set properties, as shown in Figure 5-35:

Figure 5-35 Dragging a JSP Tag into the Design View – Properties for Add User JSP Tag



For information about the Java class associated with each JSP tag, refer to the *Javadoc*.

Viewing Available Controls

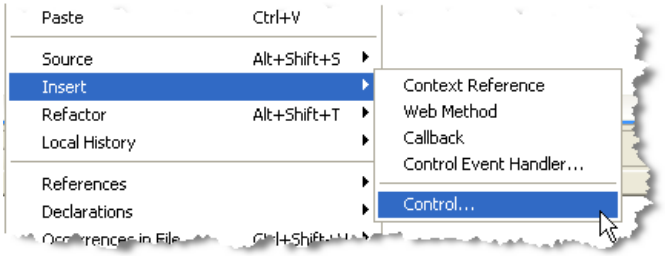
To view the available custom controls provided by WebLogic Portal when viewing a page flow:

1. Open an existing page flow (.jpf file) or create a new page flow.

For information about creating page flows using Workshop for WebLogic, refer to the *BEA Workshop for WebLogic Platform Programmer's Guide*.

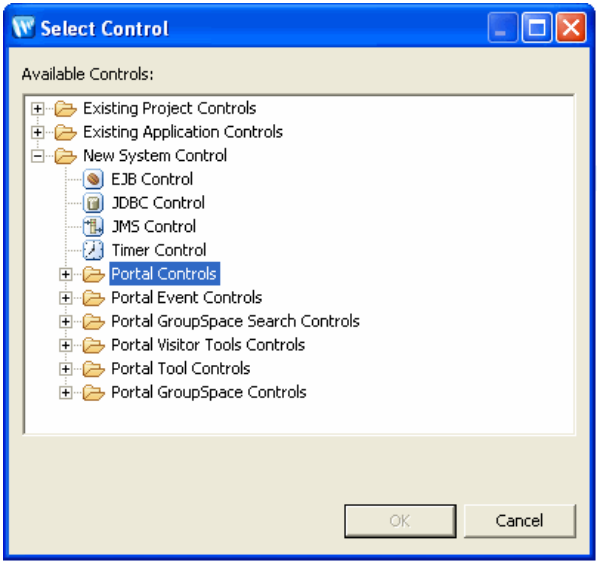
2. If you are not already using the Page Flow Perspective, Workshop for WebLogic asks if you want to switch to it. Do so.
3. Right-click in the source view for the Page Flow and select **Insert > Control**, as shown in Figure 5-36.

Figure 5-36 Insert > Control Menu Selection



The Select Control dialog box displays, as shown in Figure 5-37.

Figure 5-37 Select Control Dialog



4. Expand the desired folder to view the custom Java controls for WebLogic Portal that you can choose from.

After you add a custom WebLogic Portal control, al the methods in the control become available to your Page Flow.

For more information about the custom controls provided by WebLogic Portal, refer to the *Portal Development Guide*. For details about each control, refer to the *Controls Javadoc*.

Portlet State Persistence

You can control portlet state persistence using the `persistence-enabled` attribute in the `netuix-config.xml` file, which is located by default in the WEB-INF directory. Using this attribute causes the state to be saved in the WebLogic Portal database. The attribute is set to `false` by default.

The following code segment shows an example of the attribute syntax:

```
<control-state-location>
<session persistence-enabled="true"/>
</control-state-location>
```

WebLogic Portal places an entry for the control tree state in the PROPERTY_KEY table, with the following PROPERTY_SET_NAME value:

- BEA_PORTAL_FRAMEWORK_CONTROL_TREE_STATE

Adding a Portlet to a Portal

In the development phase of the portal life cycle, you add portlets to a portal using the Workshop for WebLogic workbench.

Note: A page must have a layout before you can add a portlet to it. The vertical or horizontal placement of portlets in a placeholder is determined by the selected layout for the page.

Follow these steps:

1. In the Package Explorer view, double-click the portal (`.portal` file) to which you want to add the portlet.

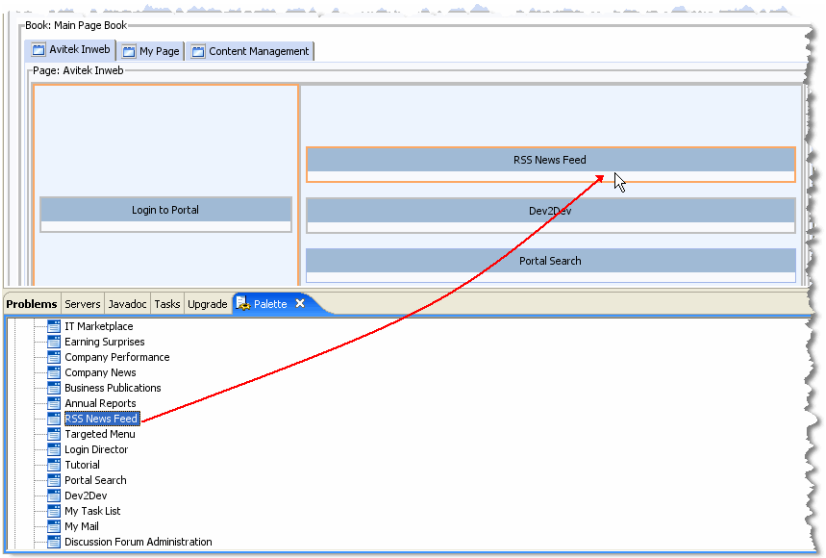
   The portal displays in the editor.

2. If your portal has multiple pages, click the desired page to select it.

3. From the Palette view, drag the portlet (the `.portlet` file) onto the portal page at the desired location.

   Figure 5-38 shows an example of this step.

Figure 5-38 Dragging a Portlet from the Palette onto a Portal Page in Editor View



With the portlet selected, you can use the Properties view to customize desired portlet properties.

For detailed information about portlet properties, refer to "Portlet Properties" on page 5-32.

When you add a portlet to a page in the workbench editor, a reference to that portlet is added to the `.portal` file. You can use the `.portal` file as a template for creating desktops in the WebLogic Portal Administration Console. When a portal administrator creates a desktop based on that template, the portlet is added to the portal resource library where it can be added to pages in streaming desktops. For an overview of file-based portals compared with streaming portals, refer to the *Portal Development Guide*.

In the Staging phase of the portal life cycle, you use the WebLogic Portal Administration Console to configure portlets on desktops. A single portlet definition can be associated with one or more portals (desktops) by creating instances of the portlet. Each of these portlet instances can have its own "personality" and behavior as specified by a variety of different configuration options.

For details in adding a portlet to a portal desktop in the WebLogic Portal Administration Console, refer to "Managing Portlets on Pages" on page 8-5.

Deleting Portlets

To remove a portlet from a portal without deleting the portlet from your portal web project, right-click the portlet in the Workshop for WebLogic workbench *editor* and click **Delete**.

To delete a portlet from your portal web project, right-click the portlet in the Package Explorer view and choose **Delete**.

To remove a portlet after you have assembled portlet instances into portal desktops using the Administration Console, refer to "Deleting a Portlet" on page 8-5.

Third-Party Portlets

WebLogic Portal partner companies create special-purpose portlets that you can easily incorporate into your portal; these companies include Autonomy, Documentum, and MobileAware.

The following sections provide more information about third-party portlets:

- Autonomy Portlets
- Documentum Portlets
- MobileAware Portlets

Autonomy Portlets

WebLogic Portal includes an embedded license of Autonomy-based search capabilities. You can use these capabilities to integrate enterprise-class search into your portal; common use cases include integration with content management systems, relational databases, and external web sites. You can expose these sources of information for searches using portlets that some with WebLogic Portal, and developers can use Autonomy APIs as they author new portlets and business logic for integrating search into your portal as well.

In WebLogic Portal 9.2, the BEA proprietary search APIs are deprecated; we recommend that you use Autonomy APIs to implement search capabilities.

For more information about Autonomy, see the Autonomy documentation.

Documentum Portlets

EMC Documentum has partnered with BEA to offer *EMC Documentum Content Services for BEA Weblogic Portal*. This product provides a packaged set of Documentum functionality

exposed through the BEA WebLogic Portal infrastructure, allowing users to access and interact with all types of enterprise content including web pages, documents, and rich media such as audio and video.

From a portlet development perspective, a key feature of this product is the inclusion of Documentum portlets—application components that expose standardized, enhanced content management user functions through the portal interface.

Documentum portlets expose four key applications:

- Content management portlets allow users to manage any type of content.
- Web Publisher portlets permit casual users to publish content to web sites and portals.
- eRoom portlets provide dashboard views into EMC Documentum eRooms and allow multiple project management.
- The Enterprise Content Integration (ECI) Services portlet enables continuous access to content in other repositories, databases, and Web sites.

See the Documentum web site for more information on Documentum portlets for WebLogic Portal

MobileAware Portlets

BEA WebLogic Mobility Server provides a standards-based, non-proprietary environment that extends BEA WebLogic deployments to offer multichannel mobile services in significantly reduced time frames. Enterprises can broaden the effectiveness of business-critical systems for employees and customers, and mobile carriers can rapidly deploy new, data-centric services, without the need for re-training and re-tooling.

For more information about BEA WebLogic Mobility Server and how to use it with WebLogic Portal, see the product documentation on the *e-docs web site*.

Advanced Portlet Development with Tag Libraries

During the Development phase, you can add other resources to a GroupSpace Community, a custom Community, or a portal web application. Those resources are contained in three tag libraries:

- The `ActiveMenus` JSP tag library
- The `DragDrop` JSP tag library

- The `DynamicContent` JSP tag library

See the *Communities Guide* for additional information.

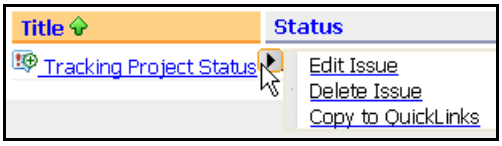
Adding ActiveMenus

You can add the `ActiveMenus` JSP tag library to a GroupSpace Community, a custom Community, or a portal web application.

The ActiveMenus JSP tag library lets you set up a popup menu that displays when the mouse hovers over specific text. An `activemenus-config.xml` file controls the contents of each menu. The `activemenus_taglib.jar` file contains the ActiveMenus tag library.

By default, a GroupSpace Community has ActiveMenus enabled, so you only need to configure the ActiveMenus tag (see "Configuring the ActiveMenus Tag" on page 5-90). See Figure 5-39 for an example of the ActiveMenus tag in a GroupSpace Community.

Figure 5-39 ActiveMenus in the GS Issue Portlet



You can tie a user's capability to the ActiveMenu that you see when you hover your mouse over an item (an Issue, for example) and hover over the arrow that appears. In this example, if your assigned capabilities include the ability to delete items, you will see the **Delete** choice, as shown in Figure 5-39.

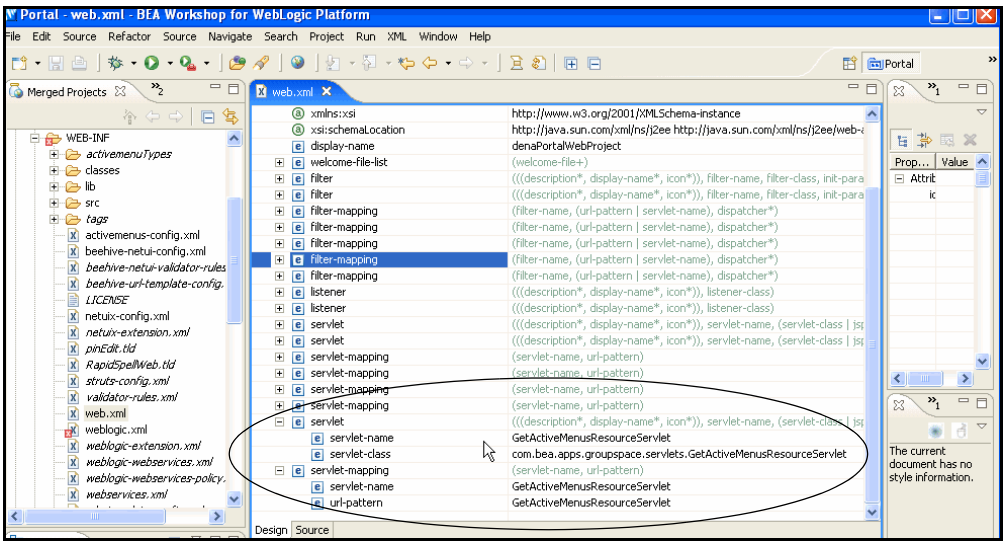
Tip: You do not need to perform the following steps if you have a GroupSpace Community; ActiveMenus are enabled by default for GroupSpace Communities.

Perform the following steps to enable ActiveMenus in a custom Community:

1. In Workshop for WebLogic, make the `activemenus_taglib.jar` file available to your portal web project. When you create your portal web project, you must enable the GroupSpace facets by selecting the **WebLogic Portal Collaboration** check boxes.
2. Add the `activemenus-config.xml` file to your `/WEB-INF` directory in your portal web project. Add the file by right-clicking the `activemenus-config.xml` file and choosing **Copy To Project**. Configure the file by follow the instructions in Configuring the ActiveMenus Tag to edit the `activemenus-config.xml` file.

3. Register the `GetActiveMenusResourceServlet` by adding the servlet and servlet-mapping to the `web.xml` file in the `/WEB-INF` directory in your portal web project. You can edit the file in Workshop for WebLogic by double-clicking the `web.xml` file. Right-click the **web-app** line in the file and choose **Add Child > message-destination - welcome-file-list > servlet**. Add `GetActiveMenusResourceServlet` to the `servlet-name` line. Add `com.bea.apps.groupspace.servlets.GetActiveMenusResourceServlet` to the `servlet-class` line. See Figure 5-40 to view the edited file in Workshop for WebLogic.

Figure 5-40 Editing the web.xml File in Workshop for WebLogic



The code sample in Listing 5-1 shows the new information you added.

Listing 5-1 Code Sample of GetActiveMenusResourceServlet

```
<!-- ActiveMenus Servlet Mappings -->
<servlet>
     <servlet-name>GetActiveMenusResourceServlet</servlet-name>
     <servlet-class>
          com.bea.apps.groupspace.servlets.GetActiveMenusResourceServlet
     </servlet-class>
</servlet>
<servlet-mapping>
     <servlet-name>GetActiveMenusResourceServlet</servlet-name>
```

```
    <url-pattern>GetActiveMenusResourceServlet</url-pattern>
</servlet-mapping>
```

4. Redeploy the application for the changes to take effect.

After you enable the ActiveMenus, you must configure the `ActiveMenus` tag.

Configuring the ActiveMenus Tag

To use the ActiveMenus tag, you must set up the `activemenus-config.xml file` (the `XSD` that defines this config file is located in the `activemenus_taglib.jar` file as `activemenus-config.xsd`). This `activemenus-config.xml file` file must exist in your web application's `/WEB-INF` directory. Multiple menus can be set up that consist of completely different items, styles, and icons.

Use the following sections to configure the `activemenus-config.xml file` file:

- Using The TypeInclude tag
- Using The Type Tag
- Using The TypeDefault Tag
- Using The menuItem Tag

Using The TypeInclude tag

Use the `typeInclude` tag to keep your configuration file clean. Rather than adding the `type` tag (see Using The Type Tag) you can add this tag and point its `href` attribute to an XML file (relative to the web application) that contains all of the `type` information. An example of the `typeInclude` tag is: `<typeInclude xhref="/WEB-INF/activemenuTypes/username.xml"/>`.

You can also use the `type` tag with the `typeInclude` tag in the configuration file. See the code sample in Listing 5-2.

Listing 5-2 You Can Use the typeInclude Tag with the Type Tag in the activemenus-config.xml File

```
<typeInclude xhref="/WEB-INF/activemenuTypes/username.xml"/>
<type>
    <menuItem>
```

```
          <param name="linkId"/>
          <action action="editLink">
               <i18nNamebundleName="com.bea.apps.groupspace.links.
                    LinksPopupMenu" key="edit.link"/>
          </action>
          <img xsrc="wlpAppsCollaborationCore/images/wlp-edit-16.gif"/>
     </menuItem>
</type>
```

When you point to another XML file, ensure that you namespace it correctly, as shown in Listing 5-3.

Listing 5-3 Pointing to Another XML File Called username.xml

```
<type name="username"
     xmlns="http://www.bea.com/servers/apps/groupspace/ui/
          activemenus-config/9.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.bea.com/servers/apps/groupspace/ui/
          activemenus-config/9.0">
...
</type>
```

Using The Type Tag

The `type` tag defines the individual menus to use within the web application. The `name` attribute must be unique for each menu, because the name is how the menu is referenced when you use the ActiveMenus tag. Following is an example of the `type` tag:

```
<type name="foo">
</type>
```

Note: The `TypeDefault` and `MenuItem` tags must be contained within the `type` tag.

Using The TypeDefault Tag

The `typeDefault` tag defines what displays in the browser where the `ActiveMenus` tag is used. You can control the text that displays, the style of the text, and the image that appears on the mouseover of that text (which denotes the menu itself).

The following items display within the browser where you used the `ActiveMenus` tag:

- The `displayText` Attribute – Defines the actual text that displays. If the `displayText` is not defined, whatever text is placed in the `display` attribute of the `ActiveMenus` tag appears. However, if you want to display other text, you can specify a class and a method within that class that returns a String to display. The following example shows how to display other text.

  `GetUserNameFromProfile.java`

  ```
  public class GetUserNameFromProfile
  {
       public static String getName(String userName)
       {
            return "XXX-" + username + "-XXX";
       }
  }
  ```

  If you use this code, the configuration defined above, and the following `ActiveMenus` tag: `<activemenus display="UserName" type="foo"/>`, the following displays in the browser: `XXX-UserName-XXX`.

  This example allows you to use the information entered in the body of the `ActiveMenus` tag to look up other information to display. For instance, a username can be used to look up a user's full name to display. The only rules surrounding this action is that the method used for the display text is public, static, takes in a String, and returns a String. No other information can be passed into that method.

- The `displayTextStyle` Attribute – Defines the CSS style or class that stylizes the display text. In order for the `class` attribute to work correctly, the class must be defined on the page (or the CSS file that defines the class must be imported).

- The `displayMenuImage` Attribute – Defines the image that appears when the display text is passed over with the mouse. If this tag is not defined, the default image is used. This image is in the `activemenus_taglib.jar` file and is called `menu_default.gif`.

- The `menuStyle` Attribute – Defines the CSS style or class that stylizes the menu itself, which can include the border or background color. For the `class` attribute to work correctly, the class must be defined on the page (or the CSS file that defines the class must be imported).

Note: The `TypeDefault` and `MenuItem` tags must be contained within the `type` tag.

Using The menuItem Tag

The `menuItem` tag defines the individual items within the popup menu. Listing 5-4 shows a code sample using the `menuItem` tag.

Listing 5-4 The menuItem Tag

```
<menuItem>
     <param name="userId"/>
     <xmlHttp url="GetFirstNameServlet"/>
     <row class="menuRow" style="backround-color:red"/>
     <text class="menuText" style="color:#000000"/>
     <rowRollover class="menuRowRollover" style="background-color:green"/>
     <textRollover class="menuTextRollover" style="color:#FFFFFF"/>
</menuItem>
<menuItem>
     <javascript>
     <name>Testing</name>
     <script>testing(this);</script>
     </javascript>
</menuItem>
<menuItem default="true" showMenuItem="false">
     <param name="q" value="foo"/>
     <link url="http://www.google.com">
          <name>Google</name>
     </link>
</menuItem>
<menuItem>
     <showMenuItem className="com.foo.CheckUserRights" methodName=
          "doesUserHaveRights">
          <rights name="can_view"/>
          <rights name="can_edit"/>
     </showMenuItem>
     <allParams/>
     <action action="addEditLink" disableAsync="true">
          <i18nName bundleName="com.foo.LinksPopupMenu" key="edit.link"/>
```

```
     </action>
</menuItem>
<menuItem>
     <allParams/>
     <dcAction action="showFeedData" dcContainerId="feedDataContainer">
          <i18nName bundleName="com.foo.LinksPopupMenu" key="show.
               feedData"/>
     </dcAction>
</menuItem>
```

The `menuItem` tag defines the individual items within the popup menu with the following four types:

- The `javascript` Element – This element can be any JavaScript that you want to run when the user clicks this menu item. To make this more useful, you can retrieve the values that you specify in the `param` tag (see the code sample below) through custom parameters that are added to the menu item. Following is a basic example of how to implement JavaScript.

  ```
  ...
       <activeMenus:activemenus display="Foo Link" type="link">
            <param name="linkId" value="${fooLink.id}"/>
            <param name="linkParent" value="${fooLink.parent}"/>
        </activeMenus:activemenus>
  ...
  ```

  The next step is to define the custom JavaScript in your configuration file. The JavaScript must pass in the code shown in Listing 5-5.

Listing 5-5 The activemenus-config.xml File

```
...
     <type name="link">
          <menuItem>
               <allParams/>
               <javascript>
                    <name>Testing</name>
                    <script>fooTest(this);</script>
               </javascript>
          </menuItem>
```

```
    </type>
...
```

The last step in implementing the JavaScript element is to access the values in your JavaScript function, as shown in the following code sample.

```
...
     <script>
     function fooTest(object)
     {
          var linkId = object.getAttribute("linkId");
          var linkParentName = object.getAttribute("linkParent");
     }
     </script>

 ...
```

- The `xmlHttp` Element – The `xmlHttp` references a servlet (which must follow all standard servlet configuration). Whatever the servlet outputs is shown in that row of the menu. If `""` or `null` is returned from the `xmlHttp` servlet, the menu item row does not appear in the menu. The information is retrieved through an `xmlHttp` request, which allows the information to be updated without refreshing the page. For example, you could show a user's online status that would update without having to make a full post. The two rules that surround writing your servlet for this is that all the processing must happen in the servlet's `doPost()` method. The second rule is that the defined parameters are passed in as request parameters. Following is an example of getting the query parameters:

  ```
  String userName = request.getHeader("linkId");
  ```

- The `link` Element – This static URL opens a new browser window pointed to the defined URL. This tag can take in either a `name` tag or an `i18nName` tag (defined below) that is displayed within the menu itself. Any defined parameters are added to the end of the link as regular request parameters.

- The `action` Element – This `action` name must be available to the page or portlet that contains the `ActiveMenus` tag. This element runs the action within the current browser, so you can use forwards to control your page flow. This tag can take in a `name` tag or an `i18nName` tag (defined below) that will appear within the menu itself. Any defined parameters passed in are available on the request as parameters. Following is an example of retrieving these values from a page flow:
  ```
  String linkId = getRequest().getParameter("linkId");
  ```

You can also use an attribute called `disableAsync` within AJAX-enabled portlets. If you want your menu item action to submit outside of the AJAX framework (so the page makes a full post), set this attribute to `true`. By default, the attribute is set to `false`.

- The `dcAction` Element – If you have a Dynamic Content container set up within your page, you can set up a menu item to call an action and have it update the Dynamic Content container. This works the same as an `action` menu item, and takes in the action name to execute. The only difference is you must specify the `dcContainerId` and it must correspond to a `dcContainerId` that is defined within a `<dc:executeContainerAction>` tag on the page.
- Other attributes and elements that you might use include the following:
  - The `showMenuItem` Element – Add this element if you need to conditionally show the menu item (for example, based on a set of rights for the current user). You define a class name and a method name that determines if the menu item should be shown. You can use multiple `showMenuItem` tags, each using different classes, methods, or rights. If you use more than one tag, all cases must be satisfied in order for the menu item to appear. For example, if the user passes nine of 10 cases, the menu item does not appear because all cases were not passed. Listing 5-6 shows how you can use the `showMenuItem` tag.

Listing 5-6 The CheckUserRights.java Class with the showMenuItem Tag

```
public class CheckUserRights
{
     public static boolean doesUserHaveRights(HttpServletRequest request,
          String[] rights)
     {
          for(int i=0;i<rights.length;i++)
          {
               if(!checkAccess(request, rights[i]))
               {
                    return false;
               }
          }
          return true;
          }
     }
```

– The `default` Attribute – When this attribute is used in a `menuItem` tag and set to `true`, the display text anchor's `href` will be the link or action. Use this attribute when you want a default action to occur when clicking the main link, and you also want to display the action for consistency purposes. The default value for this attribute is `false`.

– The `showMenuItem` Attribute – When this attribute is used in a `menuItem` tag and set to `false`, the menu item does not appear in the ActiveMenu. Use this attribute when you want a default action to occur when you click the main link, but you do not want to display the action. The default value for this attribute is `true`.

Note: Do not wrap an `ActiveMenus` tag in an `anchor` tag because you can get undesired results. Instead, use the `default` and `showMenuItem` attributes to control the ActiveMenu display text link

– The `allParams` Element – This element specifies that all of the parameters defined on the tag (see Using the ActiveMenus Tag) are set up on this menu item. If this element is not used (and the `param` element is not used), then parameters are not set up on the menu item.

– The `param` Element – This element sets the specified parameters on the menu item. The `param` element has a `name` attribute that must match the `name` attribute on a `param` element that is set within the `ActiveMenu` tag (see Using the ActiveMenus Tag). This also has a `value` attribute that can be used to hard code a value at configuration time. If this `value` attribute has been set, but a `value` was also specified at run-time (for example, using the `param` tag within the `ActiveMenu` tag), the run-time value takes precedence over the hard-coded value. Also, if just the hard-coded value is to be used, the `param` tag does not have to be specified when you use the `ActiveMenus` tag.

– The `name` Element – This element displays only the static name defined within the tag as the menu item.

– The `i18nName` Element – This element has both a `bundleName` attribute, which must map to an available `.properties` file, and a key attribute. The `bundleName` attribute uses the standard Java ResourceBundle convention. The `key` attribute defines the key to grab within the specified bundle. The text that relates to this key within this bundle is what appears in the menu item.

– The `img` Element – This element adds the specified image to the left column as an icon. You must specify the path to the image file in relation to your web application.

- The bgImg Element – This element replaces the background image used in the left column with the specified image. You must specify the path to the image file in relation to your web application.
- The row Element – This element defines the CSS style or class that stylizes the row of the menu item. For the class attribute to work correctly, the class must be defined on the page (or the CSS file that defines the class must be imported).
- The text Element – This element defines the CSS style or class that stylizes the text of the menu item. For the class attribute to work correctly, the class must be defined on the page (or the CSS file that defines the class must be imported).
- The rowRollover Element – This element defines the CSS style or class that stylizes the row of the menu item when it is rolled over. For the class attribute to work correctly, you must define the class on the page (or the CSS file that defines the class must be imported).
- The textRollover Element – This element defines the CSS style or class that stylizes the text of the menu item when it is rolled over. For the class attribute to work correctly, you must define the class on the page (or the CSS file that defines the class must be imported).

Note: The TypeDefault and MenuItem tags must be contained within the type tag.

Using the ActiveMenus Tag

The taglib.tld file is located in the activemenus_taglib.jar file.

You can use the following attributes and elements with the ActiveMenus tag:

- The display Attribute – This attribute defines what appears in place of the tag itself. If you use the displayText attribute, this is the value that is passed to the method defined in the displayText tag.
- The type Attribute – This required attribute defines what is in the menu and must match a type defined in the activemenus-config.xml file.
- The href Attribute – This optional attribute can override the default anchor href for the display text of the tag.
- The newWindow Attribute – This optional href attribute specifies the link to open in a new browser window. This is a Boolean attribute, and you set it to true or false.
- The class Attribute – This optional attribute defines a CSS class for the display text.

- The `style` Attribute – This optional attribute defines a CSS style to place on the display text.
- The `rightClick` Attribute – This Boolean attribute turns the menu into a right-click menu, rather than a rollover menu. The default is `false`. If this attribute is set to `true`, you right-click the display text to bring up the menu. The menu appears under the mouse.
- The `escapeXml` Attribute – This attribute is the same as `escapeXml` within the JSTL tags. If you set it to `true`, characters are converted to their corresponding character entity codes.
- The `param` Element – This element sets up parameters that can be passed in and used for the different menu items. The following two attributes are both required:
  - The `name` Attribute – This is the parameter name and must match the `name` attribute (if used) when defining a menu item in the `activemenus-config.xml` file. The `name` attribute also references the parameter within your menu item code. You can use a runtime expression.
  - The `value` Attribute – This is the parameter value, and you can use a runtime expression.

Notes: If a class is specified on the tag, the default class specified in the `activemenus-config.xml` file is overridden and the default style is not placed on the `activename`. If a style is specified on the tag, the default class is placed on the `activename`. If a `class=""` is specified on the tag, the default class is not placed on the `activename`.

Enabling Drag and Drop

You can use the DragDrop JSP tag library to enable drag and drop functionality in a GroupSpace Community, a custom Community, or a portal web application. You must identify draggable objects that are displayed on a JSP, and identify drop zones that are configured to react to a dropped draggable object. The drop zones react by triggering Page Flow actions, calling JavaScript functions, or posting data to a servlet.

Perform the following actions before you use the DragDrop tag library:

- Include the `dragdrop_taglib.jar` file in the web application's CLASSPATH
- Place the code shown in Figure 5-1 into your `web.xml` file

Figure 5-1 Code Entry in the web.xml File

```
<servlet>
     <servlet-name>DragDropResourceServlet</servlet-name>
     <servlet-class>com.bea.apps.communities.servlets.
          GetDragDropResourceServlet
</servlet-class>
</servlet>
<servlet-mapping>
     <servlet-name>DragDropResourceServlet</servlet-name>
     <url-pattern>DragDropResourceServlet</url-pattern>
</servlet-mapping>
```

Using the DragDrop Tags

Three tags are defined in the DragDrop tag library. Following are descriptions of how each tag is used, along with sample JSP code:

- The `dragDropScript` Tag – This tag includes the necessary DragDrop JavaScript libraries in the page. The logic embedded into the tag ensures that these libraries are included only once per request.
- The `draggableResource` Tag – This tag identifies a draggable resource on the page.
- The `resourceDropZone` Tag – This tag identifies an area on the page that reacts when a draggable resource is dropped.

Using the dragDropScript Tag

You must include the `dragDropScript` tag before you use any other DragDrop tags on the page. This tag ensures that the appropriate JavaScript libraries are included. The `dragDropScript` tag does not take any attributes.

The following example shows how to use the `dragDropScript` tag: `<dragdrop:dragDropScript/>`.

Using the draggableResource Tag

The `draggableResource` tag specifies a draggable resource on the page. The tag takes the following attributes:

- The `resourceId` Attribute – The unique identifier of the resource that is being dragged. This identifier should be an ID that can be used by the underlying business logic to uniquely identify the resource.
- The `resourceName` Attribute – The representative name of the resource being dragged.

The `draggableResource` tag performs a search for a child `img` tag that has a `dragdrop:image` attribute. This image becomes the image that is displayed while performing the drag operation. The image must have an absolute height and width attribute.

The `resourceId` value is accessible through the JavaScript function `getSourceId()`, when the value is dropped onto a `resourceDropZone`. The `resourceId` value is also available as a parameter in the request named `sourceId`, when it is dropped onto a `resourceDropZone` that triggers a `POST` action. See Listing 5-7.

Listing 5-7 The sourceId Request Dropped onto a resourceDropZone

```
<dragdrop:draggableResource imageId="0" resourceId="${id}"resourceName=
    "${name}">
    <img src="/image.gif" width="16px" height="16px"dragdrop:image="true"/>
         ${name}
</dragdrop:draggableResource>
```

Using the resourceDropZone Tag

The `resourceDropZone` tag identifies an area where draggable resources can be dropped.

The tag takes the following attributes:

- The `targetId` Attribute – The unique identifier of the drop zone object. This identifier can be an ID that can be used by the underlying business logic to uniquely identify which object received the drop action.
- The `jsFunctionCall` Attribute – A JavaScript function that executes when a `draggableResource` is dropped on this `resourceDropZone`.
- The `pageFlowAction` Attribute – A valid Page Flow action that is initiated when a `draggableResource` is dropped on this `resourceDropZone`.
- The `formAction` Attribute – A valid JSP or servlet that receives a `POST` action when a `draggableResource` is dropped on this `resourceDropZone`.

Only one of the following attributes is required: `jsFunctionCall`, `pageFlowAction`, or `formAction`. The `jsFunctionCall` takes precedence, then `pageFlowAction`, and finally `formAction`.

The `targetId` value is accessible through the JavaScript function `getTargetId()` when a draggable resource is dropped. It is also available as a parameter in the `targetId` request when a draggable resource is dropped that triggers a POST action. The following code shows how this works:

```
<dragdrop:resourceDropZone targetId="${id}" pageFlowAction="moveIssue">
     <img src="/folder.gif"/>Issues Folder
</dragdrop:resourceDropZone>
```

Listing 5-8 demonstrates how the `moveIssue` action can be coded in a file called `IssuesPageFlowController.java`.

Listing 5-8 Coding the moveIssue Action

```
@Jpf.Action(forwards={ @Jpf.Forward(name = "success", path =
     "displayIssuesTree.do")})
     protected Forward moveIssue() {
          Forward forward = new Forward("success");
          String sourceId = getRequest().getParameter("sourceId");
          String targetId = getRequest().getParameter("targetId");
          move(sourceId, targetId);
          return forward;
     }
```

Enabling Dynamic Content

You can use the `DynamicContent` tag library to quickly update parts of a JSP page in a GroupSpace Community, a custom Community, or a portal web application.

The `DynamicContent` tags let you use an AJAX request to update part of a JSP page within a Page Flow-based portlet. The tags allow parts of the page to be updated without performing a full portal request. These AJAX requests are smaller and faster than full portal requests, and therefore provide a more responsive user experience when interacting with a portal application.

These tags are easy to incorporate into standard Page Flow-based portlet development and can help create advanced user interface features that improve a user's portal experience.

Note: The `DynamicContent` tags are not related to Asynchronous Portlet Content Rendering. Asynchronous portlets allow for the entire portlet content to be rendered independently of the portal. The `DynamicContent` tags are designed to affect small parts of a JSP page within a portlet.

Understanding the DynamicContent Tags

This section describes the main tags in the `DynamicContent` tag library.

The Container Tag

The `Container` tag designates a place on the JSP page that contains the HTML output from the execution of a Page Flow action. The only required attribute for this tag is a container id. This id is referenced by other `DynamicContent` tags to identify the container. The following code shows how this tag is used: `<dc:container dcContainerId="outputContainer"/>`.

The Container Action Script Tag

This tag is a child of the `Container` tag and identifies a Page Flow action that can be executed and whose HTML output is placed inside the parent container. The `containerActionScript` tag takes the following attributes:

- The `action` attribute – The Page Flow action name.
- The `initial` attribute – Designates an action in the container as the initial action. This is the action that initially populates the container.
- The `async` attribute – Specifies if the action is performed synchronously or asynchronously. The default is synchronous.
- The `onErrorCallback` Attribute – A user-defined JavaScript function that is called if a client-side error occurs during the AJAX request creation and processing.

Only the `action` attribute is required. The following code sample shows how this tag is used in the parent `Container` tag:

```
<dc:container dcContainerId="outputContainer">
     <dc:containerActionScript action="resetDynamicContentContainer"
          initial="true"/>
     <dc:containerActionScript action="showServerTime"/>
<dc:container/>
```

The Execute Container Action Tag

The Execute Container Action tag is used to create a call to a specific action inside a container. This tag takes the following attributes:

- The `dcContainerId` attribute – The id of the container in which the action is defined.
- The `action` attribute – The Page Flow action name.
- The `async` attribute – This specifies if the action is performed synchronously or asynchronously. The default is synchronous.
- The `var` attribute – A request attribute variable that holds a reference to the action JavaScript call.

The `dcContainerId` and `action` attributes are required. Following is a sample of how this tag is used:

```
<dc:executeContainerAction action="showServerTime" dcContainerId=
     "outputContainer"
          var="showServerTimeVar"/>
```

In the previous example, the call to the specified action is stored in the variable `showServerTimeVar`. This variable can then be referenced, as shown in the following HTML code:

```
<form>
     <input type="button" onclick="${showServerTimeVar}" value="Show Server
          Time"/>
</form>
```

When the user clicks a button, an AJAX request is created that executes the `showServerTime` action and places the HTML output generated by that action into the container with the id of `outputContainer`.

The Parameter Tags

The `DynamicContent` tags also include tags for parameters that are passed into the action through the request. You can define parameters within the `executeContainerAction` tag or the `containerActionScript` tag. These parameters are then accessible in the Page Flow action by calling the `request.getParameter()` method.

Using the DynamicContent Tags

Some critical limitations are associated with the `DynamicContent` tags. The AJAX requests used to trigger the Page Flow actions are not processed through the main portal servlet. These requests go through a special servlet that performs some processing to ensure that the proper Page Flow instance is used. Many key elements that are normally available in the request are not accessible from these AJAX requests. For example, in Community-based portal applications, the `CommunityContext` object is not accessible from the AJAX request. The lack of access to some of these framework elements could have an impact on things like entitlements and security.

Because of these limitations, the `DynamicContent` tags are best suited for specific uses that involve small amounts of processing, with few dependencies on larger framework services. The following use cases could benefit from the `DynamicContent` tags:

- Update a small location on a JSP page to display frequently updated data obtained through periodic client-side polling. For example, you could notify users of unread mail or display the number of users logged onto a system.
- Use the tags as a pagination mechanism for tabled data presented across multiple pages.
- Send multiple requests to the server to obtain successive images to navigate through a series of images in a photo gallery. The `DynamicContent` tags provide a tool to avoid an expensive portal request to view each photo.
- Obtain remote data, such as stock quotes or weather information from remote sites. The obtained data can be displayed in a designated area on the page without updating other parts of the page.

See dev2dev for sample code and utilities contained in the `sample.zip` file.

CHAPTER 6

Optimizing Portlet Performance

The process of optimizing your portlets for the best possible performance spans all phases of development. You should continually monitor performance and make appropriate adjustments.

This chapter describes performance optimizations that you can incorporate as you develop portlets.

This chapter contains the following sections:

Performance-Related Portlet Properties

Customizing performance-related portlet properties can help you improve performance. For example, you can set process-expensive portlets to pre-render or render in a multi-threaded (forkable) environment. If a portlet has been designed as forkable (multi-threaded) you have the option of adjusting that setting when building your portal.

The following portlet properties are performance related:

- Render Cacheable/Cache Expires

- Forkable/Fork Render/Fork Render Timeout
- Fork Pre-Render/Fork Pre-Render Timeout
- AsyncContent

"Portlet Properties" on page 5-32 includes descriptions of these properties. If you design your portlets to allow portal administrators to adjust cache settings and rendering options, you can modify those properties in the Administration Console.

Portlet Caching

You can cache the portlet within a session instead of retrieving it each time it recurs during a session (on different pages, for example). Portlets that call web services perform frequent, expensive processing; caching web service portlets greatly enhances performance. Portlet caching is well-suited to caching personalized content; however, it is not well suited to caching static content that is presented identically to all users and that rarely expires.

The ideal use case of the portlet cache is for content that is personalized for each user and expires often. In other situations, it might be more beneficial to use other caching alternatives such as using the `wl:cache` tag or the portal cache.

For a detailed examination of the *Render Cacheable* property and a discussion of when you should or should not use it, refer to the dev2dev article *Portlet Caching* by Gerald Nunn, available at *http://dev2dev.bea.com/pub/a/2005/01/portlet_caching.html*.

Remote Portlets

Remote portlets are made possible by Web Services for Remote Portlets (WSRP), a web services standard that allows you to "plug-and-play" visual, user-facing web services with portals or other intermediary web applications. WSRP allows you to consume applications from WSRP-compliant Producers, even those far removed from your enterprise, and surface them in your portal.

While there might be a performance boost related to the use of remote portlets, it is unlikely that you would implement them for this reason. The major performance benefit of remote portlets is that any portal framework controls within the application (portlet) that you are retrieving are rendered by the producer and not by your portal. The expense of calling the control life cycle methods is borne by resources not associated with your portal.

Implementations using remote portlets also have limitations; for example:

- Fetching data from the producer can be expensive. You need to decide if that expense is within reason given the resources locally available.
- Some features, such as customizations, are unavailable to the remote portlet.

If the expense of portal rendering sufficiently offsets the expense of transport and the other limitations described above are inconsequential to your application, using remote portlets can provide some performance boost to your portal.

For more information on implementing remote portlets with WSRP, refer to the *Federated Portals Guide*.

Portlet Forking

Portlet forking allows portlets to be processed on multiple threads. Depending on the available server resources, this means that the portal page will refresh more quickly than if all portlets were processed sequentially. Forking is supported for JSP, Page Flow, Java, and WSRP portlets (consumer portlets only).

Note: Although using this feature might reduce the response time to the user in most situations, on a heavily loaded system it can actually decrease overall throughput as more threads are being used on the server/JVM for each request—adding to contention for shared resources.

This section includes these topics:

Configuring Portlets for Forking

Forking is easy to enable – you just set properties using the portlet Properties editor in WorkSpace Studio, as shown in Figure 6-1. The available forking properties are described in this section. For detailed information on the Portlet Properties editor, see "Portlet Properties" on page 5-36.

Figure 6-1 Forking Properties

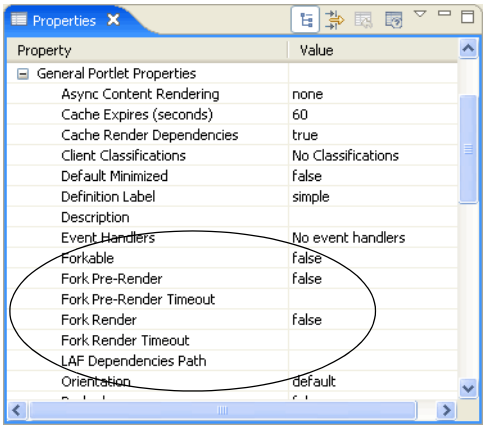


Table 6-1 Portlet Forking Properties

Property	Value
Forkable	This property must be set to true if you want the portlet to be forked. This property identifies the portlet as safe to run forked. If this attribute is false (the default), the portlet will not be forked regardless of the settings of the other two forking properties. See "Best Practices for Developing Forked Portlets" on page 6-10 for tips on developing forked portlets. When set to true, a portal administrator can use the **Run the Portlet in a Separate Thread** property. If set to false, that property is not available to administrators. See the *Portal Development Guide* for information on using the Administration Console to edit portlet properties.
Fork Pre-Render	Enables forking (multi-threading) in the pre-render life cycle phase. For an overview of the portal life cycle, see "Architectural Details of Forked Portlets" on page 6-6. See also "How the Control Tree Affects Performance" in the *Portal Development Guide* for more information about the control tree life cycle. Setting Fork Pre-Render to `true` indicates that the portlet's pre-render phase should be forked. See "Dispatching Pre-Render Forked Portlets to Threads" on page 6-9 for more information on the pre-render phase.

Property	Value
Fork Pre-Render Timeout (seconds)	If Fork Pre-Render is set to `true`, you can set an integer timeout value, in seconds, to indicate that the portal framework should wait only as long as the timeout value for each fork pre-render phase. The default value is -1 (no timeout). If the time to execute the forked pre-render phase exceeds the timeout value, the portlet itself times out (that is, the remaining life cycle phases for this portlet are cancelled), the portlet is removed from the page where it was to be displayed, and an error level message is logged that looks something like the following example. `<May 26, 2005 2:04:05 PM MDT> <Error> <netuix> <BEA-423350> <Forked render timed out for portlet with id [t_portlet_1_1]. Portlet will not be included in response.>`
Fork Render	Setting to `true` tells the framework that it should attempt to multi-thread render the portlet. This property can be set to true only if the `Forkable` property is set to `true`. See "Dispatching Render Forked Portlets to Threads" on page 6-9 for more information on the render phase.
Fork Render Timeout (seconds)	If Fork Render is set to `true`, you can set an integer timeout value, in seconds, to indicate that the portal framework should wait only as long as the timeout value for each fork render portlet. The default value is -1 (no timeout). When a portlet rendering times out, an error is logged, but no markup is inserted into the response for the timed-out portlet. Selecting a value of `0` or `-1` removes the timeout attribute from the portlet; use this value if you want to revert to the framework default setting for this attribute.

The forking properties, if set, appear as XML elements a .portlet file. Listing 6-1 shows a sample of a portlet configured for both pre-render and render forking:

Listing 6-1 Forking Properties Set in a .portlet File

```
<netuix:portlet title="Forked Portlet"
    definitionLabel="forkedPortlet1"
    forkable="true"
    forkPreRender="true"
    forkRender="true">
    <netuix:content>
        <netuix:jspContent contentUri="/portlets/forked.jsp"
```

```
        backingFile="backing.PreRenderBacking"/>
    </netuix:content>
</netuix:portlet>
```

Architectural Details of Forked Portlets

Generally, forking is easy to understand and to enable. However, with a deeper understanding of how forking works, you can avoid potential problems and unwanted side effects. This section discusses the architectural design of forked portlets. For specific implementation tips, see "Best Practices for Developing Forked Portlets" on page 6-10.

This section includes these topics:

- Understanding Request Latency and the Portal Life Cycle
- Queuing and Dispatching Forked Portlets for Processing
- Threading Details and Coordination
- Forking Versus Asynchronous Rendering

Understanding Request Latency and the Portal Life Cycle

For most requests to the portal, the total time to process the request, or *request latency*, is roughly related to the time needed to run through the portal life cycle phases successively for all the portlets. Each life cycle phase is performed by walking through a tree of objects, called the *control tree*, that make up the portal. Each phase is essentially a depth-first walk over the tree, where the root of the tree is the desktop, and the leaves of the tree are the books, pages, portlets, and other so-called controls. Figure 6-2 illustrates the general structure of a portal control tree.

Figure 6-2 Simple Portal Schematic Example

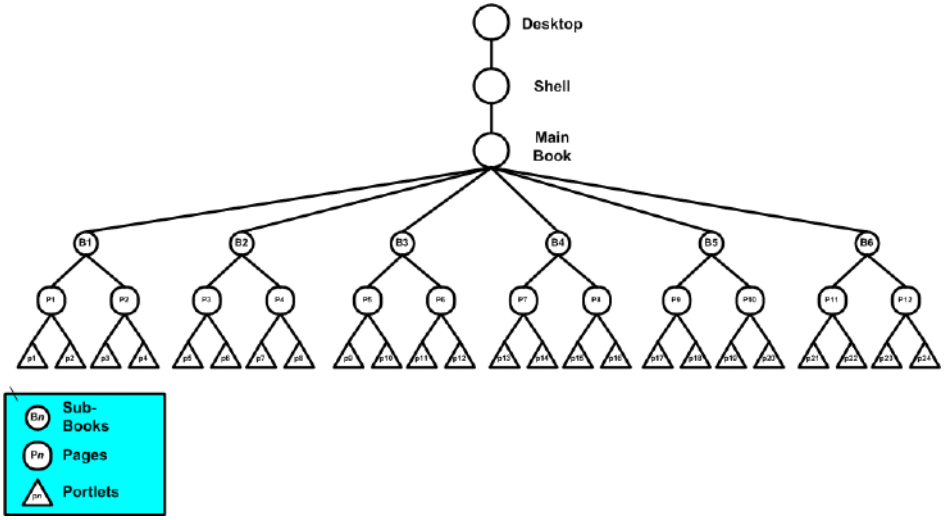
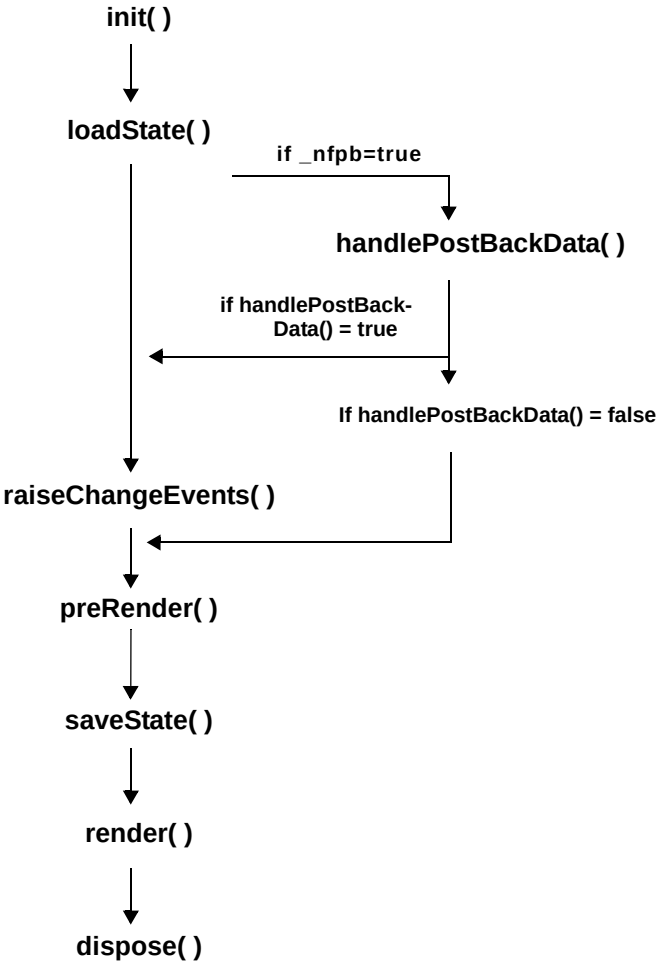


Figure 6-3 illustrates the successive phases of the portal rendering life cycle. During the first traversal of the control tree, the init() method is called on each control. On the second traversal, loadState() is called, and so on, until every control is processed.

Typically, portlet processing time is dominated by the execution of business logic, especially if the portlets must access remote resources such as databases or web services, or if they are computationally intensive. Forking allows you to parallelize some of these longer running portlet operations to decrease the overall request latency. If forking is enabled, these operations are collected in a queue and dispatched to multiple threads for processing. Depending on your server's resource availability, forking can theoretically reduce request latency to the maximum latency of any of the forked portlets.

Figure 6-3 Flow of Portal Life Cycle Methods



Queuing and Dispatching Forked Portlets for Processing

During the *pre-render phase* of the portal life cycle, all portal controls are iterated and pre-rendering operations are executed. Any portlets that are marked for either pre-render or render forking are identified during this pass and, if they are marked for forking, they are placed in separate queues: a pre-render queue and a render queue. (See "Configuring Portlets for Forking" on page 6-3 for details on how to mark portlets for pre-render and render forking.)

At the appropriate times, these queues are dispatched to threads and processed, as explained in the following sections. See also "Threading Details and Coordination" on page 6-9.

Dispatching Pre-Render Forked Portlets to Threads

In the pre-render phase of the portal life cycle, portlets typically perform business logic, typically by handling postback data or by calling a backing file method, such as the AbstractJSPBacking.preRender() method.

During normal pre-render processing of the portal, any portlet that is marked for pre-render forking is placed into a queue and the pre-render processing is skipped. After the entire pre-render phase has been performed, the queue is inspected. If it is not empty, the queue is dispatched and the portlets in the queue are assigned to a worker thread. After the queue is fully dispatched, the main portal thread waits until either all the worker threads are completed or timed out.

Dispatching Render Forked Portlets to Threads

In some cases, business logic is performed during the render phase of the portal life cycle, typically when JSP scriptlets are used.

Before running through the render life cycle, the render queue is examined. If it is not empty, the queue is dispatched and any portlets in the queue are assigned to worker threads. As with pre-render forking, the main portal thread waits until all of the render threads are either completed or timed out. The resulting buffered response from each thread is saved for each completed forked portlet. At this point, the actual render life cycle phase is run. When a portlet is encountered that was marked for forking, the render processing is skipped and the saved buffered response data for the portlet is written to into the response.

Some types of portlets, notably Struts or Page Flow portlets, provide a mapping between the underlying application technology and the portal life cycle model. Usually in these cases, actions are provided to handle business logic during the handle postback or pre-render phases of the life cycle.

Threading Details and Coordination

The worker threads used by the forking feature are implemented as WLS WorkManager classes. WebLogic Portal does not directly allocate any threads; rather, a WorkManager is identified by its JNDI name. If found, the WorkManager is used to dispatch the worker threads (Work instances). The default WorkManager for dispatching forked portlets is called wm/forkedRenderQueueWorkManager, with a default called wm/Default. If you need to customize the WorkManager for any reason, you can specify an alternate instance through the

weblogic.xml or weblogic-config.xml file by associating the alternate instance with the JNDI name wm/forkedRenderQueueWorkManager. See also "Consider Thread Safety" on page 6-11.

The framework uses a ForkedLifecycleContext object to coordinate between the mainline life cycle thread and the forked Worker instances. During initialization of a Worker, the ForkedLifecycleContext is created and registered with the forking dispatch queue. When the Work instance has completed, the ForkedLifecycleContext is set to completed and the waiting mainline thread is notified. Alternately, if the waiting mainline thread determines that the forked Work instance is taking too long and should be timed out, the ForkedLifecycleContext is marked as timed out and the Work instance is removed from the dispatch queue. Note that in this case, the Work item is not aborted, and will keep running until the portlet code being run for either the pre-render or render phase is completed. You can obtain the current ForkedPreRenderContext or ForkedRenderContext using a utility method on those classes from the request. You can then check if a timeout has been set to detect cases where the Worker thread was timed out by the portal framework and should be aborted.

Forking Versus Asynchronous Rendering

Regardless of whether or not you use render forking, the portal does not render until all portlets complete rendering. If you want portlets to render individually, you can use asynchronous portlet rendering.

Asynchronous portlet content rendering refers to page processing that occurs on the client browser; multiple threads are spawned, using AJAX or IFRAME technology. Asynchronous portlet rendering allows the contents of a portlet to render independently from the surrounding portal page. This can provide a significant performance boost; for example, when a portal visitor works within a portlet, only that individual portlet needs to be redrawn.

WARNING: Using forked rendering with asynchronous portlet content rendering is unnecessary, is not recommended, and could result in unexpected behavior.

For details on asynchronous rendering, see "Asynchronous Portlet Content Rendering" on page 6-13. For a comparison of portlet forking and asynchronous rendering, see "Comparison of Asynchronous and Conventional or Forked Rendering" on page 6-17.

Best Practices for Developing Forked Portlets

This section discusses three primary issues you need to consider when developing forked portlets: thread safety, runtime environment, and interportlet communication issues.

Consider Thread Safety

Although the portal framework handles thread safety issues that affect the framework itself, any code you write that is intended to be used in forked portlets should be written in a threadsafe manner.

- Only mark thread-safe portlets as forkable. This helps to ensure that administrators do not incorrectly enable forking for portlets that were not written with thread safety in mind.
- Cautiously evaluate interactions between your code and portal framework constructs. For example, do not unwrap the request and response objects. They are used specifically to isolate the request and response. For certain types of portlets, particularly Page Flow and Struts portlets, an additional wrapper is put in place, so one level of unwrap may work, but unwrapping to the root request or response will cause threading issues.
- Avoid using portal-managed objects, such as the request and response, for your own code synchronization. These objects are used by the portal framework for synchronization. If you use them for that purpose, out of order lock acquisition and deadlocks can occur.

Consider the Runtime Environment for Forked Portlets

When designing forked portlets, try to maximize their independence from other constructs in the portal (such as BackingContext) and from other portlets. Such dependencies create problems for forked portlets because forked portlets are inherently isolated from the runtime environment.

Isolation of Forked Portlets from the Runtime Environment

The primary difference between the runtime environment for forked portlets and non-forked portlets is in their level of isolation. This difference occurs because of the way that forked portlets are collected and dispatched outside of the life cycle execution for the main portal control tree.

Each life cycle iteration of the control tree results in a life cycle method being called for that control. In this way, each control has the opportunity to perform life cycle specific business logic. Additionally, each life cycle method invocation involves both a begin and end operation, which enables setup and teardown for controls that require such functionality.

Enabling preRender or render forking moves the execution of a portlet's life cycle processing from occurring within the main portal control tree walk to outside of it. The main side effects of this are:

- The forked portlet is essentially isolated from any stateful setup that its placement in the control tree provided.

- Forked portlets are executed out of order, both in terms of other nodes in the control tree and even amongst other sibling portlets. For the preRender phase, controls deeper in depth-first order will be executed ahead of forkPrerender portlets. For the render phase, all forkRender portlets will be executed before any other control in the tree processes its render phase.

As a developer of forked portlets, be aware that code meant to be executed in a forked portlet should be as stand-alone as possible. Avoid relying on interaction with other portlets, other controls higher in the control tree, or state provided by other controls in the control tree.

Do not rely on any processing done during the same life cycle in other portlets, because forking a portlet both takes it out of order with respect to control tree execution and applies an arbitrary ordering among forked portlets in the dispatch queue.

BackingContext and Pre-Render Forked Portlets

For preRender forked portlets, one of the main areas of concern for forked portlets is the BackingContext framework. This framework is managed in part by a stack-based implementation involving the request, which depends on Backable controls in the control tree to push and pop their BackingContext instances onto and off of the request. All of these activities happen during the pre-render life cycle phase. When writing a portlet that expects a particular BackingContext stack environment, problems can occur with Fork Pre-Render mode. Any access to BackingContexts through the request will result in that BackingContext not being available while forked.

To work around this BackingContext issue, you can use non-contextual methods to obtain BackingContexts for other presentation controls in the control tree, but these generally involve explicit walking of the context tree, and some contexts may be unavailable because the context in question has already been cleaned up by the control that manages it in preRender.

Use Caution with Interportlet Communication and Forked Portlets

Interportlet communication (IPC) is another area of concern for forked portlets. Again, the more you can isolate a portlet's logic, the more successfully it will run in a forked environment.

IPC is performed in several different life cycles. When an IPC scenario is enabled that results in an IPC call initiated during preRender, and a portlet is also enabled for forking, that IPC will not be performed, since the actual dispatch of the IPC event queue happens immediately following the main execution of preRender() over the control tree. This is of primary concern to portlets that raise IPC events in a backing file preRender() method, from a Page Flow, a Struts begin action, or from a JSF beginning view root.

Asynchronous Portlet Content Rendering

Asynchronous portlet rendering allows you to render the content of a portlet independently from the surrounding portal page. This can provide a huge performance boost; for example, when a portal visitor works within a portlet, only that individual portlet needs to be redrawn.

You can use either AJAX technology or IFRAME technology to implement asynchronous rendering. When using asynchronous portlet rendering, a portlet renders in two phases. The normal portal page request process occurs first; during this process, the portlet's non-content areas, such as the title bar, are rendered. Rather than rendering the actual portlet content, a placeholder for the content is rendered. A subsequent request process displays the portlet content.

This section contains the following topics:

Implementing Asynchronous Portlet Content Rendering

A new portlet property `asyncContent` in the Properties view allows you to specify whether to use asynchronous rendering, and to select which technology to use. An editable dropdown menu provides the selections `none`, `ajax`, and `iframe`. If you want to create a customized implementation of asynchronous rendering, you can do so by editing the `.portlet` file to set this up; more information about this task will be available in a dev2dev article in the future.

Portlet files that do not contain the `asyncContent` attribute appear with the initial value `none` displayed in the Properties view. Any changes to the setting are saved to the `.portlet` file.

Note: Although Browser portlets use an internal implementation that appears similar to that of an asynchronous portlet and both portlet types use IFRAME HTML elements, the actual implementations are quite different. Browser portlets are merely displaying static embedded documents, but asynchronous IFRAME portlets are managed by the framework.

Keep the following global considerations in mind for any asynchronous rendering implementation:

- As a best practice, do not depend on the built-in navigation features (Back and Forward buttons) of a browser. Build navigation into your portlets so that navigation is handled at that level of interaction.

  If navigation is handled by the browser, the behavior of a portlet will vary according to the type of asynchronous rendering technology used, and this inconsistency can be confusing to the end user. For example, with IFRAME technology each portlet interaction is tracked, but this interaction does not update the "view" from the server's perspective; if the user clicks the Back button, the server takes the user to a state preceding the interaction with the portlet.

- The initial (completion of) page load for an asynchronously rendered portlet page will be longer because, for example, loading a page containing five asynchronous portlets entails six requests to the server. However, because the portal page begins to load quickly, the user might perceive a faster load time even if the completion takes more time overall.

- You should pre-define portlet sizes using Look & Feel settings; otherwise, as the page loads, the portlets might resize several times as they adjust to their arrangement on the page.

- Portlet backing files are run twice: once for the outer (portal) request and another for the inner (content) request. You can use the set of framework APIs in the PortletBackingContext class to distinguish between these two requests; for more information, refer to the Javadoc information for these APIs:

  com.bea.netuix.servlets.controls.portlet.PortletPresentationContext.isAsyncContent()
  com.bea.netuix.servlets.controls.portlet.PortletPresentationContext.isContentOnly()
  com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext.isAsyncContent()
  com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext.isContentOnly()

- Asynchronous portlet rendering can be used with control tree optimization. Most of the best practices for control tree optimization also apply to the design of asynchronous rendering. For more information on control tree optimization, refer to the *Portal Development Guide*.

- Interportlet communication is not supported when asynchronous content rendering is enabled; however, you can temporarily disable asynchronous rendering in specific situations if needed. For details, refer to "Asynchronous Content Rendering and IPC" on page 6-19.

- HTTP redirects are not supported when asynchronous content rendering is enabled; however, you can temporarily disable asynchronous rendering using the same mechanisms as those described in "Asynchronous Content Rendering and IPC" on page 6-19.
- Using forked pre-rendering or forked rendering in an asynchronous portlet is unnecessary and in any case is not recommended, and although this is not an error condition, it could result in unexpected behavior.
- Using PostbackURLs (not derived types) within an asynchronous portlet (or a floated portlet) causes the portlet to lose various aspects of its state, including the results of render caching. Additionally, multiple instances of such portlets will begin to share state. To avoid this issue, you can use one of these workarounds:
  - Use alternative mechanisms for generating URLs more appropriate to the portlet type, such as `<render:jspContentUrl>` or `<netui:anchor>`.
  - Add `GenericURL.WINDOW_LABEL_PARAM` directly to the PostbackURL with the value returned from `PortletPresentationContext.getLabel()` or `PortletBackingContext.getLabel()`.
- WebLogic Portal allows portlets to change the current window state or mode of a portlet either programmatically, or using parameters added to URLs. When you enable asynchronous rendering for a portlet, these mechanisms will not provide a consistent view to the end user; for example, the title bar rendered above the portlet will not immediately reflect the change in the mode or state.
- In addition to the issues described in "Asynchronous Content Rendering and IPC" on page 6-19, you must carefully consider the implications whenever a portlet tries to communicate with the portal (or the portal communicates with the portlet). For example, suppose a portlet or JSP places data in the request for the *doobie* portlet to process; if portlet *doobie* is asynchronous, it is running on a different request and will never see the data. Because of this behavior, there will be cases when you should not use asynchronous portlets in your implementation.

Thread Safety and Asynchronous Rendering

If you use asynchronous portlet content rendering, be sure that your code (for example, in backing files) is thread safe. The portal framework handles the major issues outside of a developer's control, such as concurrent access to the request and response; and it manages coordination of issues such as waiting for all async operations to finish and assembling the results in the correct order. But the portlet developer has the responsibility for ensuring that the user code is thread safe.

This consideration also applies to parallel (forked) portlet processing.

Considerations for IFRAME-based Asynchronous Rendering

Some special considerations associated with IFRAME-based asynchronous rendering include:

- IFRAME rendering varies depending on the browser. Making an IFRAME implementation seamless to an end user involves several factors, such as proper skin/skeleton development conventions, cross-browser development, and so on.
- If the content is larger than the IFRAME region, horizontal and/or vertical scrolling will be enabled. Be careful of content which itself contains scrolling regions, as it can be difficult to manipulate all scrolling regions to view all embedded content.
- IFRAME rendering might complicate other aspects of portal development, such as cross-portlet drag and drop, which is used in the GroupSpace sample application.
- IFRAME rendering works whether or not Javascript is enabled.
- You can disable asynchronous portlet content rendering for certain operations by using the `<render:context>` tag or the `AsyncContentContext` class as described in "Disabling Asynchronous Rendering for a Single Interaction" on page 6-19; however, these mechanisms do not work correctly when IFRAME-based asynchronous rendering is used. To avoid this issue, turn off asynchronous rendering or use AJAX-based asynchronous rendering.

Considerations for AJAX-based Asynchronous Rendering

Some special considerations associated with Asynchronous JavaScript and XML (AJAX)-based asynchronous rendering include:

- AJAX technology relies on Javascript. If users disable Javascript in their browser, AJAX-based portlets will be broken (the content will never render).
- This mechanism might not be compatible with other AJAX mechanisms, such as those that might typically be used by content authors to dynamically populate forms, for example. Generally speaking, a best practice is to either let WebLogic Portal manage AJAX at the portal level, or turn off AJAX for a portlet if you intend to incorporate AJAX behaviors into your portlet.
- The current AJAX implementation does not support XHTML. The implementation performs DOM operations that are known not to work in some browsers when using an

XHTML content type. This problem arises when a Look & Feel skeleton is configured to use an XHTML content type.You can avoid this problem by doing one of two things:

- Use an HTML content type for the portal
- Use the IFRAME-based implementation of async portlet rendering

Comparison of IFRAME- and AJAX-based Asynchronous Rendering

Table 6-2 summarizes the advantages and disadvantages of IFRAME- and AJAX-based asynchronous rendering. BEA recommends AJAX as a more robust implementation.

Table 6-2 IFRAME-based and AJAX-based Asynchronous Portlet Summary Table

Type	Advantages	Disadvantages
IFRAME	Works with Javascript enabled or disabled Support for embedded media (non-HTML) files Supports XHTML.	Generally perceived as providing a less intuitive user experience Can complicate more full-featured portlet development tasks, such as cross-portlet drag and drop
AJAX	Generally perceived as providing a more intuitive user experience Provides a single document for full-featured portlet development tasks, such as cross-portlet drag and drop Provides better Look & Feel integration	Works only with Javascript enabled Does not currently support XHTML

Comparison of Asynchronous and Conventional or Forked Rendering

The following table compares some of the behavior and features of conventional or forked rendering and asynchronous portlet content rendering.

Table 6-3 Comparison of Behaviors - Forked/Conventional Rendering and Asynchronous Rendering

Behavior/Feature	Forked or Conventional Rendering	Asynchronous Rendering
Completed rendering of page	Page does not render until all portlet processing is complete	Page, and portlet frames, render immediately; individual portlet content renders as processing completes
HTML page	No changes between conventional rendering and forked rendering	Page uses AJAX or IFRAME for rendering.
Rendering requests	Requires only one request.	Requires *n* + 1 requests (where *n* is the number of asynchronous portlets) True only for page requests; when interacting with an individual portlet, only one request is required.
Refresh	Entire page refreshes when interaction occurs on any portlet	Refresh required only for an individual portlet.
IPC Support	IPC supported	IPC not supported, although some workarounds exist for AJAX asynchronous portlets.
Page request/response	Server response to page request includes content of page	Portal page does not include portlet content (less information needs to be returned by the server); page starts loading faster

Portal Life Cycle Considerations with Asynchronous Content Rendering

This section provides more information about life cycle and control tree implications associated with using asynchronous content rendering.

For the *initial* request for a portal page, backing files attached to the portlet will run in the context of a full portal control tree. However, portlet content—such as page flows, managed beans, JSP pages, and so on—will not run for this initial request.

The values for the above-referenced APIs will be:

```
PortletBackingContext.isAsyncContent() = true
PortletBackingContext.isContentOnly() = false
```

For the *subsequent* content requests, backing files attached to the portlet, and the portlet content itself—such as page flows, managed beans, JSP pages, and so on—will run in the context of a "dummy" control tree.

The values for the above-referenced APIs will be:

```
PortletBackingContext.isAsyncContent() = true
PortletBackingContext.isContentOnly() = true
PortletPresentationContext.isAsyncContent() = true
PortletPresentationContext.isContentOnly() = true
```

An important consequence of this model is that when asynchronous content rendering is enabled for portlets, the portlet content will run in isolation from the rest of the portal. Such portlets therefore cannot expect to have direct access to other portal controls such as books, pages, desktops, other portlets, and so on.

Asynchronous Content Rendering and IPC

Although IPC is not supported when asynchronous content rendering is enabled, WebLogic Portal provides some features that allow these two mechanisms to coexist in your portal environment. In addition, you can disable asynchronous rendering for single requests using the mechanisms described in this section.

This section also applies to HTTP redirects.

Note: The techniques described in this section do not currently work with IFRAME portlets.

File Upload Forms

For forms containing file upload mechanisms, the WebLogic Portal framework automatically causes page refreshes without the need for any workarounds.

Disabling Asynchronous Rendering for a Single Interaction

Generally, if a portlet needs to disable asynchronous content rendering for a single interaction (such as a form submission, link click, and so on), you should use the mechanism described in this section.

Tip: When you use these mechanisms to disable asynchronous rendering, the portlet's action/rendering will be performed using two requests. The portlet's action is performed in the page request, while the portlet's rendering is performed on a subsequent request.

Ensure that your action does not use request attributes to pass information to your JSP page.

From a JSP page:

```
<render:controlContext asyncContentDisabled="true">
   Form, anchor, etc. would appear here
   (that is, <netui:form action="submit"…)
</render:controlContext>
```

From Java code:

```
try {
AsyncContentContext.push(request).setAsyncContentDisabled(true);
// Code that generates a URL would appear here
} finally {
AsyncContentContext.pop(request)
}
```

URL Compression

URL compression interferes with some of the AJAX-specific mechanisms for page refreshes. Because of this, URL compression must also be disabled whenever asynchronous content rendering is disabled to force page refreshes. WebLogic Portal disables URL compression automatically except when file upload forms are used; in this situation, you must explicitly disable it. Use the following examples as a guide:

From a JSP page:

```
<render:controlContext urlCompressionDisabled="true">
   Form, anchor, etc. would appear here
   (that is, <netui:form action="submit"…)
</render:controlContext>
```

From Java code:

```
try {
UrlCompressionContext.push(request).setUrlCompressionDisabled(true);
// Code that generates a URL would appear here
} finally {
UrlCompressionContext.pop(request)
}
```

For more information about implementing URL compression, refer to the *Portal Development Guide*.

CHAPTER 7

Local Interportlet Communication

Interportlet communication (IPC)—also called portlet-to-portlet communication—allows multiple portlets to use or react to data. For example, you might want to use IPC in a self-service or sales implementation where common data elements, such as order ID or customer ID, are used across multiple projects. All portlet types supported by WebLogic Portal can implement IPC. Examples of IPC include:

- A page flow portlet talks to a non-page flow portlet using the page flow's outer (portal) request.
- A non-page flow portlet talks to a page flow portlet, using the ActionResolver class.

IPC in WebLogic Portal is based on the use of event handlers—objects that listen for predefined events on other portlets in the portal and fire actions when that event occurs. You can set up interportlet communication in two ways: using the Workshop for WebLogic interface, or using the WebLogic Portal API.

This chapter includes a tutorial-based example of establishing interportlet communications using an out-of-the-box portal event handler ("Basic IPC Example" on page 7-13). This example will familiarize you with event handlers and show you some of their common uses.

This example is specific to interportlet communications within a single portal web project. For information on establishing IPC with federated portals (WSRP), refer to the *Federated Portals Guide*.

Note: IPC is not compatible with asynchronous portlet rendering, but workarounds can allow them to co-exist in some cases. For details, refer to "Asynchronous Content Rendering and IPC" on page 6-19.

This chapter includes the following sections:

Definition Labels and Interportlet Communication

IPC behavior is based on portlet definition labels; that is, all portlet instances of a given `.portlet` file respond to the same events. You can use the event handler options *Only If Displayed* and *From Self Instance Only* to discriminate among the instances of the same `.portlet` file. For a description of these options, refer to "Portlet Event Handlers Wizard - Add Handler Field Descriptions" on page 7-7.

Portlet Events

Portlet events (not to be confused with page flow events) allow portlets to communicate. One portlet can create an event and other portlets can listen for that event. A portlet event can also carry accompanying data called a *payload*, where the payload is a serializable Java object.

This section contains the following topics:

Event Handlers

Event handlers listen for events raised on subscribed portlets and fire one or more actions when a specific event is detected. An event handler tag is a child of the `<portlet>` tag, and a portlet can have any number of events associated with it. The following event handlers are available with WebLogic Portal:

- Portal (framework) - Responds to a portal framework event on a portlet by firing an action.

- Page Flow- Fires an action when an event occurs on that portlet.

  You can define a page flow event handler (on that portlet) that responds to these events and performs actions, such as to notify other portlets (that is, raise a custom event) or invoke a backing file call-back method, and so on.

- Struts - Responds to an event on a portlet by firing a struts action.

- Custom - Responds to an event that you define.

  A custom event handler is triggered by an event and can pass a developer-defined payload or fire any predefined action. Custom event handlers can be triggered declaratively or they can be based on a methods called in a backing file. You can specify that an event should be handled by a method in a backing file.

- Generic - Allows you to set up an event that will fire in several possible situations. For details, see Generic Event Handlers below.

Note: Java Server Faces event handlers are not supported out-of-the-box with WebLogic Portal using the existing declarative event handling mechanisms. However, it is possible to associate a backing file with the <netuix:portlet/> that contains the JSF content. The backing file can transform events from Weblogic Portal to an appropriate JSF bean (a "managed bean"). In the case of a breadcrumb style event, for example, JSF portlet_1 (with a backing file) could have a form where some user data is submitted. The backing file for JSF portlet_1 gets the data from the request and updates a list within a JSF managed bean. JSF portlet_2 then displays this list of data using an HTML table (using JSF tags) databound to the list in the JSF managed bean.

To send events from JSF to WebLogic Portal, it is possible to use the PortletBackingContext to fire events from the JSF application back into WebLogic Portal during the action phase. More information about this process will be available in a future documentation release or dev2dev article.

Generic Event Handlers

The generic event handler, with an event attribute value of *myEvent*, will be triggered on the following conditions:

- A custom event with event=*myEvent* is fired within the portal.
- A page flow action with name *myEvent* is raised by a portlet within the portal.
- The same conditions to which the `<handlePortalEvent event=myEvent>` handler would react.

- A generic event (see below) with event=*myEvent* is fired within the portal.

Using a generic event handler allows you to more effectively decouple your portal design, because your application does not need to know the source or type of an event. You can change the portlet type (for example, from a page flow portlet to a JSP portlet, with a backing file firing custom events) without affecting how you events are processed.

Event Types

An event action depends upon the type of event being raised. Except for portal events, all other events can be identified in the **Events** field on the Portlet Event Handlers Wizard, as described in "Portlet Event Handlers Wizard Reference" on page 7-5. Events available with the portal event handler are listed in Table 7-1.

Table 7-1 Events Available to a Portal Event Handler

This event...	Fires an action when the portlet...
onActivation	Becomes visible
onDeactivation	Ceases to be visible
onMinimize	Is minimized
onMaximize	Is maximized
onNormal	Returns to its normal state from either a maximized or minimized state
onDelete	Is deleted from the portal
onHelp	Enters the help mode
onEdit	Enters the edit mode
onView	Enters the view mode
onRefresh	Is refreshed
on`CustomEvent`	Mode change to the custom mode `CustomEvent` Refer to "Event Handlers" on page 7-2.

Event Actions

Event handlers fire an action on the host portlet when that handler detects an event from another portlet in the application (or possibly the same portlet, for example in the case of a page flow portlet). For example, when the user minimizes the appropriate portlet, a portal event called *onMinimize* might cause the handler listening for it to fire an action that invokes an attached backing file.

Table 7-2 lists the event actions available for portlets.

Table 7-2 Event Actions

This action...	Has this effect...
Change Window Mode	Changes the mode from its current mode to a user-specified mode; for example, from help mode to edit mode.
Change Window State	Changes the state from its current state to a user-specified state; for example, from maximized to delete state.
Activate Page	Opens the page on which the portlet currently resides.
Fire Generic Event	Fires a user-specified generic event.
Fire Custom Event	Fires a user-defined custom event.
Invoke BackingFile Method	Runs a method in the backing file attached to the portlet. For more information on backing files, refer to "Backing Files" on page 5-64.

Portlet Event Handlers Wizard Reference

The Portlet Event Handlers wizard included in Workshop for WebLogic allows you to implement several types of event handlers and actions without programming. The following steps summarize the process of setting up an event handler using the wizard:

1. Select a type of event handler to create.
2. Determine the portlets to which that handler will listen.
3. Select an event for which the handler will listen.
4. Select and configure an action to fire when the event occurs.

The following sections describe the dialogs of the wizard and provide information about the information required in each field of the dialogs.

For a specific procedural example of how to use the event handler wizard, refer to "Basic IPC Example" on page 7-13.

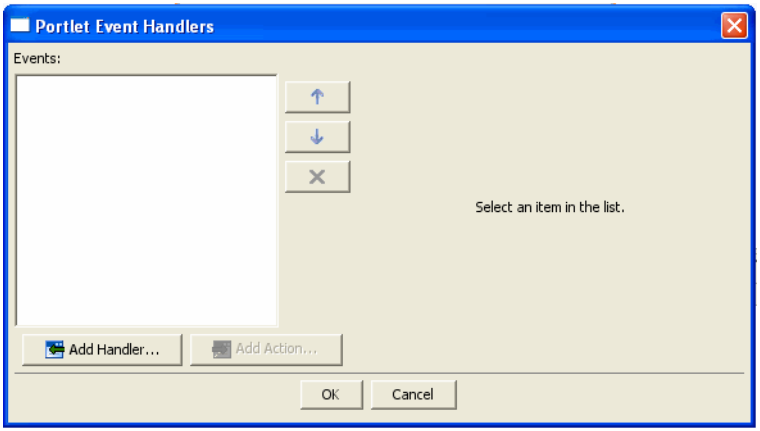
Portlet Event Handlers Wizard Dialogs

The wizard opens when you open a portlet in Workshop for WebLogic and click the ellipsis button  next to Event Handlers in the Properties view.

Note: If no event handlers have been added, the Event Handler field indicates that. If any event handlers have been added, the field indicates the number that currently exist.

The wizard appears, as shown in Figure 7-1.

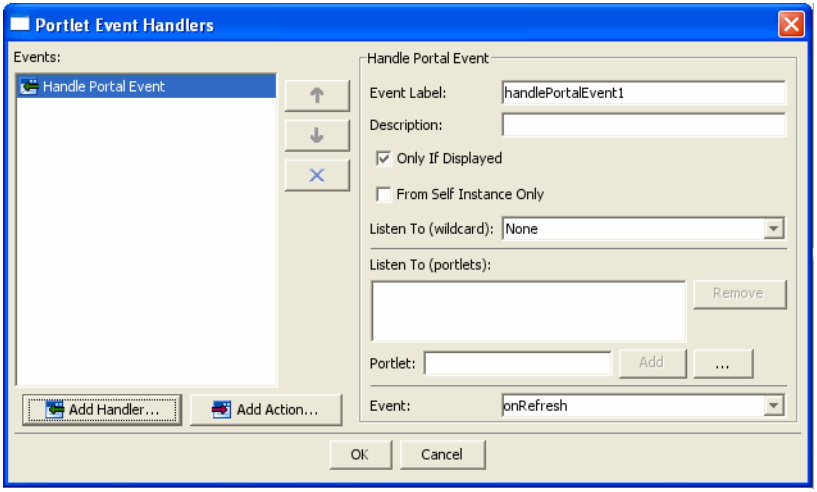
Figure 7-1 Portlet Event Handlers Wizard



When you click **Add Handler**, the event handler drop-down menu allows you to select a handler; to add an action, click **Add Action** to open the event action drop-down menu.

Based on your selection, the dialog box expands, displaying additional fields that you can use to set up the handler or action. Figure 7-2 shows an example of the expanded dialog for adding an event handler.

Figure 7-2 Expanded Event Handlers Dialog



Portlet Event Handlers Wizard - Add Handler Field Descriptions

Table 7-3 explains the fields in the Add Handler dialog and how your selections affect the behavior of the event.

Table 7-3 Portlet Event Handlers Wizard - Add Handler

Field	Description
Event Label	Required. This identifier can be used by the `<filterEvent>` tag in the portal file to distinguish multiple event handlers in the same portlet.
Description	Optional.
Only If Displayed check box	Optional. Indicates that the portlet to receive the event must be on the current page and not minimized or maximized—the portlet's content must be currently in a rendered state. (Remember that the user must also be entitled to see the portlet.) The default is `true`. **Note:** If the event is <handlePortalEvent event="onMinimize" fromSelfInstanceOnly="true"> then it is logically impossible for this event to fire if onlyIfDisplayed="true".

Table 7-3 Portlet Event Handlers Wizard - Add Handler (Continued)

Field	Description
From Self Instance Only checkbox	Optional. Defines whether the handler for a given portlet instance is invoked only when the source event originates from that instance. The default is `false`. If From Self instance Only is set to true, any Listen To values are ignored.
Listen To (wildcard)	Optional. Identifies the portlet(s) that this portlet can listen to. The values include: • This – The definition label of this portlet • None • Any **Note:** Currently, None and Any are functionally equivalent. **Note:** If both Listen to (wildcard) and Listen To (portlets) are defined, the system will "union" their values during processing; that is, if the wildcard is "this," then the owning portlet definition label will be added to those in Listen To (portlets), and if the wildcard is 'any" then the value of Listen To (portlets) is ignored.
Listen To (portlets)	Optional. Allows you to specify the portlets that this portlet can listen to. You can choose a .portlet file from the file system by clicking the '...' button). When you select a .portlet file and hit Open, the portlet is added to the Listen To list. **Caution:** The values that you enter here are not validated. A typo in either an event name or a definition label can be very difficult to resolve later. **Note:** When you click **Open**, the definition label is also added to the Listen To list and the Add button is enabled.Although the enabled Add button might make it appear that the portlet still needs to be added, it does not.

Table 7-3 Portlet Event Handlers Wizard - Add Handler (Continued)

Field	Description
Portlet	You can type a portlet name in the field and click **Add**, or click the browse button to navigate to the portlet for which you want to listen.
Event or Action	Depending on the event handler you added, you will choose an event or an action for which the portlet will listen. For example, if you added the `HandlePortalEvent` handler, you can use the **Event** drop-down menu to select portal events, such as the `onRefresh` event. If you choose a handler that exposes actions, type the name of the action in the **Action** field. For example, if you chose `HandlePageFlowEvent`, you could type `submitReport`. The `submitReport` action of the page flow is now visible in the **Action** drop-down menu.

Portlet Event Handlers Wizard - Add Action Field Descriptions

The available fields for the action depend on the type of action that you select. Table 7-4 explains the possible fields in the expanded Add Action dialog and how your selections affect the behavior of the action.

Table 7-4 Portlet Event Handlers Wizard - Add Action

Field	Description
Change Window Mode	Enter the value of the new window mode.
Change Window State	Enter the value of the new window state; possible values are normal, minimized, maximized.
Activate Page	This action activates the page on which the portlet `<portlet_def_id>` currently resides. This action will fire only when triggered during the handlePostBack life cycle. Do not select the Activate Page action if the Only If Displayed check box is selected. Logically, if the portlet is responding to the event only if it is displayed, the page that it is on must be active anyway.
Invoke Struts Action	Valid only for Struts portlets. Use this selection to cause a struts action to be raised. The value must be an unqualified name of a struts action defined in the embedded content.

Table 7-4 Portlet Event Handlers Wizard - Add Action

Field	Description
Fire Generic Event	Use this selection to cause a generic event to be raised. Enter the name of the generic event.
Fire Custom Event	Use this selection to cause a custom event to be raised. Enter the name of the custom event.
Invoke BackingFile Method	Use this selection to cause a backing file method to run. Enter the name of the method that you want to invoke. This option displays in the Add Action selection list only if you have an existing backing file in the project.
Invoke Page Flow Action	Use this selection to cause a page flow action to be raised.

IPC Example

This section contains the following topics:

- Before You Begin - Environment Setup
- Basic IPC Example

Before You Begin - Environment Setup

Before you use the interportlet communication example in this chapter, you must have an existing portal development environment, consisting of a domain, Portal EAR project, Portal Web project, Datasync project, and portal. To complete the pre-requisite tasks, perform the tasks described in the *Getting Started with WebLogic Portal* tutorial, using the information in Table 7-5 to enter the necessary values.

1. Create a Portal domain (server).
2. Create a Portal EAR project.
3. Associate the EAR project with the server.
4. Create a Portal web project.
5. Create a portal.

Table 7-5 IPC Example - Environment Setup Values

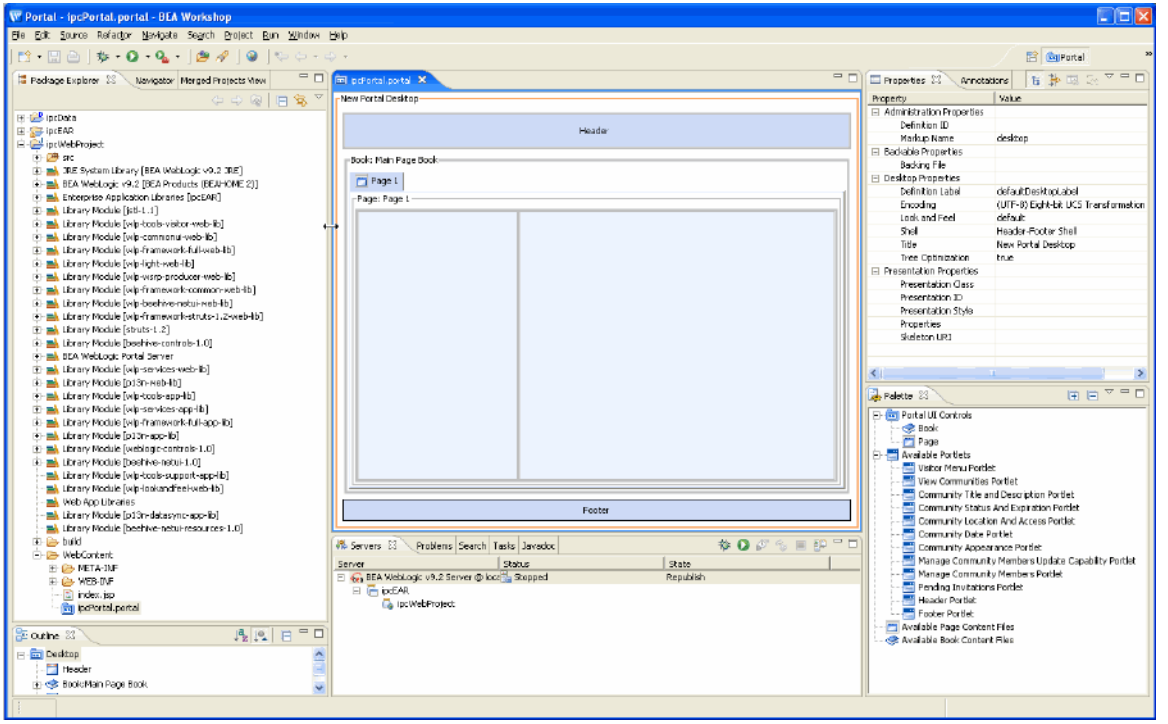
Setup Information	Notes/Values
Domain Configuration Wizard - Welcome	Create a new WebLogic domain (the default)
Domain Configuration Wizard - Select Domain Source	In the **Generate a domain configured automatically to support the following BEA products** list, select **WebLogic Portal**. When you do this, other components are selected automatically; *keep all of them selected.*
Domain Configuration Wizard - Configure Administrator Username and Password	User name: `weblogic` User password: `weblogic` Confirm user password: `weblogic`
Domain Configuration Wizard - Configure Server Start Mode and JDK	Development Mode (the default) JRockit SDK
Domain Configuration Wizard - Customize Environment and Services Settings	No (the default)
Domain Configuration Wizard - Create WebLogic Domain	Domain name: `ipcDomain` Domain location: Accept the default, or specify another directory on your system.
Portal EAR Project Wizard	EAR Project Name: `ipcEAR` Switch to the Portal Perspective if you are not already using it.
Servers view	Right-click the server in the Servers view and select **Add and Remove Projects** Associate the `ipcEAR` project with the portal domain `ipcDomain`.

Table 7-5 IPC Example - Environment Setup Values (Continued)

Setup Information	Notes/Values
Portal Web Project Wizard	Web Project Name: `ipcTestWebProject` In the **Add project to an EAR** checkbox: Check the box and add to `ipcEAR`
Portal Wizard	Right-click the `ipcWebProject/WebContent` folder and select **New** > **Portal** Portal Name: `ipcPortal`

Figure 7-3 shows how your workbench should look after you complete the pre-requisite tasks:

Figure 7-3 Workbench with Portal Perspective and Merged Projects View - Completed IPC Pre-Setup



With a development environment set up, you can complete the steps described in this section:

- Basic IPC Example

In this exercise, you create individual page flows, portlets, JSPs, and backing files to establish interportlet communications within the portal project. You then add these portlets to a portal and test the project to ensure that communication is successful.

Basic IPC Example

This section describes the process of setting up interportlet communications between two portlets by using the Portal Event Handlers wizard in Workshop for WebLogic. This is a simple example in which minimizing one portlet changes the text string in another portlet in the portal.

You should become familiar with the Portal Event Handlers Wizard and backing files before attempting to replicate this example. For more information about the wizard, refer to "Portlet Event Handlers Wizard Reference" on page 7-5. For more information on backing files, refer to "Backing Files" on page 5-64.

This exercise includes five main tasks:

1. Create the Portlets
2. Create the Backing File
3. Attach the Backing File
4. Add the Event Handler to bPortlet
5. Test the Project

Create the Portlets

In this section, you create two JSP files and the JSP portlets that surface these files. You also create a backing file that contains the instructions necessary to complete the communication between the two portlets, and you add an event handler to one of the portlets. After you have created the portlets and attached the backing file, you test the project in your browser.

Note: Before continuing with this procedure, ensure that Workshop for WebLogic is running and the ipcWebProject node is expanded.

Create the JSP Files and Portlets

To create the JSP files that the portlets will surface, do the following:

1. Under the ipcWebProject node, double-click `index.jsp`.

`index.jsp` opens in the workbench editor, displaying the source code.

2. Replace the body text with the phrase Minimize Me! as shown in figure

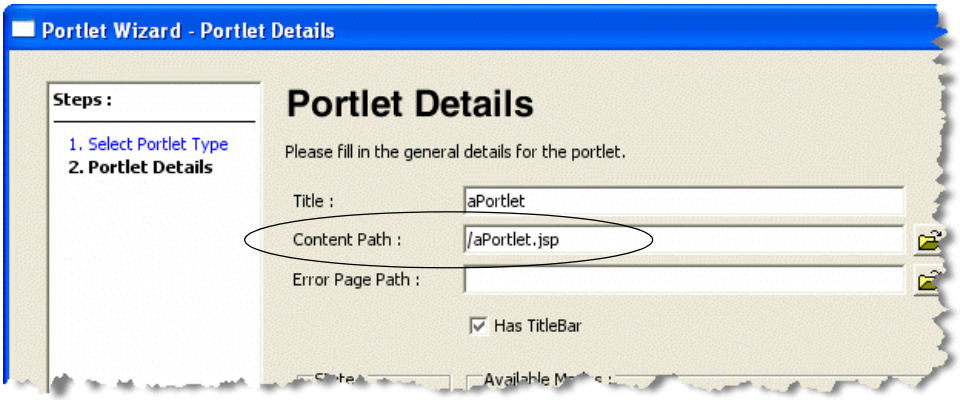
Figure 7-4 index.jsp after Editing the Body Text in the Workbench Editor

3. Save the file as `aPortlet.jsp`

4. Right-click `aPortlet.jsp` in the Package Explorer view and select **Generate Portlet** from the context menu.

   The Portal Details dialog appears (Figure 7-5). with `aPortlet.jsp` in the Content Path field.

Figure 7-5 Portal Details Dialog Box for a Portlet



5. Select **Minimizable** and **Maximizable** and click **Create**.

   `aPortlet.portlet` appears in the `ipcWebProject/WebContent` folder in the Package Explorer view.

6. In the same directory, make a copy of aPortlet.jsp and give the name `bPortlet.jsp` to the copy.

7. Open `bPortlet.jsp` in the workbench editor if it is not already open.

   The XML code for the JSP file appears.

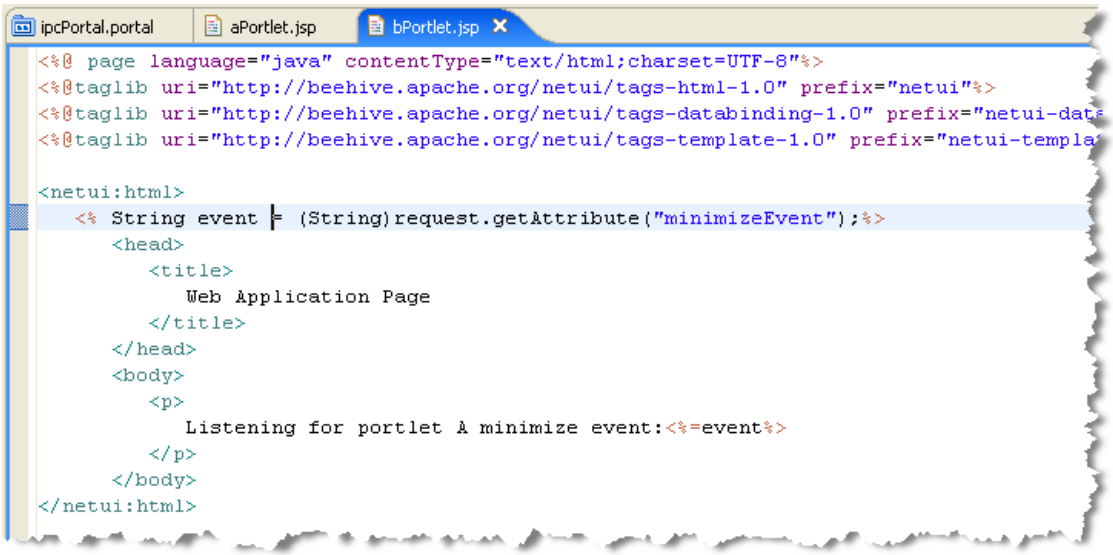
8. Copy the code from Listing 7-9 into the JSP, replacing everything from `<netui:html>` through `</netui:html>`. This code displays event handling from the backing file that you will create and attach in a subsequent step.

Listing 7-9 New JSP Code for bPortlet.jsp

```
<netui:html>
   <% String event = (String)request.getAttribute("minimizeEvent");%>
      <head>
         <title>
            Web Application Page
         </title>
      </head>
      <body>
         <p>
            Listening for portlet A minimize event:<%=event%>
         </p>
      </body>
</netui:html>
```

The source should look like the example in Figure 7-6.

Figure 7-6 Updated bPortlet JSP Source



9. Save the file.
10. Following the same steps you used previously, generate a portlet from the `bPortlet.jsp` file.

Checkpoint: At this point the `ipcWebProject/WebContent` folder contains these files: `aPortlet.jsp`, `aPortlet.portlet`, `bPortlet.jsp`, and `bPortlet.portlet`.

Create the Backing File

To create the backing file, do the following:

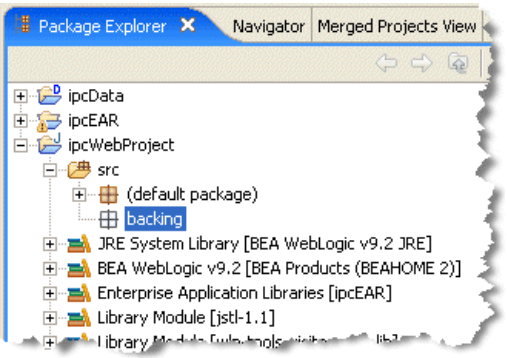
1. In `ipcWebProject`, right-click the `src` folder and select **New** > **Folder** from the menu.

   The Create New Folder dialog box appears.

2. Create a folder called `backing`.

   The folder `backing` will appear under `ipcWebProject/src`, as shown in Figure 7-7.

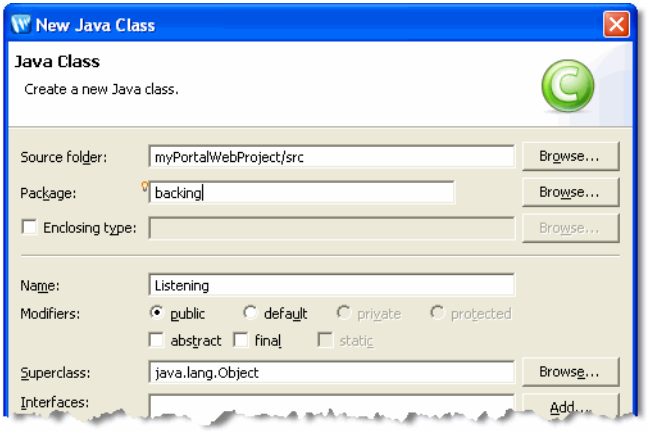
Figure 7-7 New Backing File Folder in Package Explorer View



3. Right-click the `backing` folder and select **New > Other**.
4. In the **New – Select a wizard** dialog, select **Java > Class**, and click **Next**.

   The **New Java Class** dialog appears, as shown in Figure 7-8. The **Source folder** field auto-fills with the default path; leave it as is. The **Package** field auto-fills with `backing`; leave it as is.

Figure 7-8 New Java Class Dialog



5. In the **Name** field, enter `Listening` and click **Finish**.

   The new Java class appears in the editor.
6. Delete the entire default contents of `Listening.java`, and copy the code from Listing 7-10 into the file.

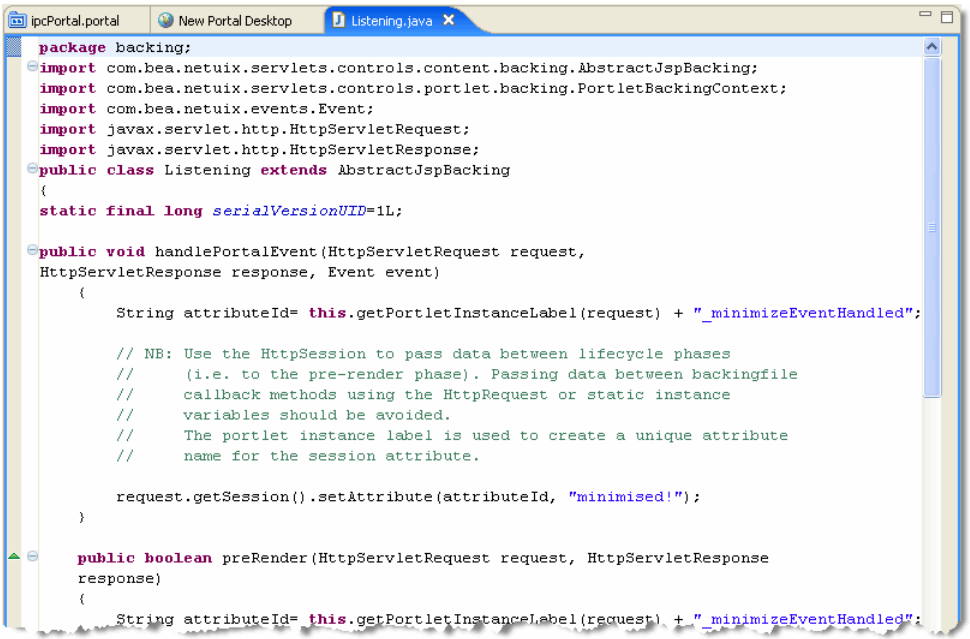
Listing 7-10 Backing File Code for Listening.java

```
package backing;

import com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking;
import com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext;
import com.bea.netuix.events.Event;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Listening extends AbstractJspBacking
{
   static final long serialVersionUID=1L;
   public void handlePortalEvent(HttpServletRequest request,
   HttpServletResponse response, Event event)
   {
      String attributeId= this.getPortletInstanceLabel(request) +
      "_minimizeEventHandled";

       // NB: Use the HttpSession to pass data between lifecycle phases
       //     (that is, to the pre-render phase). Passing data between
       //     backing file callback methods using the HttpRequest or static
       //     instance variables should be avoided.
       //     The portlet instance label is used to create a unique
       //     attribute name for the session attribute.

      request.getSession().setAttribute(attributeId, "minimized!");
   }
   public boolean preRender(HttpServletRequest request, HttpServletResponse
   response)
   {
      String attributeId= this.getPortletInstanceLabel(request) +
      "_minimizeEventHandled";
         if (request.getSession().getAttribute(attributeId) != null)
         {
            // Reset the session flag
             request.getSession().removeAttribute(attributeId);

            // Pass minimize event notification to the JSP via the request.
            request.setAttribute("minimizeEvent", "Minimize event handled");
         }
         else
         {
            request.setAttribute("minimizeEvent", null);
         }
      return true;
   }

   private String getPortletInstanceLabel(HttpServletRequest request)
   {
```

```
        PortletBackingContext context=
        PortletBackingContext.getPortletBackingContext(request);
        return context.getInstanceLabel();
    }
}
```

Figure 7-9 shows the top portion of the Listening.java file as it should look after you paste the code into it.

Figure 7-9 Listening.java with Updated Backing File Code



7. Save `Listening.java`.

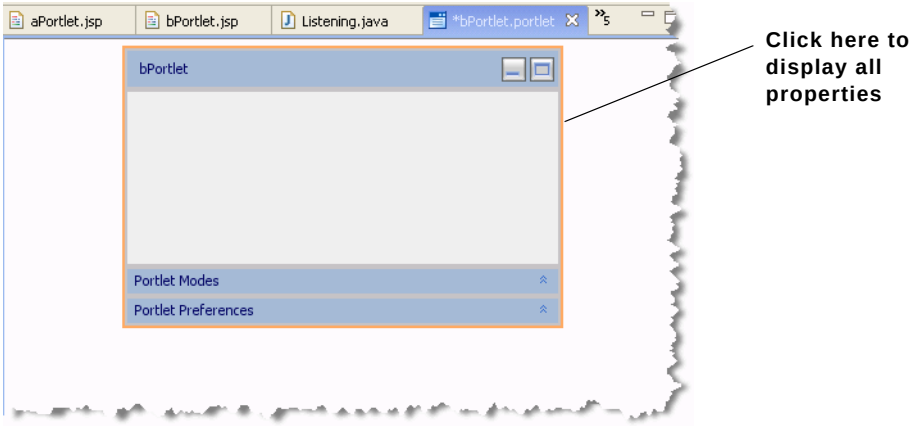
Attach the Backing File

Now you will attach the backing file created in the previous section to `bPortlet.portlet`. Perform the following steps:

1. In the Package Explorer, double-click `bPortlet.portlet` to open it.

2. Click on the portlet in the editor, if needed, to display the portlet's properties. You should see an orange border around the outside of the portlet, as shown in Figure 7-10.

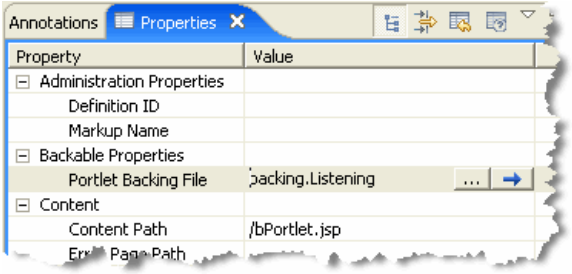
Figure 7-10 bPortlet with Outer Border Selected to Display Properties



Tip: The Properties view is a default view in the Portal perspective. If it is not visible, select **Window > Show View > Properties**.

3. In the Properties view, enter `backing.Listening` into the **Backable Properties > Portlet Backing File** field, as shown in Figure 7-11.

Figure 7-11 Attaching the Backing File in the Properties View



4. Save the portlet file.

Add the Event Handler to bPortlet

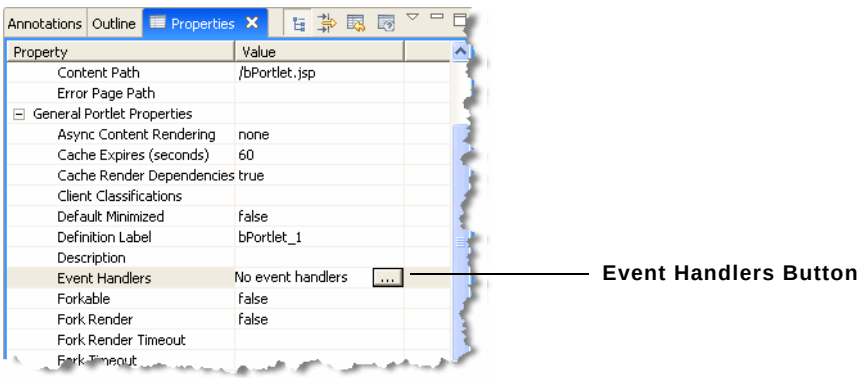
You now add the event handler to `bPortlet.portlet`. This handler will be set up so that it will listen for an event on a specific portlet and fire an action in response to that event. To add the event handler, perform the following steps:

Note: `bPortlet.portlet` should be displayed in the Workshop for WebLogic editor. If it isn't, locate it in the `producerWeb/WebContent` folder in the application panel and double-click it.

1. Click on the portlet in the editor if needed to display its properties.

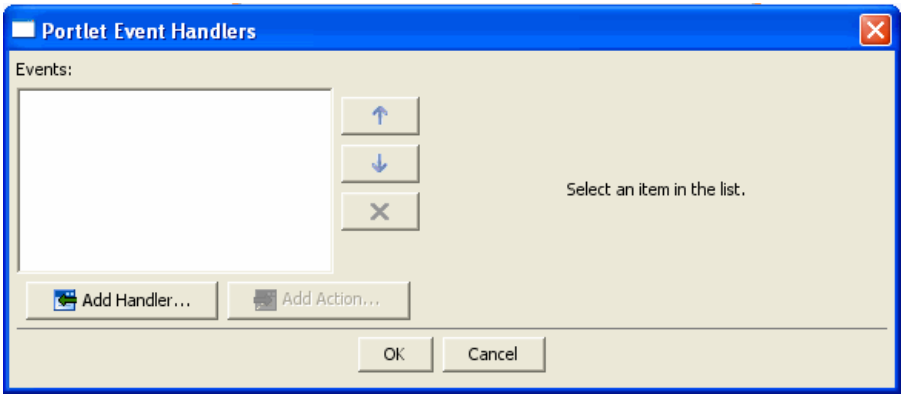
1. In the Properties view, click in the **Value** column of the **Event Handlers** property. A browse button [...] appears, as shown in Figure 7-12.

Figure 7-12 Event Handlers Button



2. Click the ellipsis button [...] to display the Portlet Event Handlers dialog, as shown in Figure 7-13.

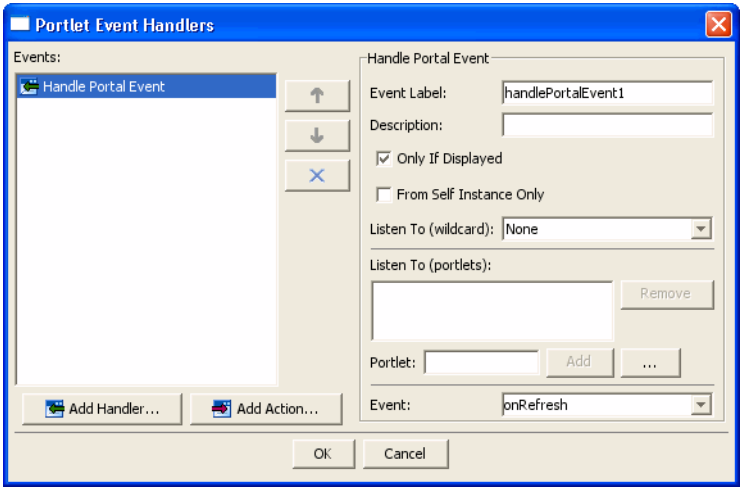
Figure 7-13 Portlet Event Handlers Dialog Box



3. Click **Add Handler** to open the **Event Handler** drop-down list.
4. From the drop down list, select **Handle Portal Event**.

   The Portlet Event Handlers dialog box expands to allow entry of more details, as shown in Figure 7-14.

Figure 7-14 Event Handler Dialog Box Expanded



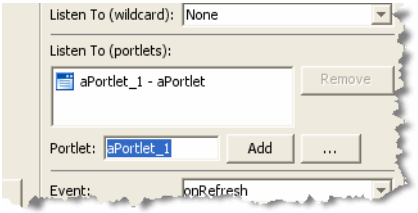
5. Accept the defaults for all fields except **Portlet**.
6. In the **Portlet** field, click the ellipses button ... .

The **Please Choose a File** dialog appears.

7. Click `aPortlet.portlet` and click **OK**.

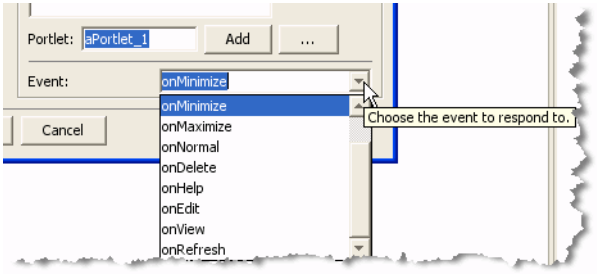
   The dialog box closes and **aPortlet_1** appears in the **Listen to (portlets)**: list and in the **Portlet** field, as shown in Figure 7-15. The label **aPortlet_1** is the definition label of the portlet to which the event handler will listen.

Figure 7-15 Adding portlet_1



8. Click the **Event** drop-down control to open the list of portal events that the handler can listen for and select **onMinimize**, as shown in Figure 7-16.

Figure 7-16 Event Drop-down List



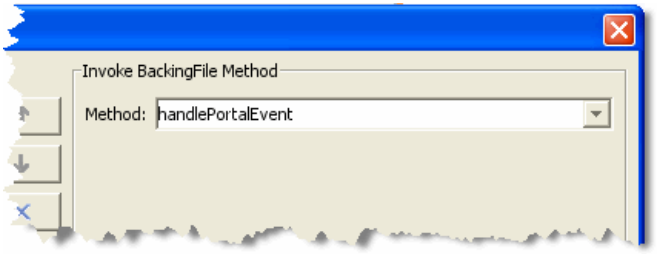
9. Click **Add Action** to open the action drop-down list and select **Invoke BackingFile Method**.

   The Invoke BackingFile selection will not appear unless a backing file is detected by WebLogic Portal.

10. In the **Method** field, enter **handlePortalEvent**, as shown in Figure 7-17.

    The dropdown menu for this field displays the last several values that you entered, if applicable.

Figure 7-17 Adding the Backing File Method



11. Click **OK**.

    The event handler is added. Note that the **Value** field of the **Event Handlers** property now indicates `1 Event Handler`.

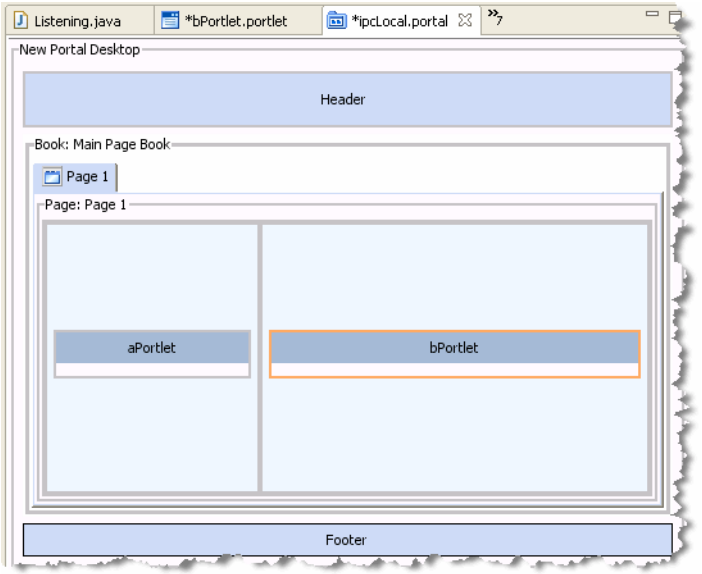
Test the Project

Test the communication between your portlets by following these steps:

Note: Before you begin, ensure that all files are saved.

1. Select `ipcPortal.portal` to display it in the workbench editor.
2. Drag both `aPortlet.portlet` and `bPortlet.portlet` from the Package Explorer view onto the portal layout, as shown in Figure 7-18.

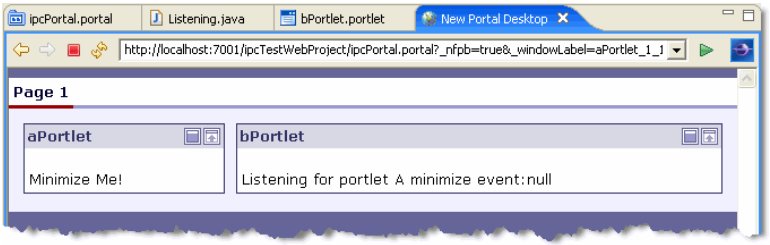
Figure 7-18 Portal Layout with aPortlet and bPortlet Added



3. Save the portal.
4. Run the portal. To do this, right-click `ipcPortal.portal` in the Package Explorer view and select **Run As > Run on Server**.
5. At the **Run On Server – Define a New Server** dialog, click **Finish**.

   Wait while the server starts and the application is published to the server. The portal will render in your browser (Figure 7-19).

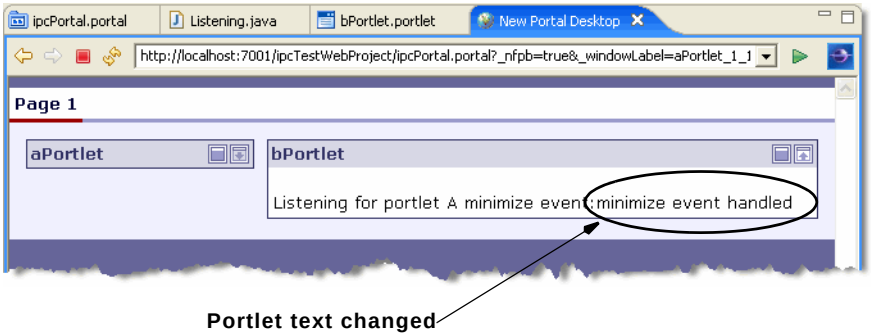
Figure 7-19 ipcLocal Portal in Browser



6. Click the minimize button to minimize aPortlet.

   Note the content change in bPortlet, as shown in Figure 7-20.

Figure 7-20 ipcPortal Showing the Effect of Minimizing aPortlet



Summary

In this example, you set up your environment and you added two JSP portlets to a local portal. One portlet, aPortlet, was fairly simple, while the second portlet, bPortlet, surfaced a more complex JSP file, used a backing file, and contained a portal event handler. When you tested the communication between the portlets, you observed how the bPortlet changed when an event occurred on aPortlet. This is called local interportlet communication.

IPC Special Considerations and Limitations

The following sections describe special considerations that you should keep in mind as you implement interportlet communications.

This section contains the following topics:

- Using Asynchronous Portlet Rendering with IPC
- Generic Event Handler for WSRP
- Consistency of the Listen To Field

Using Asynchronous Portlet Rendering with IPC

Although IPC is not supported when asynchronous content rendering is enabled, WebLogic Portal provides some features that allow these two mechanisms to coexist in your portal environment. In addition, you can disable asynchronous rendering for single requests using the mechanisms described in “Asynchronous Content Rendering and IPC” on page 6-19.

Generic Event Handler for WSRP

Use a generic event handler to work with WebLogic Portal WSRP. To do this, first select **Generic Event Handler**, then select **Add Action** and select **Window Mode|State**. Then manually type in the event name—for example, `onMinimize`.

Consistency of the Listen To Field

Pay attention to the **Listen To** field when you set up the listener portlet. The portlet definition you use on the consumer *must match* the WSRP portlet's portlet definition. For example, if you have "portlet_2" listening to "portlet_1", the WSRP portlet corresponding to "portlet_1"—the proxy on the consumer—must also have its portlet definition label set to "portlet_1". For more information on using IPC with WSRP, refer to the *Federation Guide*.

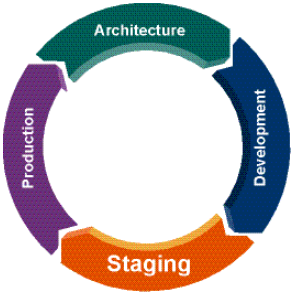
Part III Staging

Part III includes the following chapters:

- Chapter 8, “Assembling Portlets into Desktops”
- Chapter 9, “Deploying Portlets”

BEA recommends that you deploy your portal, including portlets, to a staging environment, where it can be assembled and tested before going live. In the staging environment, you use the WebLogic Portal Administration Console to assemble and configure desktops. You also test your portal in a staging environment before propagating it to a live production system.

For a view of how the tasks in this section relate to the overall portal life cycle, refer to the *WebLogic Portal Overview*.



CHAPTER 8

Assembling Portlets into Desktops

You perform the tasks described in this chapter to prepare the individual portlets that are part of your portal application for public consumption. After you add portlets to desktops, you can configure and test the application as a whole, and then deploy it to the production environment when it is ready for public access.

Before you perform the tasks described in this chapter, use the *Portal Development Guide* to create the framework into which you will add the portlets— this includes the portal and its menus, layouts, the Look & Feel components for the overall portal, and the framework of the actual desktop. Also, you must have already created the set of portlets in the portlet *library*, from which you will choose the portlets to add to the desktop.

The primary tools used in this chapter are the WebLogic Portal Administration Console, the WebLogic Portal Propagation Utility (to move database and LDAP data between staging, development, and production), WebLogic Server application deployment tools, and any external content or security providers that you are using.

This chapter contains the following sections:

Portlet Library

The WebLogic Portal Administration Console organizes portal resources in a tree that consists of Library resources and desktop resources. Understanding the relationship between Library and desktop resources helps you to understand the effects and consequences of propagation.

The following text describes the relationships between the following instances of portal assets:

- **Primary instance** – Created in WebLogic Workshop and stored in a `.portal` or `.portlet` file.
- **Library instance** – Created or updated in the Administration Portal, and displayed in the Portal Resources tree under the Library node.
- **Desktop instance** – Created or updated in the Administration Portal, and displayed in the Portal Resources tree under the Portals node.
- **Visitor instance** – Created or updated in the Visitor Tools.

For more details on portlets in libraries and in desktops, refer to the *Production Operations Guide*.

Managing Portlets Using the Administration Console

This section contains instructions for performing portlet-related tasks using the WebLogic Portal Administration Console.

This section contains the following topics:

Copying a Portlet in the Library

You can use this feature of the WebLogic Portal Administration Console to duplicate an existing portlet and use it as a template for a "new" portlet.

Perform these steps:

1. Expand the Library node in the Portal Resources tree and navigate to the portlet that you want to copy.
2. Click **Copy Portlet**. The Copy Portlet dialog displays.
3. Enter a title and description for the copied portlet.
4. Click **OK**. The portlet is added at the bottom of the portlet list.

You can now customize the copied portlet by modifying its properties and preferences.

Modifying Library Portlet Properties

Portlet properties include all of the features and elements that make up the portlet. As a portal administrator, you can modify some of these properties from the Details tab. You can also edit the title, description, and locale information from the Title & Description tab, as described below.

To modify the properties of a portlet that resides in the library, perform these steps:

1. Expand the Library node in the Portal Resources tree and navigate to the portlet that you want to modify.
2. From the Details tab, select the type of property that you want to change. Use the table below for guidance.

Table 8-1 Modifying Library Portlet Properties

Title and Description	
Change title and description of the portlet in the current locale	1. Click **Title & Description**. 2. Click the locale (for example, **en**) in the **Locale** cell; the Add a Localized Title & Description dialog displays. 3. Enter a new Title and/or Description. 4. Click **Update**.

Table 8-1 Modifying Library Portlet Properties (Continued)

Add a localized title for the portlet	1. Click **Title & Description**. 2. Click **Add Localized Title**; the Add a Localized Title & Description dialog appears. 3. Enter a Language and Country identifier, Variant if applicable, Title, and a Description for the localized title. 4. Click **Create**.
Portlet Preferences	Refer to "Creating a Portlet Preference" on page 8-9 and "Editing a Portlet Preference" on page 8-10.
Portlet Theme	1. Click **Appearance**; the Edit Appearance dialog displays. 2. From the drop-down menu, select a Theme. 3. Click **Update**.
Render caching and timeout	1. Click **Advanced Properties**. 2. In the Render Caching Enabled drop-down menu, select True or False. 3. If you selected True, enter a cache expiration value in the Cache Expiration field. 4. Click **Update**.

Modifying Desktop Portlet Properties

Portlet properties include all of the features and elements that make up the portlet. As a portal administrator, you can modify some of these properties from the Details tab. You can also edit the title, description, and locale information from the Title & Description tab, as described below.

To modify the properties of a portlet that resides on a desktop, perform these steps:

1. Expand the Portals node in the Portal Resources tree and navigate to the portlet that you want to modify.
2. From the Details tab, select the type of property that you want to change. Use the table below as a guide.

Table 8-2 Modifying Desktop Portlet Properties

Title and Description	You must edit these values within the Library resource tree. Expand the Library node, select the portlet that you want to edit, and follow the instructions in "Modifying Library Portlet Properties" on page 8-3.
Portlet Preferences	Refer to "Creating a Portlet Preference" on page 8-9 and "Editing a Portlet Preference" on page 8-10.
Portlet Theme	1. Click **Appearance**; the Edit Appearance dialog displays. 2. From the drop-down menu, select a Theme. 3. Click **Update**.

Deleting a Portlet

You can delete portlets from the Administration Console only if they were created there; for example, if you used the Copy Portlet feature to duplicate the portlet. Portlets created in Workshop for WebLogic cannot be deleted using the Administration Console.

Perform these steps:

1. Expand the Library node in the Portal Resources tree and navigate to the portlet that you want to delete.
2. Click **Delete Portlet**.

Managing Portlets on Pages

The contents of a page include portlets and books. You can view the portlets that are already on your page, and add and remove portlets to construct your page.

Adding Portlets to a Page

Library: To add a content to a page, perform these steps:

1. In the Portal Resource tree, expand the Library node and navigate to a page. The Details tab displays.
2. Click **Page Contents**. The Edit Contents tab displays.
3. Click **Add Contents**. The Add Books and Portlets to Placeholder dialog displays.

4. Display the pages that you want to choose from, using the Search area if needed.
5. Choose the portlets that you want to add by selecting the desired check boxes, and click **Add**.
6. When finished, click **Save**.

Desktop: To add a portlets to a page, perform these steps:

1. In the Portal Resource tree, expand the Portals node and navigate to a page. The Details tab displays.
2. Click **Page Contents**. The Edit Contents tab displays.
3. Click **Add Contents**; search for existing portlets if needed, then select the portlets that you want, and click **Add**. When finished, click **Save**.

Positioning Elements on a Page

The page layout is the grid structure of a page that holds placeholders for portlets and books on the page. You can select a layout for your portlets/books, and drag and drop them between the placeholders to customize the layout of each page.

Perform these steps:

1. In the Portal Resource tree, expand either the Library node or the Portals node as applicable, and select a page. The Details tab displays.
2. Click **Page Contents**. The Edit Contents tab displays.
3. If you want to change to a different layout, select a layout in the Layout drop-down menu.
4. Select the method that you want to use to position the elements on the page by selecting an option in the Position Elements area. The default is Drag & Drop.
5. Move portlets or books between placeholder columns.
6. If you want to prevent users from moving or deleting elements from a placeholder, select the Lock Placeholder check box.
7. When finished, click **Save Changes**.

Overview of Portlet Categories

Portlet categories provide for the classification of portlets, which is useful when organizing a large collection of portlets into meaningful groupings. The portlet categories are similar to other hierarchical structures in that parent “folders” can contain child folders and/or portlets. You must

first create a portlet category, and then you can manage portlets by adding them to a category or moving them between categories.

Creating a Portlet Category

To create a portlet category:

1. In the Portal Resources tree, expand the Library folder and select **Portlet Categories**. The Browse Category tab displays.
2. Click **Create New Category**.
3. Type a title and description for the new category in the pop-up window.
4. Click **Create**.

Modifying Portlet Category Properties

Portlet category properties include all of the features and elements that make up the category. As a portal administrator, you can modify some of these properties from the Summary tab. You can also edit the title, description, and locale information from the Titles & Descriptions tab, as described below.

Perform these steps:

1. In the Portal Resources tree, expand the Library node and navigate to a portlet category.
2. From the Summary tab, select the type of property that you want to change. Use the table below as a guide.

Table 8-3 Modifying Portlet Category Properties

Title and Description	
Change title and description of the category in the current locale	1. Click **Title & Description**. 2. Click the locale (for example, **en**) in the Locale cell; the Add a Localized Title & Description dialog displays. 3. Enter a new Title and/or Description. 4. Click **Update**.

Table 8-3 Modifying Portlet Category Properties (Continued)

Add a localized title for the category	1. Click **Title & Description**. 2. Click **Add Localized Title**; the Add a Localized Title & Description dialog appears. 3. Enter a Language and Country identifier, Variant if applicable, Title, and a Description for the localized title. 4. Click **Create**.
Portlets in Category	Refer to "Adding Portlets to a Portlet Category" on page 8-8.
Categories in Category	1. Click **Categories In Category**; the Browse Category tab displays. 2. Click **Create New Category**; the Create New Category dialog displays. 3. Enter a Title and Description for the new category. 4. Click **Create**. The category is created and added to the currently selected category.

Adding Portlets to a Portlet Category

To add portlets into a category:

1. Expand the Library node in the Portal Resources tree and navigate to a portlet category. The Summary tab displays.
2. Click **Portlets In Category**.
3. Click **Add Portlets**.
4. In the Available Portlets area, select the portlets that you want to add, and click **Add** to include them in the Selected Portlets area.
5. Click **Save**.

Overview of Portlet Preferences

A portlet preference is a property in a portlet that can be customized by either an administrator or a user. Your portlet might already have preferences, but if you have the appropriate Delegated Administration rights you can create additional portlet preferences.

Creating a Portlet Preference

To create a portlet preference, perform these steps:

1. Expand the Portals node or the Library node in the Portal Resources tree, as appropriate, and navigate to the portlet for which you want to create a preference. The Details tab displays.
2. Click **Add Portlet Preference**.
3. Fill in the information in the fields. Use the table below as a guide.

Table 8-4 Creating a Portlet Preference

For this field:	Enter this information:
Name	The name you want to give this preference.
Description	A description of this preference.
Value(s)	A value for a preference.
Is Modifiable? (checkbox)	Select this check box if you want to allow end users to modify this preference.
Is Multi-Valued? (checkbox)	Select this check box if you want to enter multiple values for the preference. If you select this box, an additional data entry field displays for you to enter additional values. Click Add Another Value after entering each value, until you are finished.

4. Click **Save**.

   For library instances of portlets, when you add a preference it automatically proliferates to library page instances and desktop page instances if the instances have not been decoupled.

5. If you want to force proliferation of this preference to every instance of this portlet, click **Propagate to Instances**; WebLogic Portal overwrites all desktop instance's preferences with the library preferences are. When complete, a message appears at the top of the Administration Console.

Here are some tips related to portlet preferences that you might find useful:

- When desktop instances of a portlet have no preferences, they automatically inherit the preferences from the library instance of the portlet.

- When desktop instances of a portlet have their own preferences set, they will not automatically inherit preferences from the library instance.
- If a desktop instance of a portlet has its own preferences set and these preferences are removed, it will automatically inherit all preferences from the library instance.
- If a desktop instance of a portlet has inherited preferences from the library instance and the desktop instance of this preference has been modified, it will no longer automatically inherit new preferences from the library or updates made to the library portlet's instance of this preference.
- If a desktop instance of a portlet has inherited the preferences from the library instance and no desktop instance specific preferences have been set, and the inherited preferences have not been modified in the desktop instance, the desktop instance will inherit all updates to the library preferences.

Editing a Portlet Preference

If you have the appropriate Delegated Administration rights, you can edit a portlet's preferences to change the way a portlet behaves.

To edit a portlet preference:

1. Expand the Portals node or the Library node in the Portal Resources tree, as appropriate, and navigate to the portlet for which you want to edit a preference. The Details tab displays.
2. Click **Portlet Preferences**.
3. Select the portlet preference by clicking its name in the Name column.
4. Edit the information in the fields. Use the table below as a guide.

Table 8-5 Editing a Portlet Preference

For this field:	Enter this information:
Name	The name you want to give this preference.
Description	A description of this preference.
Value(s)	A value for a preference.

Table 8-5 Editing a Portlet Preference (Continued)

For this field:	Enter this information:
Is Modifiable? (checkbox)	Select this check box if you want to allow end users to modify this preference.
Is Multi-Valued? (checkbox)	Select this check box if you want to enter multiple values for the preference. If you select this box, an additional data entry field displays for you to enter additional values. Click Add Another Value after entering each value, until you are finished.

5. Click **Save**.

   For library instances of portlets, when you edit a preference it automatically proliferates to library page instances and desktop page instances if the instances have not been decoupled.

6. If you want to force proliferation of this change to every instance of this portlet, click **Propagate to Instances**. When complete, a message appears at the top of the Administration Console.

Overview of Delegated Administration

In your organization, you typically want individuals to have different access privileges to various administration tasks and resources. For example, a system administrator might have access to every feature in the WebLogic Portal Administration Console. The system administrator might then create a portal administrator role that can manage instances of portal resources in specific desktop views of your portal, and a library administrator role that can manage your portal resource library. Other delegated administration roles only have access to resources if that access has been explicitly granted.

For more information about using delegated administration as a part of your security strategy, see the *Security Guide* on e-docs.

Overview of Visitor Entitlements

Visitor entitlements allow you to define who can access the resources in a portal application and what they can do with those resources. This access is based on the role assigned to a portal visitor, allowing for flexible management of the resources.

For more information about using visitor entitlements as a part of your security strategy, see the *Security Guide* on e-docs.

CHAPTER 9

Deploying Portlets

Deploying Portlets

Generally speaking, a WebLogic Portal application consists of an EAR file, an LDAP repository, and a database. The EAR file contains application code, such as JSPs and Java classes, and portal framework files that define portals, portlets, and datasync data. The embedded LDAP contains security-related data, such as entitlements, roles, users, and groups. The database contains representations of portal framework and datasync elements used by the portal runtime in streaming mode.

Portlet data can fall into the following two categories:

- **Portal Framework Data** – Refers to desktops, books, pages, and other portal framework elements that are created with the WebLogic Portal Administration Console.
- **EAR Data** – Refers to the final product of Workshop for WebLogic development—a J2EE EAR file. The EAR must be deployed to a destination server using the deployment feature of the WebLogic Server Administration Console.

When you deploy or redeploy a portal application EAR file to a server in production mode, .portlet files are automatically loaded into the database.

The primary tools you use to perform portlet deployment are the WebLogic Portal propagation tools and the deployment feature of the WebLogic Server Administration Console. For detailed instructions on deploying a portal and its portlets, refer to the *Productions Operations Guide*.

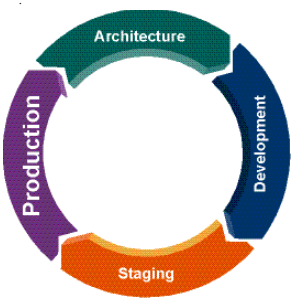
Part IV Production

Part IV includes the following chapter:

- Chapter 10, “Managing Portlets in Production”

A production portal is live and available to end users. A portal in production can be modified by administrators using the WebLogic Portal Administration Console and by users using Visitor Tools. For instance, an administrator might add additional portlets to a portal or reorganize the contents of a portal.

For a view of how the tasks in this section relate to the overall portal life cycle, refer to the *BEA WebLogic Portal Overview*.



CHAPTER 10

Managing Portlets in Production

During the life cycle of a WebLogic Portal application it moves back and forth between development, staging, and production environments. This chapter contains information about managing portlets that are on a production system.

This chapter contains the following sections:

Pushing Changes from the Library into Production

Proliferation is the process by which changes made to the Library instance of a portal asset are pushed into user-customized instances of that asset. For example, if a portal administrator deletes a portlet from a desktop, that change must be reflected into user-customized instances of that desktop.

The WebLogic Portal Administration Console includes a configuration setting for Proliferation under **Configuration Settings > Service Administration > Portal Resources**. The proliferation settings include **synch**, **asynch**, and **off**.

For more information on proliferation, refer to the *Production Operations Guide*.

Transferring Changes from Production Back to Development

WebLogic Portal utilities such as the propagation tools and the Export/Import Utility allow you to reliably move and merge changes between environments. The Export/Import Utility allows a full round-trip development life cycle, where you can easily move portals from a production environment back to your Workshop for WebLogic development environment.

For instructions on using the propagation tools and Export/Import Utility, refer to the *Production Operations Guide*.

APPENDIX A

Portlet Database Data

This appendix describes how portlet data is managed by databases, and contains the following sections:

- Database Structure for Portlet Data
- Portlet Resources in the Database

Database Structure for Portlet Data

When a portlet's data is loaded into the database, the portlet XML is parsed and a number of tables are populated with information about the portlet, including `PF_PORTLET_DEFINITION`, `PF_MARKUP_DEFINITION`, `PF_PORTLET_INSTANCE`, `PF_PORTLET_PREFERENCE`, `L10N_RESOURCE`, and `L10N_INTERSECTION`.

`PF_PORTLET_DEFINITION` is the master record for the portlet and contains columns for properties that are defined for the portlet, such as the definition label, the forkable setting, edit URI, help URI, and so on. The definition label and web application name are the unique identifying records for the portlet. Portlet definitions refer to the rest of the actual XML for the portlet that is stored in `PF_MARKUP_DEF`.

In the Development phase, you use Workshop for WebLogic to create portlets and place them onto a portal. In the Staging phase, you use the Administration Portal to add portlets to portal desktops. Each time you add a portlet to a desktop, you create an *instance* of that portlet. Portlet instances allow for multiple variations of the same portlet definition.

The following four types of portlet instances are recorded in the database for storing portlet properties:

- **Primary** – Properties defined in development and stored in the .portlet file.
- **Library** – Properties defined in the Portal Library, which may be changed using the WebLogic Administration Portal.
- **Admin** – A customized instance of the portlet in a desktop. This allows you to customize a portlet in a particular way for a desktop without affecting other instances of the portlet in other desktops.
- **User** – User-customized instances of the portlet defined in the Visitor Tools.

PF_PORTET_INSTANCE contains properties for the portlet for attributes such as DEFAULT_MINIMIZED, TITLE_BAR_ORIENTATION, and PORTLET_LABEL.

If a portlet has portlet preferences defined, those are stored in the PF_PORTLET_PREFERENCE table.

Finally, portlet titles can be internationalized. Those names are stored in the L10N_ RESOURCE table which is linked using L10N_INTERSECTION to PF_PORTLET_DEFINITION.

Removing Portlets from Production

If a portlet is removed from a newly deployed portal application and it has already been defined in the production database, it is marked as IS_PORTLET_FILE_DELETED in the PF_PORTLET_DEFINITION table. It displays as grayed out in the WebLogic Administration Portal, and user requests for the portlet, if it is still contained in a desktop instance, return a message indicating that the portlet is unavailable.

Portlet Resources in the Database

During the development phase, the `.portlet` files for portal web projects are stored as XML in the portal web application. As a developer creates new `.portlet` files, a file polling thread monitors changes and loads the development database with the `.portlet` information. When a portlet's data is loaded into the database, the portlet XML is parsed and a number of tables are populated with information about the portlet. Changes that you make using the WebLogic Portal Administration Portal are directly reflected in the database.

This section contains the following sections:

Types of Database Tables

Separate database tables store information about portlet resources, including the following:

- **Definitions** – Portlet definition properties including creation date, content URI, whether the portlet is forkable or cacheable, whether it has a backing file, and so on.
- **Instances** (including a subset of tables for WSRP) – Instance properties indicate whether the portlet is minimized by default, title bar orientation (top, left, right, bottom), the parent portlet instance if applicable, and so on.WSRP portlet properties include proxy portlet instance values.
- **Categories** – Portlet categories provide for the classification of portlets, which is useful when organizing a large collection of portlets into meaningful groupings. The database stores values for the category ID and creation/modification dates.
- **Category definitions** – The database stores values for the category ID and creation/modification dates, parent category, and so on.
- **Preferences** – Preference properties, such as whether or not the preference can be multi-valued or whether it is modifiable, are stored in this table.
- **Preference values** – The database stores the actual value of portlet preferences.
- **User properties** – The database table maintains values of portlet user properties for WSRP user profile propagation.

Tip: The tool you use to manipulate these resources varies according to the resource, and the phase of development you are in; for example, you can change portlet preferences using either Workshop for WebLogic or the WebLogic Portal Administration Portal, but you must use the Administration Portal to create portlet categories.

Management of Portlet Data

When a portlet is loaded into the database, the portlet XML is parsed and a number of tables are populated with information about the portlet, including `PF_PORTLET_DEFINITION`, `PF_MARKUP_DEFINITION`, `PF_PORTLET_INSTANCE`, `PF_PORTLET_PREFERENCE`, `L10N_RESOURCE`, and `L10N_INTERSECTION`.

PF_PORTLET_DEFINITION is the master record for the portlet and contains rows for properties that are defined for the portlet, such as the definition label, the forkable setting, edit URI, help URI, and so on. The definition label and web application name are the unique identifying records for the portlet. Portlet definitions refer to the rest of the actual XML for the portlet that is stored in PF_MARKUP_DEF.

PF_MARKUP_DEF contains stored tokenized XML for the .portlet file. This means that the .portlet XML is parsed into the database and properties are replaced with tokens. For example, the following code fragment shows a tokenized portlet:

```
<netuix:portlet $(definitionLabel) $(title) $(renderCacheable)
$(cacheExpires)>
```

These tokens are replaced by values from the master definition table in PF_PORTLET_DEFINITION, or by a customized instance of the portlet stored in PF_PORTLET_INSTANCE.

The following four types of portlet instances are recorded in the database for storing portlet properties:

- **Primary** – Properties defined in development and stored in the .portlet file.
- **Library** – Properties defined in the Portal Library, which you can change using the WebLogic Portal Administration Portal.
- **Admin** – A customized instance of the portlet in a desktop. This allows you to customize a portlet in a particular way for a desktop without affecting other instances of the portlet in other desktops.
- **User** – User-customized instances of the portlet defined in the Visitor Tools.

PF_PORTET_INSTANCE contains properties for the portlet for attributes such as DEFAULT_MINIMIZED, TITLE_BAR_ORIENTATION, and PORTLET_LABEL.

If a portlet has portlet preferences defined, those are stored in the PF_PORTLET_PREFERENCE table.

Finally, portlet titles can be internationalized. Those names are stored in the L10N_ RESOURCE table which is linked using L10N_INTERSECTION and PF_PORTLET_DEFINITION.

How the Database Shows Removed Portlets

If a portlet is removed from a deployed portal project, and it has already been defined in the production database, the portlet is marked as IS_PORTLET_FILE_DELETED in the

PF_PORTLET_DEFINITION table. The portlet displays as grayed out in the Administration Portal, and user requests for the portlet (if it is still contained in a desktop instance) return a message indicating that the portlet is unavailable.

For detailed information about the content of WebLogic Portal database tables, refer to the *Database Administration Guide*.