

Oracle® WebLogic Portal

Portlet Development Guide

10g Release 3 (10.3)

February 2011

ORACLE®

Copyright © 2007, 2011, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

1. Introduction

Portlet Overview	1-1
Portlet Development and the Portal Life Cycle	1-2
Architecture	1-3
Development	1-3
Staging	1-4
Production	1-4
Getting Started	1-5
Prerequisites	1-5
Related Guides	1-5

Part I. Architecture

2. Portlet Planning

Portlet Development in a Distributed Portal Team	2-2
Portlets in a Non-Portal Environment	2-2
Planning Portlet Instances	2-2
Security	2-3
Interportlet Communication	2-3
Performance Planning	2-4

3. Portlet Types

Java Server Page (JSP) and HTML Portlets	3-2
--	-----

Java Portlets (JSR 168)	3-2
Java Page Flow Portlets	3-2
Struts Portlets	3-3
Java Server Faces (JSF) Portlets	3-3
Browser (URL) Portlets	3-4
Clipper Portlets	3-5
Remote Portlets	3-5
Portlet Type Summary Table	3-5

Part II. Development

4. Understanding Portlet Development

Portlet Components	4-1
Portlet Properties	4-2
Portlet Title Bar, Mode, and State	4-3
Portlet Preferences	4-3
Resources for Creating Portlets	4-4
Portlet Rendering	4-4
Render and Pre-Render Forking	4-4
Asynchronous Portlet Content Rendering	4-5
Portlets as Popups (Detached Portlets)	4-5
JSP Tags and Controls in Portlets	4-6
Backing Files	4-6

5. Building Portlets

Supported Portlet Types	5-2
Portlets in J2EE Shared Libraries	5-3
Portlet Wizard Reference	5-4
Order of Creation - Resource or Portlet First	5-4

Starting the Portlet Wizard	5-7
New Portlet Dialog	5-9
Select Portlet Type Dialog	5-9
Portlet Details Dialogs	5-11
How to Build Each Type of Portlet	5-11
JSP and HTML Portlets	5-11
Java Portlets	5-13
Java Page Flow Portlets	5-19
JSF Portlets	5-22
Browser Portlets	5-28
Clipper Portlets	5-31
Struts Portlets	5-31
Remote Portlets	5-34
Web Service Portlets	5-35
Detached Portlets	5-35
Considerations for Using Detached Portlets	5-36
Building Detached Portlets	5-37
Working with Inlined Portlets	5-37
Extracting Inlined Portlets	5-38
Setting the Theme of an Inlined Portlet	5-39
Extracting Books and Pages	5-39
Portlet Properties	5-40
Editing Portlet Properties	5-40
Tips for Using the Properties View	5-42
Portlet Properties in the Portal Properties View	5-43
Portlet Properties in the Portlet Properties View	5-44
Portlet Preferences	5-56
Specifying Portlet Preferences	5-57

Using the Preferences API to Access or Modify Preferences	5-62
Portlet Preferences SPI.	5-67
Best Practices in Using Portlet Preferences.	5-70
Backing Files.	5-71
How Backing Files are Executed.	5-72
Thread Safety and Backing Files	5-74
Scoping and Backing Files.	5-74
Backing File Guidelines.	5-74
Portlet Appearance and Features.	5-76
Portlet Dependencies	5-77
Portlet Modes	5-85
Creating Custom Modes.	5-87
Portlet States.	5-94
Portlet Title Bar Icons	5-96
Portlet Height and Scrolling.	5-96
Getting Request Data in Page Flow Portlets	5-99
JSP Tags and Controls in Portlets	5-100
Viewing Available JSP Tags	5-100
Viewing Available Controls.	5-102
Portlet State Persistence	5-103
Adding a Portlet to a Portal	5-104
Deleting Portlets	5-106
Advanced Portlet Development with Tag Libraries	5-106
Adding ActiveMenus.	5-106
Enabling Drag and Drop	5-118
Enabling Dynamic Content	5-121
Using the User Picker	5-124
Importing and Exporting Java Portlets	5-125

Importing Java Portlets	5-125
Exporting Java Portlets	5-128
Using the JSR168 Import Utility	5-131

6. Creating Clipper Portlets

Introduction	6-2
Creating a Clipper Portlet	6-2
Modifying Clipper Portlet Properties	6-4
Using the Properties Editor	6-5
Setting Clipper Properties Manually as Preferences	6-5
Modifying the Appearance of a Clipper Portlet	6-6
Authenticating a Clipper Portlet	6-8
Form-Based Authentication	6-8
Basic HTTP Authentication	6-10
Configuring URL Rewriting	6-11
Navigable Link Configurations	6-11
Resource URL Configurations	6-11
URL Rewriting Configuration Techniques	6-12
Clipper Portlets and HTTPS	6-14
Certificates and WebLogic Server.	6-14
Resetting the Clipper Portlet	6-15
Using Backing Files with Clipper Portlets	6-15
Updating Portlet Preferences While the Server is Running.	6-16
Clipper Portlet Limitations	6-16

7. Optimizing Portlet Performance

Performance-Related Portlet Properties	7-1
Portlet Caching	7-2

Remote Portlets	7-2
Portlet Forking	7-3
Configuring Portlets for Forking	7-3
Architectural Details of Forked Portlets	7-6
Best Practices for Developing Forked Portlets	7-10
Asynchronous Portlet Content Rendering	7-13
Implementing Asynchronous Portlet Content Rendering	7-14
Thread Safety and Asynchronous Rendering	7-16
Considerations for IFRAME-based Asynchronous Rendering	7-16
Considerations for AJAX-based Asynchronous Rendering	7-17
Comparison of IFRAME- and AJAX-based Asynchronous Rendering	7-17
Comparison of Asynchronous and Conventional or Forked Rendering	7-18
Portal Life Cycle Considerations with Asynchronous Content Rendering	7-19
Asynchronous Content Rendering and IPC	7-20

8. Monitoring and Determining Portlet Performance

Introduction	8-1
Use Case	8-2
Detecting a Misbehaving Portlet	8-2
Disabling the Bad Portlet and Enabling an Alternative Portlet	8-4

9. Local Interportlet Communication

Definition Labels and Interportlet Communication	9-2
Portlet Events	9-2
Event Handlers	9-2
Event Types	9-4
Event Actions	9-5
Portlet Event Handlers Wizard Reference	9-6

JSF Events	9-10
IPC Example	9-12
Before You Begin - Environment Setup	9-12
Basic IPC Example	9-14
IPC Special Considerations and Limitations	9-28
Using Asynchronous Portlet Rendering with IPC	9-28
Generic Event Handler for WSRP	9-29
Consistency of the Listen To Field	9-29

10. Adding the Content Presenter Portlet

Using the Content Presenter Example	10-1
Starting the Content Presenter Example	10-2
Performing Inline Editing in the Content Presenter Example	10-2
Enabling Inline Editing in Your Portlets	10-5
Configuring the Content Presenter Portlet in Your Portal	10-6
Configuring the Content Presenter Portlet	10-7

11. Adding a Third-Party Portlet

Using the Collaboration Portlets	11-1
What Are Collaboration Portlets?	11-2
Adding Collaboration Portlets To Your Portal	11-2
Configuring Collaboration Portlets for a Shared View	11-7
Using the Collaboration Portlets	11-8
Using the Collaboration Portlet Source Code	11-8
Third-Party Portlets	11-9
Autonomy Portlets	11-9
Documentum Portlets	11-10
MobileAware Portlets	11-10

12. Working With JSF Portlets

Overview	12-2
Configuring JSF Within Weblogic Portal	12-2
JSF Library Modules in WebLogic Server.	12-3
Installing the JSF Libraries into a Portal Web Project	12-3
Configuring JSF 1.2 in WLP	12-6
Creating JSF Portlets	12-7
JSF Configuration Settings	12-7
Native Bridge Architecture	12-10
Container Architecture Overview	12-10
Container Interactions	12-12
Understanding WLP and JSF Rendering Life Cycles.	12-12
WLP and JSF Life Cycles	12-12
Invocation Order of WLP and JSF Life Cycle Methods	12-12
Accessing WLP Context Objects from JSF Managed Beans	12-13
Understanding Scopes and JSF Portlets	12-14
Conceptual Scopes for Standard JSF Applications	12-15
Conceptual Scopes for Portal Applications	12-15
Implementation Patterns for Portal Scopes	12-16
State Sharing Patterns	12-18
State Sharing Concepts	12-18
HttpSession Versus HttpServletRequest	12-19
Base Code for HttpSession Patterns.	12-20
Single Portlet Pattern	12-21
Multiple Portlet Patterns	12-22
Using Common WLP Features With JSF Portlets	12-29
Portlet Container Features	12-30

Portal Container Features and JSF Portlets	12-35
Understanding Navigation	12-36
Navigating Within a Portlet with the JSF Controller	12-36
Support for Redirects	12-37
Navigation Within a Portal Environment	12-40
Programmatically Constructing JSF Portlet URLs.	12-40
Changing the Active Portal Page	12-41
Using an Output Link.	12-41
Using a Command Link or Button With Events.	12-42
Changing the Active Portal Page Using the Navigation Controller and a Portal Event.	12-42
Changing the Active Portal Page Programmatically	12-43
Interportlet Communication with JSF Portlets	12-44
Using Session and Request Attributes for IPC (Anti-pattern)	12-45
Using the WLP Event Facility for IPC with JSF Portlets	12-45
Notifications	12-50
Comparison of the IPC Approaches	12-50
Namespacing	12-51
Namespacing Managed Bean Names.	12-52
Client ID Namespacing with the View and Subview Components	12-52
Client ID Namespacing with the WLP NamingContainer	12-53
Using Custom JavaScript in JSF Portlets	12-56
DOM Manipulation within a JSF Portlet	12-56
Form Validation within a JSF Portlet	12-60
Ajax Enablement	12-61
Ajax in JSF Portlets	12-61
Partial Page Rendering Pattern.	12-61
Stateless API Request Pattern	12-62

Portlet Aware API Request Pattern	12-63
Controlling the WLP Ajax Framework	12-69
Localizing JSF Portlets	12-71
Configuring the JSF Locale	12-71
Resource Bundles.	12-71
Listing Locales in faces-config.xml.	12-72
Ensuring Parity in Configured WLP and JSF Locales.	12-73
Modularizing Resource Bundles	12-73
Preparing JSF Portlets for Production.	12-73
Configuration Tasks	12-74
Performance and Scalability	12-75
Securing JSF Portlets	12-77
Tips for Logging, Iterative Development, and Debugging of JSF Portlets.	12-78
Enabling Logging.	12-78
Using Iterative Development for JSF Portlets	12-79
Debugging	12-80
Consolidated List of Best Practices.	12-81
Configuration	12-81
Namespacing	12-81
Logging, Iterative Development, Debugging	12-82
Custom JavaScript	12-82
Preparing JSF Portlets for Production	12-82
Interportlet Communication.	12-82
Scopes	12-83
State Sharing Patterns	12-83
Rendering Lifecycles	12-83
Ajax Enablement	12-83
Login Portlet	12-83

Part III. Staging

13. Assembling Portlets into Desktops

Portlet Library	13-1
Managing Portlets Using the Administration Console	13-2
Copying a Portlet in the Library	13-3
Modifying Library Portlet Properties	13-3
Modifying Desktop Portlet Properties	13-4
Deleting a Portlet	13-5
Managing Portlets on Pages	13-5
Overview of Portlet Categories	13-6
Overview of Portlet Preferences	13-8
Creating a Portlet Preference	13-9
Editing a Portlet Preference	13-10
Overview of Delegated Administration	13-11
Overview of Visitor Entitlements	13-11

14. Deploying Portlets

Deploying Portlets	14-1
--------------------------	------

Part IV. Production

15. Managing Portlets in Production

Pushing Changes from the Library into Production	15-1
Transferring Changes from Production Back to Development	15-2

Part V. Appendixes

A. Portlet Database Data

Database Structure for Portlet Data	A-1
---	-----

Removing Portlets from Production	A-2
Portlet Resources in the Database	A-2
Types of Database Tables	A-3
Management of Portlet Data	A-3
How the Database Shows Removed Portlets	A-4

B. JSF Portlet Development

Code Examples	B-1
The JSFPortletHelper Class	B-1
Login Portlet Example	B-16
Using Facelets	B-29
Introduction to Facelets	B-30
Configuring Facelets Support	B-30
Using Tomahawk	B-32
What is Apache MyFaces Tomahawk?	B-32
Support for Tomahawk in WLP	B-33
Installing and Configuring Tomahawk	B-34
Resolving the Duplicate ID Issue	B-35
Referring to Resources	B-39
forceId Attribute	B-45
File Upload	B-46
Integrating Apache Beehive Pageflow Controller	B-46
Apache Beehive Page Flow	B-46
JSF and Page Flows	B-46
Configuring the JSF Integration with Page Flows	B-48
Building Unsupported JSF Implementations	B-48

Introduction

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

This chapter includes the following sections:

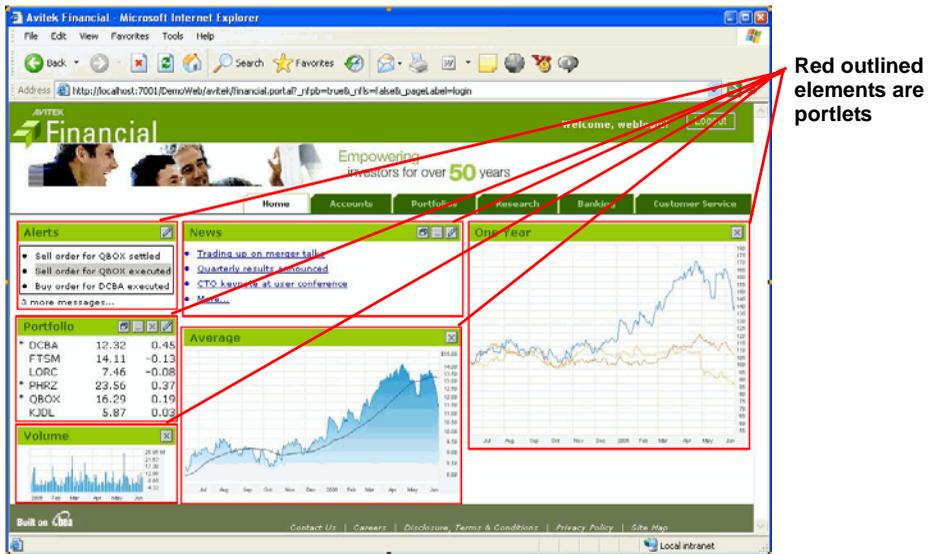
- [Portlet Overview](#)
- [Portlet Development and the Portal Life Cycle](#)

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



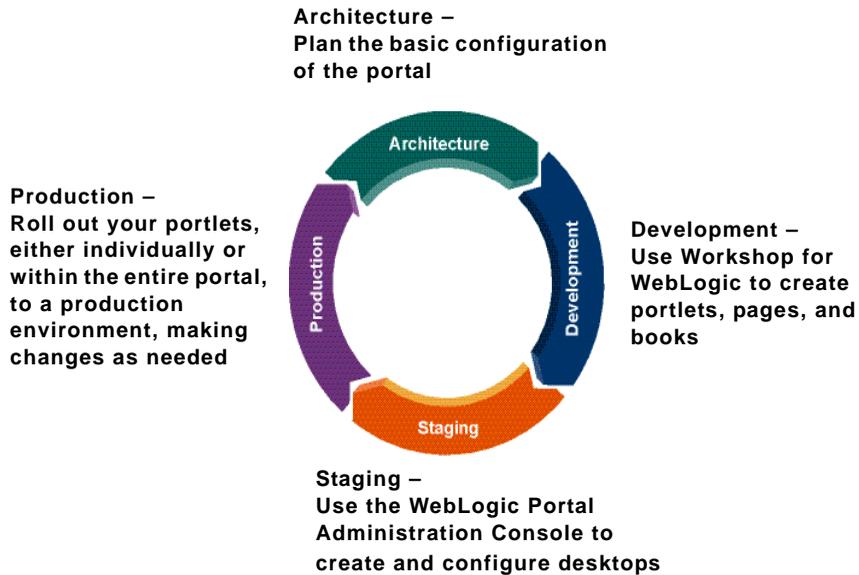
WebLogic Portal supports the development of portlets through Workshop for WebLogic (Workshop for WebLogic), which is a client-based tool. You can develop portals without Workshop for WebLogic through coding in any tool of choice such as JBuilder, VI or Emacs; portlets can be written in Java or JSP, and 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 *Oracle 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, “Creating Clipper Portlets”](#)
- [Chapter 7, “Optimizing Portlet Performance”](#)
- [Chapter 8, “Monitoring and Determining Portlet Performance”](#)
- [Chapter 9, “Local Interportlet Communication”](#)
- [Chapter 10, “Adding the Content Presenter Portlet”](#)
- [Chapter 11, “Adding a Third-Party Portlet”](#)
- [Chapter 12, “Working With JSF Portlets”](#)

Staging

Oracle 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 13, “Assembling Portlets into Desktops”](#)
- [Chapter 14, “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 15, “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

Oracle recommends that you review the following guides:

- [Oracle WebLogic Portal Overview](#)
- [Oracle WebLogic Portal Development Guide](#)

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

Introduction

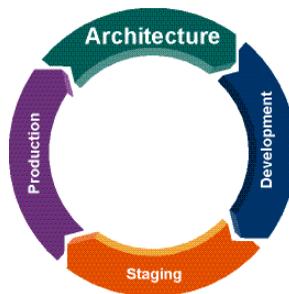
Part I Architecture

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](#).



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 *Oracle 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 *Oracle 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](#)
- [Portlets in a Non-Portal Environment](#)
- [Planning Portlet Instances](#)
- [Security](#)
- [Interportlet Communication](#)
- [Performance Planning](#)

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 Oracle 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 ORCL, which would configure the portlet to display Oracle 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 13-1](#).

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 9, “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 7, “Optimizing Portlet Performance.”](#) For post-development WebLogic Portal performance recommendations, refer to the *Performance Tuning Guide*.

Portlet Types

As part of your portlet implementation plan, Oracle 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](#)
- [Java Portlets \(JSR 168\)](#)
- [Java Page Flow Portlets](#)
- [Java Server Faces \(JSF\) Portlets](#)
- [Browser \(URL\) Portlets](#)
- [Clipper Portlets](#)
- [Struts Portlets](#)

- [Remote Portlets](#)
- [Portlet Type Summary Table](#)

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 5-13](#).

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 5-19](#).

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, Oracle 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 5-31](#).

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 5-22](#).

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.

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 5-28](#).

Clipper Portlets

Clipping is an easy technique for including content in your portal. You can clip all or part of another web site. Users can effectively view and interact with content from another web site without leaving the portal. For detailed information on creating clipper portlets, see [Chapter 6, “Creating Clipper Portlets.”](#)

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

Table 3-1 Portlet Type Summary Table (Continued)

Type	Advantages	Disadvantages
Browser	<p>Allows a portlet to display content from a URL that is outside the portal project.</p> <p>Provides a “no development needed” portlet for quick implementation.</p>	<p>Less control over formatting.</p> <p>Lacks certain features of other portlet types, such as Content Path and Error Path.</p> <p>No interportlet communication support.</p>
Clipper	<p>Lets you subset or modify the contents of a remote web page. The portal can potentially access the clipper portlet’s content.</p>	<p>Clipped content is included directly in the portal page, allowing the potential for overlapping with other parts of the portal.</p>
Struts	<p>Provides a flexible control layer based on standard technologies like Java Servlets, JavaBeans, ResourceBundle, and XML.</p> <p>Provides a more structured approach for creating and maintaining complex applications.</p> <p>Useful for importing existing applications.</p>	<p>Not quite as robust as page flow portlets, which are based on Beehive. For new development, page flow portlets provide a better solution.</p>
Remote	<p>Allows you to functionally and operationally de-couple applications within your portal.</p> <p>Allows you to leverage external sources for portlet content.</p> <p>Depending on the environment, might improve performance.</p>	<p>Implementation is more complex.</p> <p>Your application’s features might not be able to be as robust; for example, some Javascript might not perform correctly.</p> <p>Depending on the environment, might have a performance cost. For more about performance with remote portlets, refer to “Remote Portlets” on page 7-2.</p>

Portlet Types

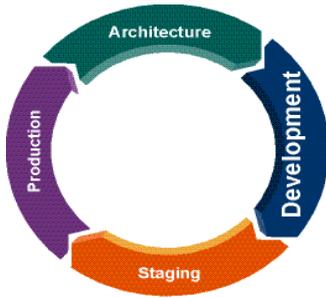
Part II Development

Part II includes the following chapters:

- [Chapter 4, “Understanding Portlet Development”](#)
- [Chapter 5, “Building Portlets”](#)
- [Chapter 6, “Creating Clipper Portlets”](#)
- [Chapter 7, “Optimizing Portlet Performance”](#)
- [Chapter 8, “Monitoring and Determining Portlet Performance”](#)
- [Chapter 9, “Local Interportlet Communication”](#)
- [Chapter 10, “Adding the Content Presenter Portlet”](#)
- [Chapter 11, “Adding a Third-Party Portlet”](#)
- [Chapter 12, “Working With JSF Portlets”](#)

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](#).



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 “How to Build Each Type of Portlet” on page 5-11.

This chapter contains the following sections:

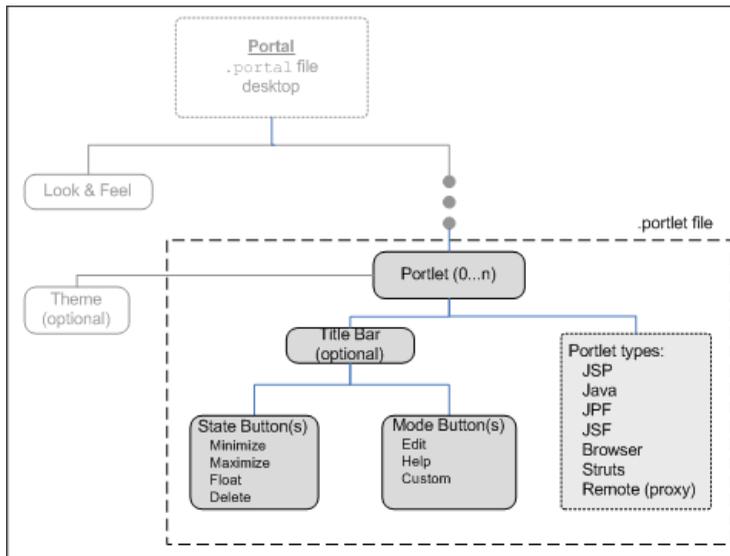
- [Portlet Components](#)
- [Resources for Creating Portlets](#)
- [Portlet Rendering](#)
- [JSP Tags and Controls in Portlets](#)
- [Backing Files](#)

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 Title Bar, Mode, and State](#)
- [Portlet Preferences](#)
- [Render and Pre-Render Forking](#)
- [Asynchronous Portlet Content Rendering](#)
- [Portlets as Popups \(Detached Portlets\)](#)

For more information about Look & Feel components, refer to the [Portal Development Guide](#).

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 5-40](#)

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 5-76](#).

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 5-56](#)

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 J2EE Shared Libraries** – Portlets are 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. For more information on the Collaboration portlets, including installation instructions, see [“Using the Collaboration Portlets” on page 11-1](#).
- **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 7-3](#).

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 7-13](#).

Tip: You can also enable asynchronous rendering for an entire portal desktop by setting a portal property in either Workshop for WebLogic or the WebLogic Portal Administration Console. For more information on asynchronous desktop rendering, see the [WebLogic Portal Development Guide](#).

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 [“Building Detached Portlets” on page 5-37](#).

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 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.

JSP tag libraries appear in the Design Palette whenever the JSP editor is open. If you do not see this palette, select **Window > Show View > Design Palette**. Select Tag Libraries from the palette's drop down menu to show only the tag libraries.

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 5-100](#).

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 5-71](#).

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:

- [Supported Portlet Types](#)
- [Portlets in J2EE Shared Libraries](#)
- [Portlet Wizard Reference](#)
- [How to Build Each Type of Portlet](#)
- [Detached Portlets](#)
- [Working with Inlined Portlets](#)
- [Extracting Books and Pages](#)
- [Portlet Properties](#)
- [Portlet Preferences](#)
- [Backing Files](#)
- [Portlet Appearance and Features](#)

- [Getting Request Data in Page Flow Portlets](#)
- [JSP Tags and Controls in Portlets](#)
- [Portlet State Persistence](#)
- [Adding a Portlet to a Portal](#)
- [Deleting Portlets](#)
- [Advanced Portlet Development with Tag Libraries](#)
- [Importing and Exporting Java Portlets](#)

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.
- **Clipper Portlets** – A clipper portlet is a portlet that renders content from another web site. A clipper portlet can include all or a subset of another web site's content using a process called "web clipping." Clipper portlets are discussed in [Chapter 6, "Creating Clipper Portlets."](#)
- **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."](#)

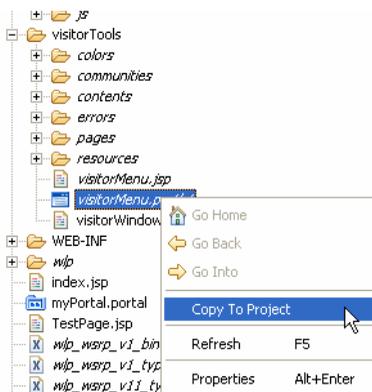
Portlets in J2EE Shared Libraries

You can copy portlets or other resources from a J2EE Shared Library 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 J2EE Shared Library. To see a list of available portlets, you can use the Merged Projects View of the workbench; resources contained in J2EE Shared Libraries 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 J2EE Libraries and their locations on your file system, you can select **Window > Preferences > WebLogic > J2EE Libraries**.

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 J2EE Shared Library portlet in the Merged Projects view with the Copy to Project option selected.

Caution: If you copy J2EE Shared 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 J2EE Shared Libraries, 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

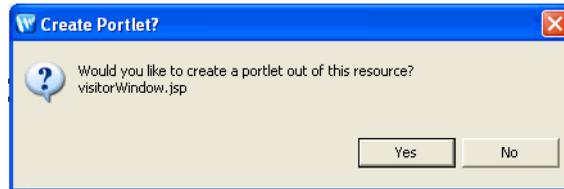
You might already have a JSP file, for example, that you want to use as the basis for a portlet. (In addition to JSP files, you can drag other resources onto the portal (such as content selectors) to automatically start the portlet wizard.)

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.

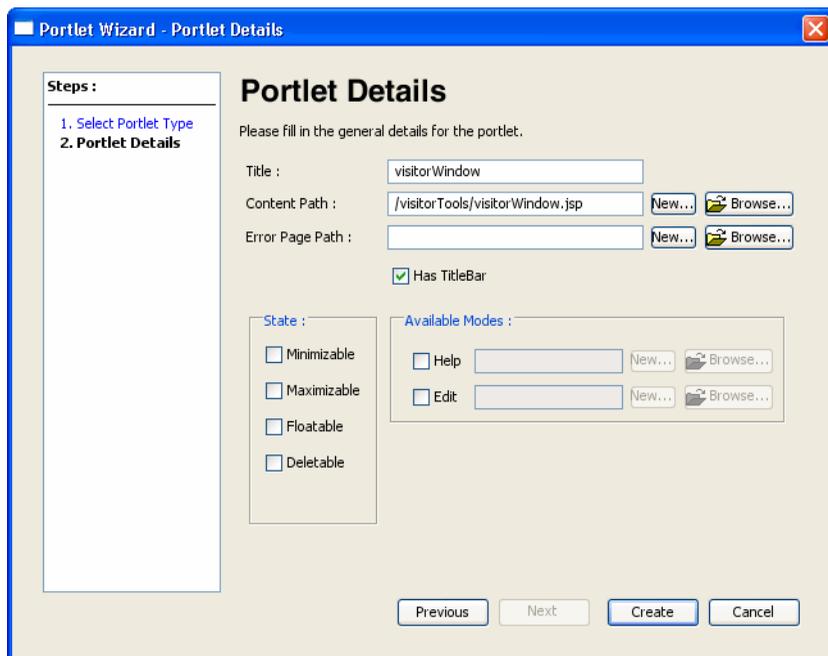
Workshop for WebLogic prompts you with a dialog similar to the example in [Figure 5-2](#).

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:

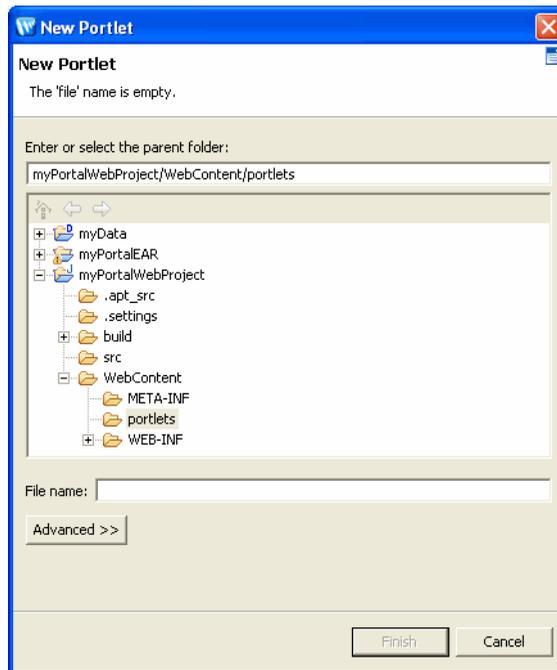
Figure 5-3 Example Portlet Wizard Details Dialog Following Drag and Drop of a Resource



Create the Portlet First

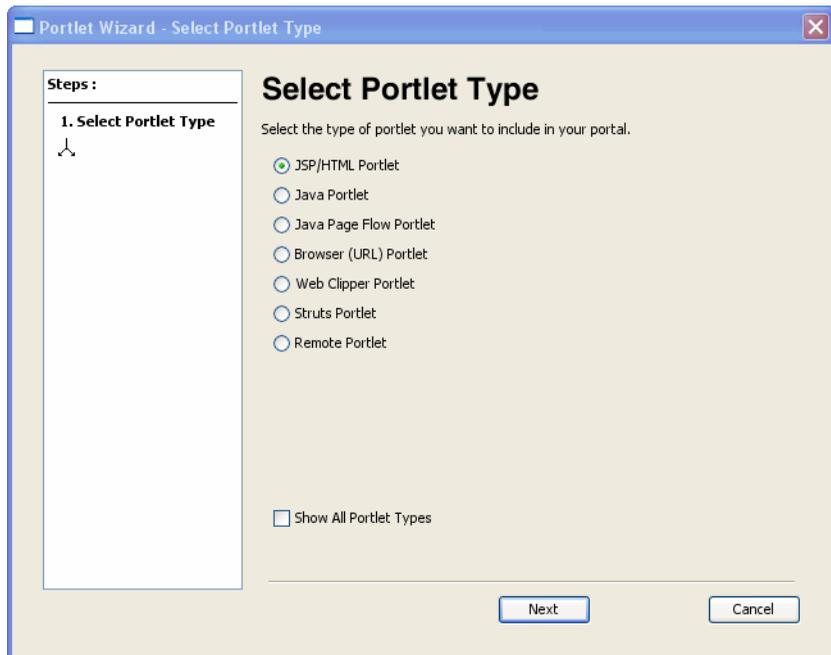
If you do not have an existing source file to start with, you can create the portlet using the New Portlet dialog and the Portlet Wizard. To do so, right-click a folder in your portal web project and select **New > Portlet**. [Figure 5-4](#) shows an example of the New Portlet dialog.

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. Detailed instructions for creating each type of portlet are contained in “[How to Build Each Type of Portlet](#)” on page 5-11.

Figure 5-5 Portlet Wizard - Select Portlet Type Dialog

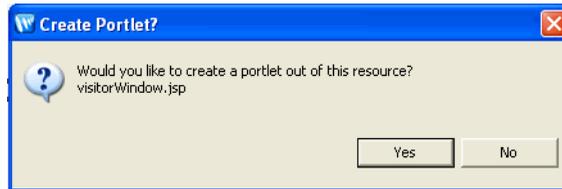


Starting the Portlet Wizard

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](#).

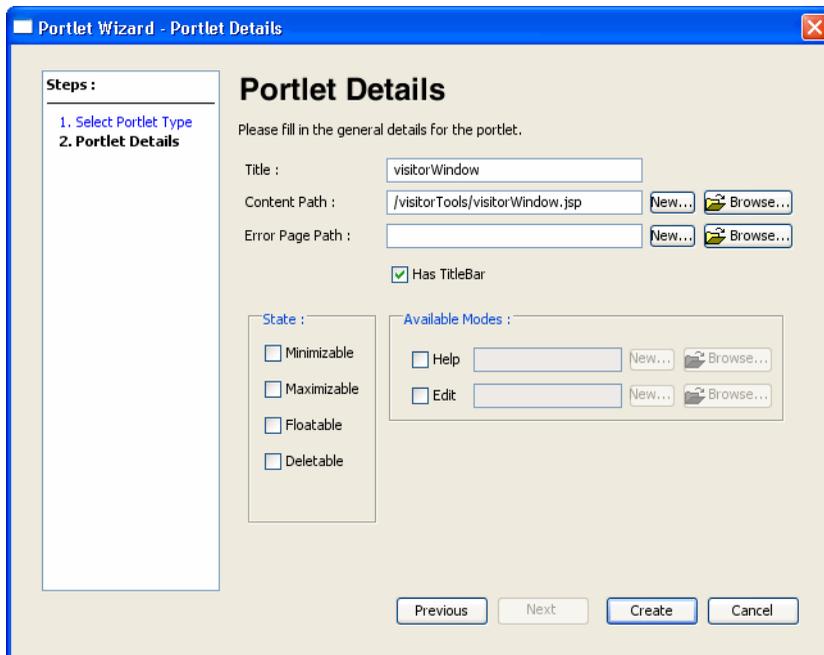
Figure 5-6 Portlet Wizard Prompt Following Drag and Drop of a Resource



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 and later versions, the option to convert a non-portal project to a portal project is not offered. For information on how to integrate portal J2EE Shared Libraries into an already existing project, see the [Portal Development Guide](#).

Select Portlet Type Dialog

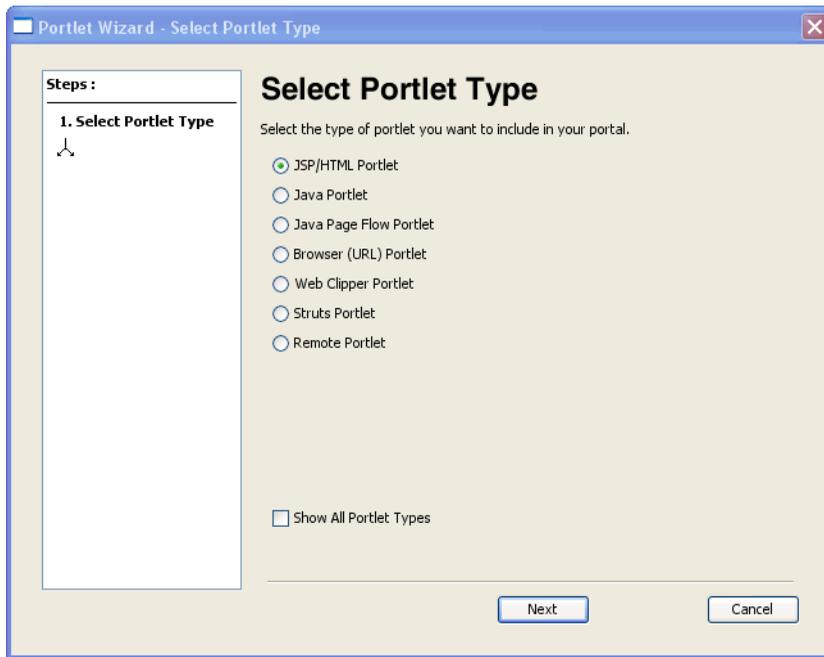
When the Portlet Wizard starts, it determines the valid portlet types to offer on the Select Portlet Type dialog, based on the type of project that you are working in.

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



Tip: The Java Server Faces (JSF) Portlet selection only appears by default if you have added the JSF Project facet to your portal web project. In some cases, you may wish to manually install the modules that are required to create JSF portlets. Although this method is not recommended, if you manually install the appropriate modules, you can force the JSF portlet option to appear in the dialog by selecting **Show All Portlet Types**. For more information on JSF portlets, see [“JSF Portlets” on page 5-22](#). .

The **Show All Portlet Types** option forces all portlet types to appear in the Select Portlet Type dialog even if their modules were not installed. For example, the Java Server Faces (JSF) Portlet selection only appears by default if you have added the JSF Project facet to your portal web project. In some cases, you may wish to manually install the modules that are required to create JSF portlets. Although this method is not recommended, if you manually install the appropriate modules, you can force the JSF portlet option to appear in the dialog by selecting **Show All Portlet Types**. For more information on JSF portlets, see [“JSF Portlets” on page 5-22](#).

WARNING: If you create and publish portlets that require modules that have not been properly installed, unexpected behavior and server runtime errors can occur.

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 5-11](#) 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](#)
- [Clipper 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.

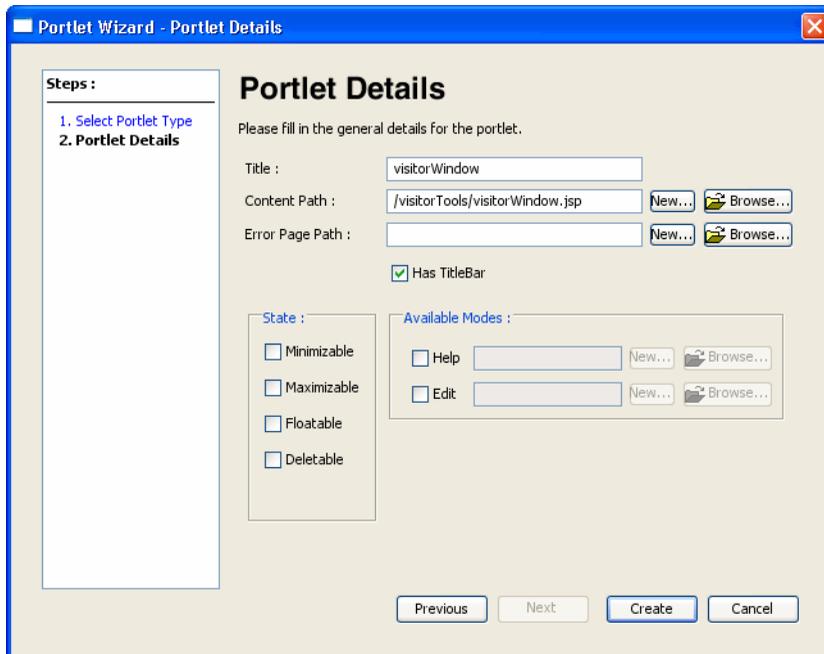
There are several ways to invoke the Portlet Wizard, as explained in the section [“Starting the Portlet Wizard” on page 5-7](#). This description assumes that you create a portlet based on an existing JSP file.

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 Path	The value for the Content URI (location of the JSP) is probably already filled in. You can change this value if you wish either by entering the path to a JSP or browsing to it. You can also create a new JSP on the fly either by entering a name in the field or by choosing the New button.

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

Field	Description
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. You can also create a new JSP on the fly either by entering a name in the field or by choosing the New button.
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 5-94 .
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 5-85 .

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.

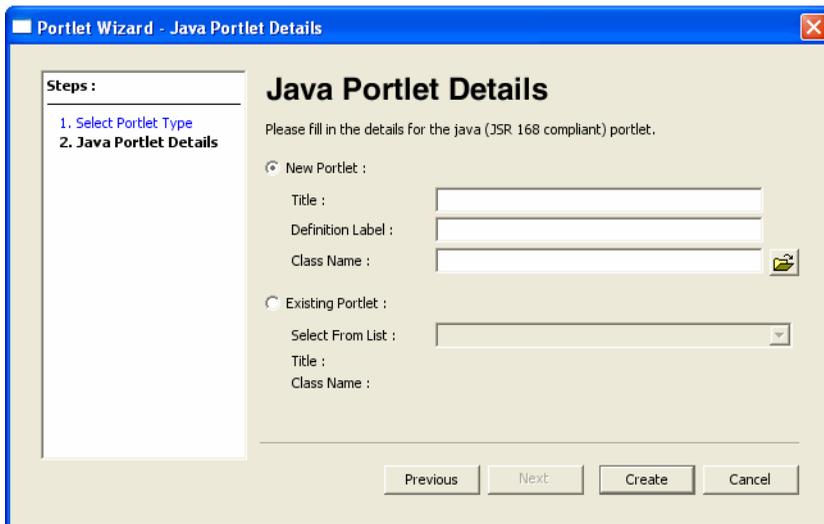
WebLogic Portal provides capabilities for Java portlets beyond those listed in the JSR168 spec. For example, you can set threading options, use a backing file, and so on. To implement these additional features, WebLogic Portal uses a combination of the typical `.portlet` file—which you create in the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

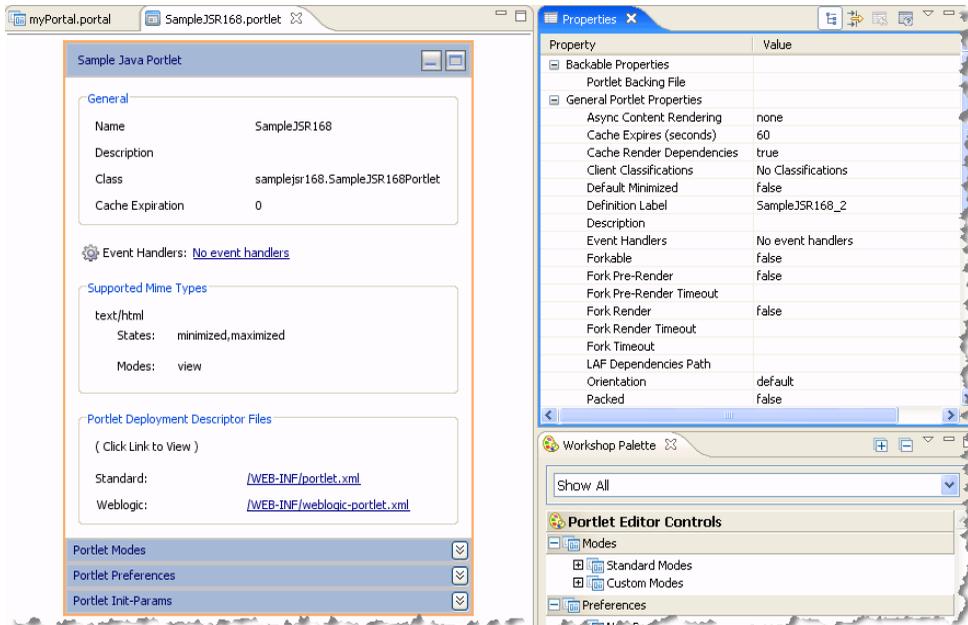
Field	Description
New Portlet – Title	The value for the Title maps to the <code><title></code> element in the file <code>portlet.xml</code> . The title in the <code>.portlet</code> file takes priority over the one in the <code>portlet.xml</code> file.
New Portlet – Definition Label	This value acts as the definition label for any portlet; more importantly, the value maps to the <code><portlet-name></code> element in the <code>portlet.xml</code> 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 <code><portlet-class></code> element. If you enter a <code>javax.portlet.Portlet</code> class that does not currently exist, the wizard will create the corresponding <code>.java</code> file when you click Create.
Existing Portlet – Select From List	The dropdown menu is populated from the <code>portlet.xml</code> file and contains the values from the <code><portlet-name></code> 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 <code>web.xml</code> file for the WebLogic Portal implementation of the JSR-168 portlet taglib; this taglib is declared implicitly. Be sure to use <code>http://java.sun.com/portlet</code> 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 Oracle-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>

```

Importing and Exporting 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. Workshop for WebLogic lets you export Java portlets to a supported archive file (WAR, JAR, or ZIP) that can be deployed on any supported server. You can also use the Import feature to import archive files containing Java portlets into your Workshop for WebLogic workspace. For details, see [“Importing and Exporting Java Portlets” on page 5-125](#).

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

Portlet Init-Params

The `init-param` element contains a name/value pair as an initialization parameter of the portlet. You can use the `getInitParameterNames` and `getInitParameter` methods of the `javax.portlet.PortletConfig` interface to return these initialization parameter names and values in your portlet code. Init-params are described in the JSR168 specification.

You can add init-params to your Java portlet by dragging a **New Init-Param** icon from the Design Palette onto the Java portlet in the portlet editor. Then, click on the init-param section of the portlet to display the parameter's properties in the Property view. In the Property view, you can enter the following init-param data:

- Description
- Name
- Value

For example, if you created an init-param called "Color" and set the default value to "green," the following entry will be made in the `portlet.xml` file:

```
<init-param>
  <description>My init param</description>
  <name>Color</name>
  <value>green</value>
</init-param>
```

Java Page Flow Portlets

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

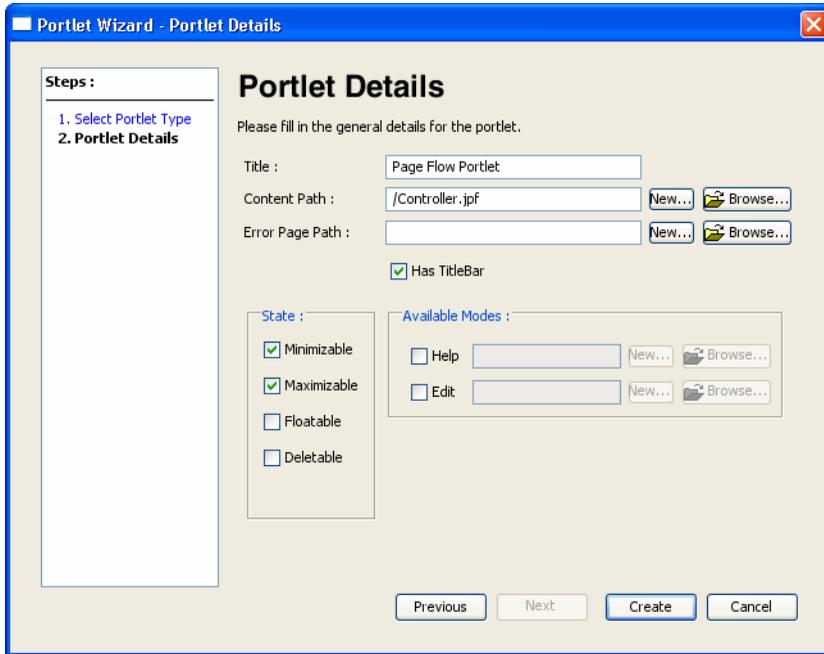
To create a page flow portlet, follow these steps:

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	<p>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.</p> <p>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 <code>.portlet</code> file when the portlet is created.</p> <p>If you enter or navigate to a <code>.java</code> that has no corresponding class in the project or J2EE Shared Libraries, the Portlet Wizard creates the <code>.java</code> file for the page flow. If multiple project source directories exist, then the wizard prompts you to store the new <code>.java</code> file in the source directory of your choice.</p>
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 5-94 .
Available Modes	<p>You can enable access to Help from the portlet or you can allow the user to edit the portlet.</p> <p>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 5-85.</p>

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 5-35](#).

JSF Portlets

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

Note: The Java Server Faces (JSF) Portlet selection only appears in the Portlet Wizard if you have added the JSF Project Facet to the portal web project. To add the JSF Project Facet, right-click the portal web project in the Package Explorer. In the Properties dialog, select **Project Facets** in the tree on the left. Click **Add/Remove Project Facets**, select **JSF** in the Project Facets dialog, and click **Finish**.

A JSF portlet will ask for a content path. This is usually a JSF enabled JSP. So, before we can create a JSF Portlet, we begin by creating a new JSF JSP.

To create a simple JSF view:

1. Right-click the WebContent folder of the Portal Web Project, and navigate to **New > JSP**.
2. Enter a file name, click Next.

If you select **New JSF JSP**, this will bring in the appropriate JSF taglibs. However, it also brings in head and body tags that you will need to remove to work within a portlet.

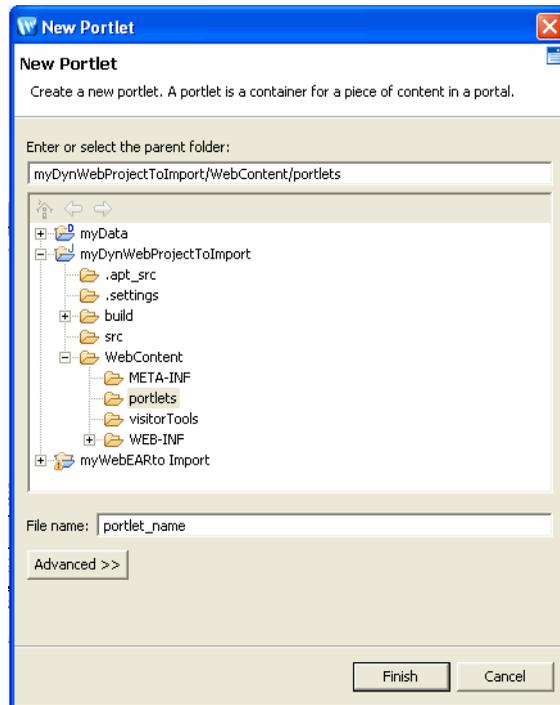
The above JSP is a very simple example of a JSF view. It contains a single line of text ("Simplest JSF Portlet") and a form with a Submit button that simply posts back to the portlet. This is a very simplistic example, but showcases the basics of writing a JSF view. The namingContainer component will be explained later. This component is basically used to namespace each portlet so that multiple instances of a JSF portlet can exist on the same page.

To create a JSF portlet, follow these steps:

1. Right-click in the Package Explorer view, within the Web Content directory, and select **File > 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.

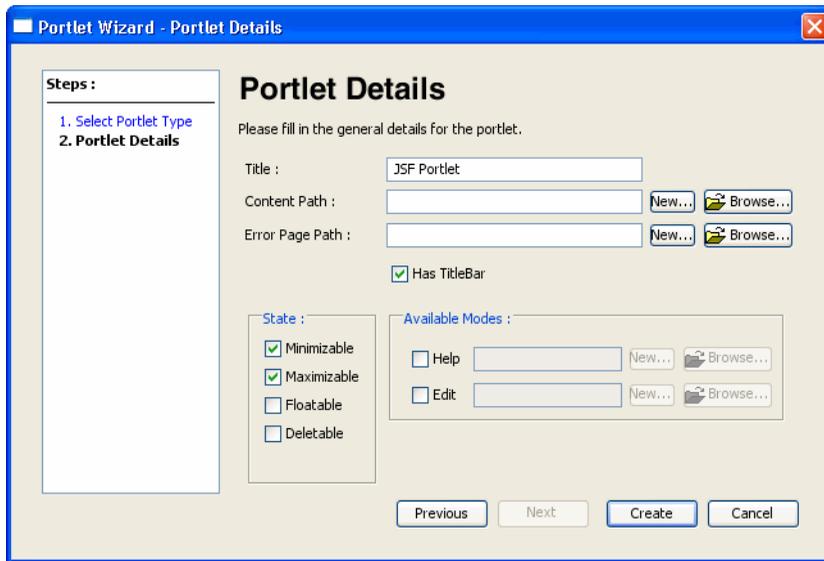
3. Type a file name for the new portlet.
4. Click **Next** to continue.

The Portlet Wizard displays the Select Portlet Type dialog.

- Click **Java Server Faces (JSF) Portlet** and then click **Next**. If this option is not available, the JSF facet was not added to the web project (see above instructions).

The Portlet Wizard displays the Portlet Details dialog; [Figure 5-14](#) shows an example.

Figure 5-14 Portlet Wizard - JSF Portlet Details Dialog



- 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, like the one created in the previous section.
Error Page Path	Every portlet can have a configured error page. This must be a standard JSP and not a JSF JSP. For details on configuring an error page, see “Portlet Properties.”

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

Field	Description
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.”
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.” Only the view mode supports JSF-JSP files. The other modes must be JSP or HTML files.

7. Click **Next** to Assign Supporting Files such as render dependencies and backing files.
8. Click **Create** to generate your new portlet.

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

Because JSF portlets are native portlet types in WLP, JSF portlets can be consumed in any .portal or streaming desktop just like any other portlet type.

The Artifacts

The portlet generated will have a `<netuix:facesContent>` element in the `.portlet.xml` file. [Listing 5-2](#) shows an example.

Listing 5-2 The `<netuix:facesContent>` Element in the `.portlet.xml` File of a Newly Created JSF Portlet

```
<?xml version="1.0" encoding="UTF-8"?>
<portal:root
xmlns:netuix="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0"
```

Building Portlets

```
xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/portal/support/1
.0.0 portal-support-1_0_0.xsd">
    <netuix:portlet
        definitionLabel="myFirstJsfPortlet"
            title="Myfirstjsfportlet">
        <netuix:titlebar>

        </netuix:titlebar>
        <netuix:content>
            <netuix:facesContent contentUri="/myFirstJSFJsp.faces"/>
        </netuix:portlet>
</portal:root>
```

WLP accomplished the JSF portlet capability by implementing a JSF bridge that connects the WLP portlet container to a standard implementation of JSF (specifically, the Sun Reference Implementation) which does the actual rendering of the JSF portlet. If there are multiple JSF portlets on the same page, each portlet is rendered as its own JSF view in isolation of the other portlets.

JSF Portlet Views

WebLogic Portal JSF portlets are simply JSF applications composed of one or more views. The technology used to create the view can vary based on needs.

View Technologies

JSF portlets can render with the following technologies:

- HTML or XHTML – The portlet developer can output either, but take care that the outer portal is rendering the same format.

- JSP 1.0 or 2.0 (XML) – The portlet developer can implement JSF portlet views in JSP, using either XML 1.0 or 2.0.
- Facelets – A declarative view technology that can be used instead of JSP.

Document Structure

Each JSF view contains components such as text, form fields, and buttons. These elements must be placed inside a properly constructed JSF view that adheres to certain rules. These rules differ when the JSF view appears inside of a portlet. This section discusses those differences.

f:view Tag

The JSF standard f:view tag is required for each JSF portlet. The WebLogic Portal framework properly handles the existence of multiple f:view components on the portal page, and uses each as a JSF naming container. The f:view tag contributes the root id namespace for the generated identifiers for the components within the portlet (at runtime, this namespace happens to be the portlet instance label). For information on identifiers and naming containers, see [“Namespacing” on page 12-51](#).

f:subview Tag

The JSF standard f:subview tag is supported within the portlet, but it cannot replace f:view as the root component.

HTML Document Tags

The following HTML page tags should be avoided within a body of a portlet, since a portlet does not own the entire web page.

- <html>
- <head>
- <body>

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.

To learn more about developing JSF Portlets, see [Chapter 12, “Working With JSF Portlets.”](#)

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.

Tip: A clipper portlet also lets you include remote web content in a portal page. For information on clipper portlets and how they differ from browser portlets, see [Chapter 6, “Creating Clipper Portlets.”](#)

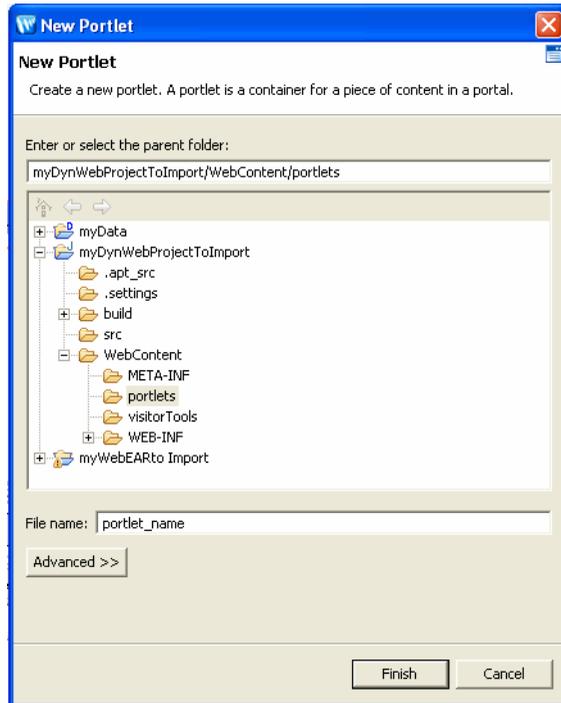
There are several ways to invoke the Portlet Wizard, as explained in the section [“Starting the Portlet Wizard” on page 5-7](#). This description assumes that you right-click in the Package Explorer view tree within a portal project and select **New > Portlet** from the menu.

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.

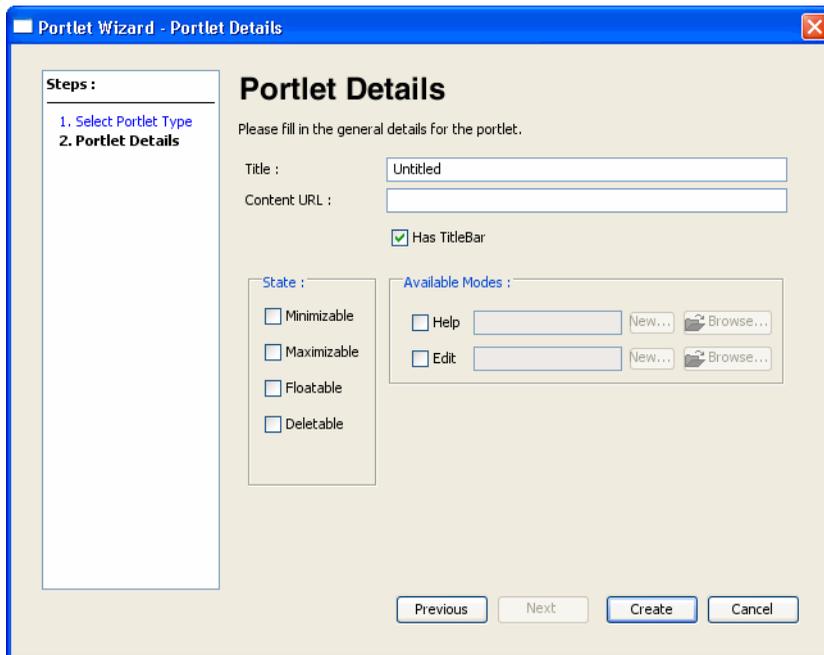
3. Type a file name for the new portlet.
4. Click **Finish** to continue.

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 5-94 .
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 5-85 .

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 7-13](#).

Clipper Portlets

A clipper portlet is a portlet that renders content from another web site. A clipper portlet can include all or a subset of another web site’s content using a process called “web clipping.” Clipper portlets are discussed in [Chapter 6, “Creating Clipper Portlets.”](#)

Struts Portlets

Use the Portlet Wizard to generate a portlet based on a Struts module, as explained in this section.

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](#).

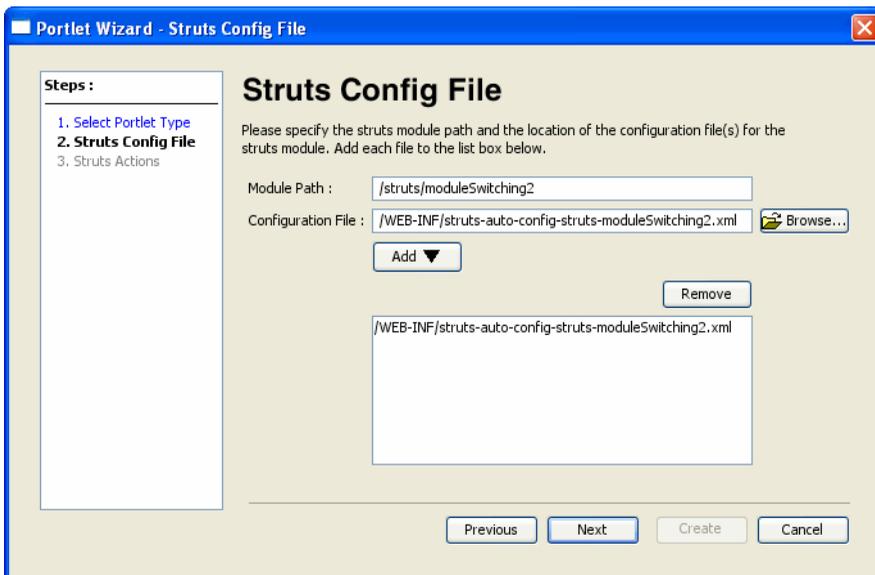
Tip: It is highly recommended that you fully develop and test a Struts application before attempting to host it within a portal. This helps to separate the complexities of developing

a working Struts application from the additional issues involved in putting the Struts application into a portlet.

To create a Struts portlet, follow these steps:

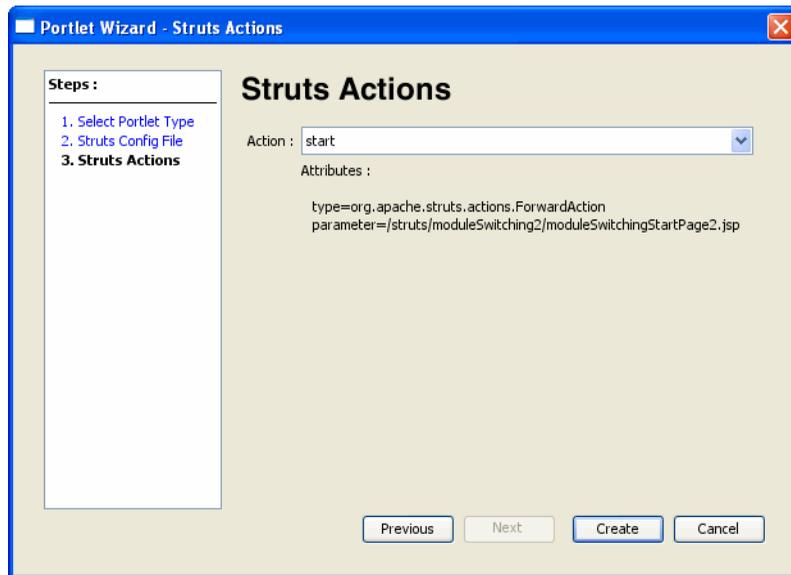
1. Right-click the Struts application module's XML configuration file located in the `WEB-INF` directory of the portal web application.
2. Select **Generate Portlet** from the menu. The wizard automatically collects and displays the module path and configuration file name(s) in the Struts Config File dialog. An example is shown in [Figure 5-17](#). Use the **Browse** and **Add** buttons to locate and add additional configuration files, if applicable.

Figure 5-17 Struts Config File Dialog



3. Click **Next**.
4. In the Struts Actions dialog, 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 were added previously.

Figure 5-18 Struts Actions Dialog



5. 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.

Configuring Multi-Part Form Data Support for a Struts Portlet

You can configure your Struts portlet to handle a multi-part struts action form after a server request has been posted.

Before you can create a Struts portlet, you must first integrate your existing Struts application into your portal web application. For detailed information about integrating Struts applications into WebLogic Portal, see "Integrating an Existing Web Application into Workshop for WebLogic" in the [Portal Development Guide](#).

To add the multi-part form data support to a struts portlet:

1. In Oracle Enterprise Pack for Eclipse, in your portal web application, create a module for the Struts portlet, as described earlier in this section.

2. In your portal web application, create a JSP page that contains an action form with the attribute `enctype= "multipart/form-data"`.
3. Open the Struts module's XML configuration file, `struts-auto-config.xml`, located in the WEB-INF directory of your portal web application.
4. Configure your struts module to point to the newly created JSP page, and save `struts-auto-config.xml`.
5. In `struts-auto-config-upload.xml`, add the following entries:

```
<controller
inputForward="true"
processorClass="com.bea.struts.adapter.action.AdapterRequestProcessor"
multipartClass="
com.bea.struts.adapter.action.ScopedMultipartRequestHandler"/>
```
6. Save `struts-auto-config-upload.xml`.
7. To access the struts application directly, open the browser and use the following URL:

`http://localhost:port/struts_webapp/module.do`

Where, `localhost:port` are the host name and port number where WebLogic Portal is deployed, `struts_webapp` is the name of your portal web application containing the struts module, and `module.do` is the module in which you want to implement the multi-part form data support. For example:

`http://localhost:7001/StrutsUploadWeb13/upload.do`

Note: If you want to run the struts portlet through WebLogic Portal, make sure your portlet points to the WebLogic Portal tag library, which is specified in the import statements at the beginning of a JSP file. If you do not use WebLogic Portal's HTML tag library, the page gets redirected outside of the portal page. As a result, the struts portlet's JSP page takes over the entire page.

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 [Oracle WebLogic Portal Federated Portlets 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 *Workshop for WebLogic*.

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 5-19](#).

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.

Note: Opening the same portal desktop in multiple browser windows that share the same process (and, therefore, the same session) is not supported. For example, using the `render:pageURL` tag or the JavaScript `window.open` function to open a portal desktop in a new window is not supported and can result in unwanted side effects. For instance, if a portlet is deleted in the new window, and then the user tries to interact with that portlet in the main window, the interaction will fail.

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 portlets 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");
```

Working with Inlined Portlets

A file-based portlet can exist either as a stand-alone `.portlet` file or as an inlined portlet. Typically, within the Workshop for WebLogic portal editing framework, `.portlet` files are included in portals by reference. For instance, when you drag a `.portlet` file onto a portal, page or book, a reference is created to the portlet file inside the portal, page, or book. On the other hand, an inlined portlet's entire XML definition is embedded directly in a page or book.

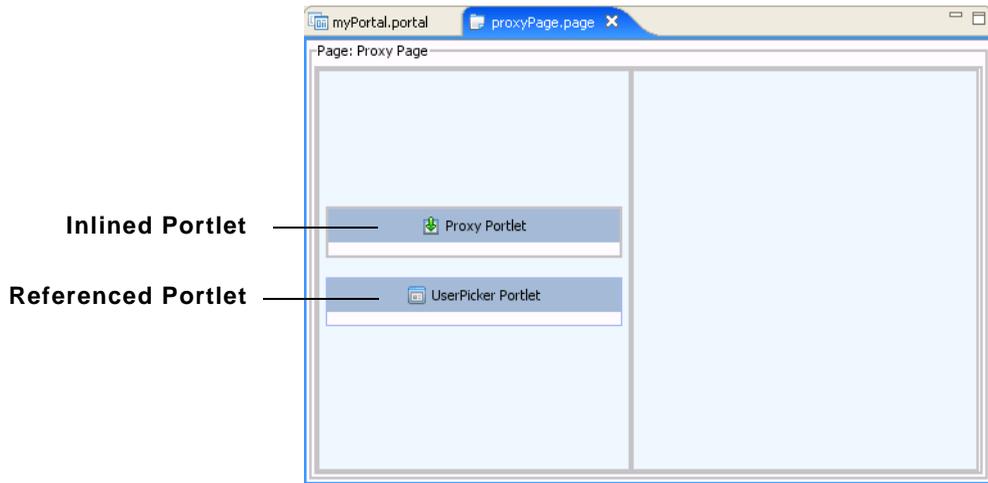
Inlined portlets are created under the following circumstances:

- If you create a remote book or page that contains portlets, those portlets will be inlined in the `.book` or `.page` file. For detailed information on creating remote books and pages, see the [Federated Portals Guide](#).
- If you use the Export/Import Utility to extract a `.book` or `.page` file that contains portlets, those portlets will be inlined if they were originally inlined. If the original page contained referenced portlets, they will be referenced when the page is extracted. For detailed information on the Export/Import Utility, see the [Production Operations User Guide](#).

You can drag and drop, cut, copy, and paste inline portlets from one page or book to another from within the portal editor.

[Figure 5-19](#) shows a remote page that contains an inlined portlet and a referenced portlet. Note that the icon used in an inlined portlet is distinct from a referenced portlet.

Figure 5-19 Inlined Portlet in the Portlet Editor



Tip: You can edit the properties of inlined portlets exactly like file-based portlets; however, portlet states and modes are not editable for inlined portlets.

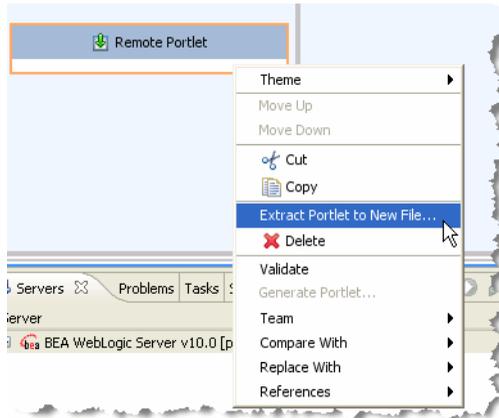
Extracting Inlined Portlets

You can export an inlined portlet to a `.portlet` file. When you do this, the resulting `.portlet` file is functionally equivalent to any other `.portlet` file. When you extract an inlined portlet, the inlined portlet XML code is automatically removed from the source file (a page or book) and replaced with a reference to the newly created `.portlet` file.

Note: After you extract an inlined portlet, you can undo the operation (re-inline the portlet). However, note that the `.portlet` file that was created during the extraction will not be deleted from your system. The source document will simply not reference the `.portlet` file any longer.

To extract an inlined portlet, do the following:

1. Right-click the inlined portlet in the Book or Page Editor and select **Extract Portlet to New File**, as shown in [Figure 5-20](#).

Figure 5-20 Extract Portlet to New File

2. In the Save As dialog, enter a name for the new portlet.

Tip: If you receive errors after extracting a portlet, be sure to save the source file and run the **Project > Clean** command.

Setting the Theme of an Inlined Portlet

You can set the theme of an inlined portlet exactly as you would for a referenced portlet. To set the theme, right-click the inlined portlet in the book or page editor, and pick a theme from the Theme menu. The theme is retained for that portlet as long as it remains referenced in the page or book.

Extracting Books and Pages

You can extract any book or page in a portal to a `.book` or `.page` file. Once a book or page is extracted, you are free to use it in another portal within the same portal web application if you wish.

The procedure for extracting books and pages is similar to the procedure for extracting inlined portlets, described in [“Extracting Inlined Portlets” on page 5-38](#). To extract a book or page, do the following:

1. Right-click border of the book or page in the Portal Editor and select **Extract Book (or Page) to New File**.

2. In the Save As dialog, enter a name for the new book or page file.

Tip: Any theme applied to a book or page is retained for an extracted book or page as long as the book or page remains referenced in the portal.

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 13-3](#) and [“Modifying Desktop Portlet Properties” on page 13-4](#).

For a detailed description of all portlet properties, refer to [“Portlet Properties in the Portal Properties View” on page 5-43](#) and [“Portlet Properties in the Portlet Properties View” on page 5-44](#).

This section contains the following topics:

- [Editing Portlet Properties](#)
- [Tips for Using the Properties View](#)
- [Portlet Properties in the Portal Properties View](#)
- [Portlet Properties in the Portlet Properties View](#)

Editing Portlet Properties

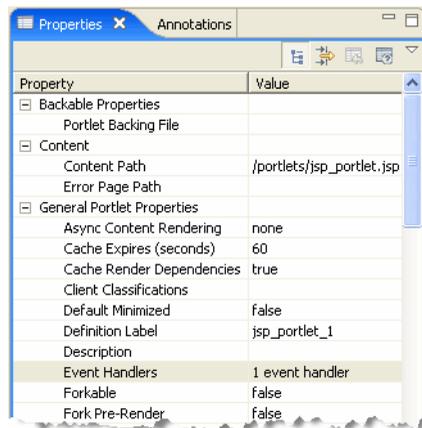
To edit portlet properties, follow these steps:

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 5-21](#) shows a segment of a JSP portlet's Properties view:

Figure 5-21 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 5-42.

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. Oracle 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 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.

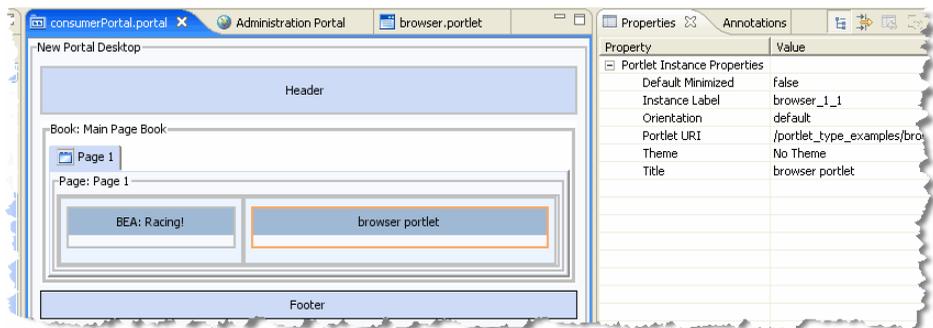
A WYSIWYG view of the portal appears in the editor.

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 5-22](#) shows an example.

Figure 5-22 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

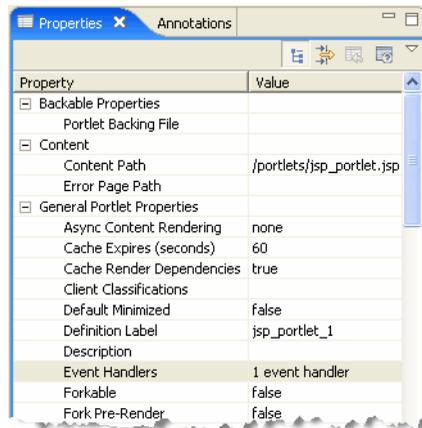
Property	Value
Default Minimized	Optional. Select <code>true</code> for the portlet to be minimized when it is rendered. The default value is <code>false</code> . Change the value for this property only if you want to override the default value provided by the <code>.portlet</code> file.
Instance Label	Required. A single portlet, represented by a <code>.portlet</code> 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: <code>default</code> , <code>top=0</code> , <code>left=1</code> <code>right=2</code> , <code>bottom=3</code> . Enter a value for this property only if you want to override the orientation specified in the <code>.portlet</code> file. Selecting <code>default</code> 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 <code>.portlet</code> file. For example, if the file is stored in <code>Project\myportlets\my.portlet</code> , the Portlet URI is <code>/myportlets/my.portlet</code> .
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 <code>.portlet</code> 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 5-1](#) shows a portion of the Properties view for a portlet.

Figure 5-1 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 <code>com.bea.netuix.servlets.controls.content.backing.JspBacking</code> or extend <code>com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking</code> . From the data field you can choose to browse to a class or open the currently displayed class.
Content	

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

Property	Value
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 <code>Project/myportlets/my.jsp</code> , the Content URI is <code>/myportlets/my.jsp</code> .
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 <code>Project/myportlets/error.jsp</code> , the Content URI is <code>/myportlets/error.jsp</code> .
General Portlet Properties	
Async Content Rendering	<p>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 <code>none</code>, <code>ajax</code>, <code>iframe</code>, and <code>iframe_unwrapped</code>. Portlet files that do not contain the <code>asyncContent</code> attribute appear with the initial value <code>none</code> displayed.</p> <p>For more information, refer to “Asynchronous Portlet Content Rendering” on page 7-13.</p> <p>Note: The <code>iframe_unwrapped</code> value is used for interoperability with WebCenter 10g ADF Faces portlets. You must use the <code>iframe_unwrapped</code> value if you are consuming (through WSRP) a WebCenter 10g ADF Faces portlet in a WebLogic Portal. Using this value prevents potential rendering problems by wrapping the ADF Faces portlet in an IFrame, while explicitly excluding WebLogic Portal-specific markup from rendering within the IFrame. For more information on WSRP interoperability between WebCenter and WebLogic Portal, see the Federated Portals Guide.</p> <p>Tip: You can also enable asynchronous rendering for an entire portal desktop by setting a portal property in either Workshop for WebLogic or the WebLogic Portal Administration Console. For more information on asynchronous desktop rendering, see the WebLogic Portal Development Guide.</p>
Cache Expires (seconds)	Optional. When the <code>Render Cacheable</code> property is set to <code>true</code> , enter the number of seconds after which the portlet cache expires.

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

Property	Value
Cache Render Dependencies	<p>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. See also “Portlet Dependencies” on page 5-77.</p> <p>The value defaults to <code>true</code> if the attribute is not already included in the <code>.portlet</code> 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.</p> <p>This property does not affect posts targeted to the portlet.</p>
Client Classifications	<p>Optional. Select the multichannel devices on which the portlet can be viewed. The list of displayed devices is obtained from the file <code>Project_Path\WEB-INF\client-classifications.xml</code>. 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.</p> <p>In the Manage Portlet Classifications dialog:</p> <ol style="list-style-type: none"> 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. <p>When you disable classifications for a portlet, the portlet is automatically enabled for the classifications that you did not select for disabling.</p>
Default Minimized	<p>Required. Select <code>true</code> if you want the portlet to be minimized when it is rendered. The default value is <code>false</code>.</p>
Definition Label	<p>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.</p> <p>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.</p> <p>Note: When you create a portlet instance on a desktop using the WebLogic Portal Administration Console, the generated definition label is not editable.</p>

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

Property	Value
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 <code>No event handlers</code> . To select or add an event handler, click Browse in the Properties view. You can also click the Event Handlers link in the portlet editor. Both of these methods bring up the Portlet Event Handlers dialog box.
Forkable	For details on this property, refer to “Portlet Forking” on page 7-3 .
Fork Pre-Render	For details on this property, refer to “Portlet Forking” on page 7-3 .
Fork Pre-RenderTimeout (seconds)	For details on this property, refer to “Portlet Forking” on page 7-3 .
Fork Render	For details on this property, refer to “Portlet Forking” on page 7-3 .
Fork Render Timeout (seconds)	For details on this property, refer to “Portlet Forking” on page 7-3 .
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 <code>.portal</code> file. Following are the numbers used in the <code>.portal</code> file for each orientation value: <code>top=0</code> , <code>left=1</code> , <code>right=2</code> , <code>bottom=3</code> . 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 <code>packed="false"</code> (the default), the portlet takes up as much horizontal space as it can. When <code>packed="true"</code> , 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 <code>packed="false"</code> , the table's relative width would likely be set to <code>"100%"</code> . When <code>packed="true"</code> , the table width would likely remain unset.

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

Property	Value
Render Cacheable	Optional. To enhance performance, set to <code>true</code> 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 7-2 .
Required User Properties Mode	Optional. Possible values are <code>none</code> , <code>all</code> , or <code>specified</code> . If the value is <code>specified</code> , 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 <code>specified</code> in the Required User Properties Mode field; enter a comma-delimited list of property names.
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 5-43).
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 <code>begin</code> 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 <code>onRefresh</code> portal event to invoke the page flow action.

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

Property	Value
Request Attribute Persistence	<p>Optional. Possible values are <code>none</code>, <code>session</code>, and <code>transient-session</code>. This attribute controls attribute persistence for Page Flow, JSF, and Struts portlets. The default is <code>session</code>, 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 <code>session</code> can cause session memory consumption and replication that would not otherwise occur in a standalone Page Flow, JSF, or Struts application. The value <code>transient-session</code> places a serializable wrapper class around a <code>HashMap</code> into the session. The value <code>none</code> performs no persistence operation.</p> <p>JPF or Struts portlets that have the <code>transient-session</code> 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 <code>session</code> persistence portlets.</p> <p>Portlets that have the <code>none</code> 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.</p>
Java Server Faces (JSF) Content	
Faces Events	<p>(Optional) Lets you add name/action pairs to a JSF portlet. The name field is simply an alias. Event handlers (and the Event Handler dialog) can simply reference this name. The action is a reference to a JSF view ID, such as <code>myfaces/foo.face</code>. For more information on adding event handlers, see “Portlet Events” on page 9-2.</p>
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	<p>A CSS class that overrides any default CSS class used by the component's skeleton.</p> <p>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.</p> <p>Sample: If you enter "my-custom-class", the rendered HTML from the default skeletons looks like this:</p> <pre data-bbox="467 630 868 652"><div class="my-custom-class"></pre> <p>The properties you enter are added to the component's parent <div> tag. This property also applies to books and pages. For more information, refer to the Portal Development Guide.</p>
Content Presentation Style	<p>Optional. The primary uses are to allow content scrolling and content height-setting.</p> <p>For scrolling, enter the following attributes:</p> <ul data-bbox="467 883 1053 909" style="list-style-type: none"> • overflow:auto – Enables vertical and horizontal scrolling <p>For setting height, enter the following attribute:</p> <ul data-bbox="467 961 646 987" style="list-style-type: none"> • height:200px <p>where 200px is any valid HTML height setting.</p> <p>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 <div> tag.</p>
Offer as Remote	<p>Optional. Defines whether the portlet is accessible using the WSRP producer. The default is <code>true</code>, which allows the portlet to be accessed. For more information about entitling remote portlets, refer to the Federated Portals Guide.</p>
JSP Content	
Content Backing File	<p>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 <code>com.bea.netuix.servlets.controls.content.backing.JspBacking</code> or extend <code>com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking</code>.</p>

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

Property	Value
Thread Safe	<p>Optional. Performance setting for books, pages, and portlets that use backing files.</p> <p>When Thread Safe is set to <code>true</code>, 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.</p> <p>When Thread Safe is set to <code>false</code>, a new instance of a backing file is created each time the backing file is requested by a different book, page, or portlet.</p>
Portlet Title Bar	
Can Delete	Optional. If set to <code>true</code> the portlet can be deleted from a page.
Can Float	Optional. If set to <code>true</code> the portlet can be floated into a separate window. For instructions on creating a floatable Java portlet, which requires editing the <code>weblogic-portlet.xml</code> file, in “Customizing Java Portlets Using <code>weblogic-portlet.xml</code>” on page 5-17.
Can Maximize	Optional. If set to <code>true</code> the portlet can be maximized.
Can Minimize	Optional. If set to <code>true</code> 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	<p>Required. The path (relative to the project) to the JSP, HTML, or <code>.java</code> file to be used for portlet's mode content, such as the edit page. For example, if the content is stored in <code>Project/myportlets/editPortlet.jsp</code>, the Content URI is <code>/myportlets/editPortlet.jsp</code>.</p> <p>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 <code>.java</code>.</p>

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

Property	Value
Error Path	<p>Optional. The path (relative to the project) to the JSP, HTML, or .java 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.</p> <p>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 .java.</p>
Portlet Backing File	<p>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 <code>com.bea.netuix.servlets.controls.content.backing.JspBacking</code> or extend <code>com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking</code>.</p>
Visible	<p>Optional. Makes the mode icon (such as the edit icon) in the title bar or menu invisible (<code>false</code>) or visible (<code>true</code>). Set Visible to <code>false</code> when, for example, you want to provide an edit URL in a desktop header.</p>
Mode Toggle Button Properties	
Name	<p>Optional. Displayed when you select an individual mode. An optional name for the mode, such as <code>Edit</code>.</p>
Presentation Properties	
Presentation Class	<p>This property is described in the <i>Portal Development Guide</i>.</p>
Presentation ID	<p>This property is described in the <i>Portal Development Guide</i>.</p>
Presentation Style	<p>This property is described in the <i>Portal Development Guide</i>.</p>
Properties	<p>Optional. A comma-delimited list of name-value pairs to associate with the object. This information can be used by skeletons to affect rendering.</p>
Skeleton URI	<p>This property is described in the <i>Portal Development Guide</i>.</p>
Proxy Portlet Properties	
Connection Establishment Timeout	<p>Optional. The number of milliseconds after which this portlet will time out when establishing an initial connection with its producer.</p>

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 <code>WEB-INF/wsrp-producer-registry.xml</code> 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	<p>This boolean property allows the consumer to obtain render dependencies from the producer during the pre-render life cycle of a proxy portlet.</p> <p>When a portlet on a producer has a <code>lafDependenciesUri</code> value, the producer exposes the <code>invokeRenderDependencies</code> boolean in the portlet description. For more information on this attribute, refer to “Portlet Dependencies” on page 5-77.</p> <p>Note: Provide an absolute path for the <code>lafDependenciesUri</code> attribute, rather than a relative path.</p> <p>The value defaults to <code>false</code> if the attribute is not included in the <code>.portlet</code> file. The value is read-only, and is initialized from the producer whenever a proxy portlet is generated from the portlet wizard.</p>
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 <code>true</code> .
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	<p>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 <code>true</code>. For third party producers it can be <code>true</code> or <code>false</code>, depending on the response from <code>getServiceDescription</code>. If it is <code>false</code>, the entire user context will be sent on every <code>getMarkup</code> and <code>performBlockingInteraction</code> request. If <code>true</code> it will be sent only once per producer session.</p> <p>This boolean value defaults to <code>false</code> if the attribute is not included in the <code>.portlet</code> file.</p> <p>The value is read-only, and is initialized from the producer whenever a proxy portlet is generated from the portlet wizard.</p>
Struts Content	
Listen To	<p>(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 9, "Local Interportlet Communication."</p>
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	<p>The struts module that is associated with this struts portlet.</p> <p>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 <code>struts-config.xml</code> file.</p>
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 “ORCL, 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](#)
- [Using the Preferences API to Access or Modify Preferences](#)
- [Portlet Preferences SPI](#)
- [Best Practices in Using Portlet Preferences](#)

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 5-63, 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>ORCL, 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 `ORCL, 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>ORCL</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>ORCL</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.

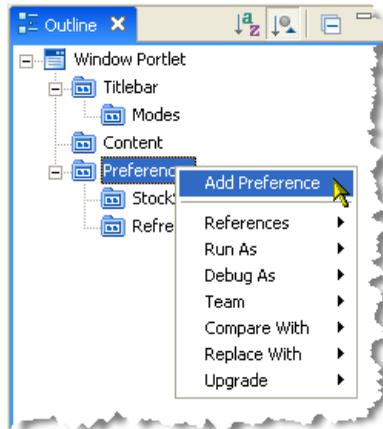
Specifying Preferences for Other Types of Portlets using Workshop for WebLogic

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

Property	Value
[-] New Preference Portlet Preference	
Modifiable	true
Multi Valued	true
Preference Description	Stock symbol pref
Preference Name	StockSymbols
Preference Value	BEAS, MSFT

[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 <code>true</code> .
Multi Valued	Indicates whether the preference can have multiple values. The default is <code>true</code> . 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 <code>java.lang.String</code> .

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
<code>String getValue(String name, String default)</code>	Use this method to get the first value of a preference.
<code>String[] getValues(String name, String[] defaults)</code>	Use this method to get all the values of a preference.
<code>boolean isReadOnly(String name)</code>	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
<code>Enumeration getNames()</code>	Use this method to get an enumeration of the names of all preferences.
<code>Map getMap()</code>	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 <code>getValues(String name, String[] defaults)</code>

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
<code>void setValue(String name, String value)</code>	Use this method to set the value of a preference
<code>void setValues(String name, String[] values)</code>	Use this method to set several values for a preference
<code>void store()</code>	Use this method to commit the changes made to preferences for a portlet.
<code>void reset(String name)</code>	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 Portlet Displays a Form to Edit Preferences

```
<%@ 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", "ORCL, 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(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 `getPortletPreferences(HttpServletRequest req)` 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
<code>getPreference</code>	Use this tag to get the value of a portlet preference.
<code>getPreferences</code>	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.
<code>forEachPreference</code>	Use this tag to iterate through all the preferences of a portlet. You can nest other tags (<code>getPreference</code> , <code>getPreferences</code> , <code>ifModifiable</code> and <code>Else</code>) inside this tag.

Table 5-11 JSP Tags for Getting Portlet Preferences

Method	Purpose
<code>ifModifiable</code>	Use this tag to include the body of this tag if the given portlet preference is not read-only.
<code>else</code>	Use this tag in conjunction with the <code>ifModifiable</code> 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
{
    /**
```

Building Portlets

```
* 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 5-72](#)) 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 9, “Local Interportlet Communication”](#) uses backing files.

This section contains the following topics:

- [How Backing Files are Executed](#)
- [Thread Safety and Backing Files](#)
- [Backing File Guidelines](#)
- [Adding a Backing File Using Workshop for WebLogic](#)

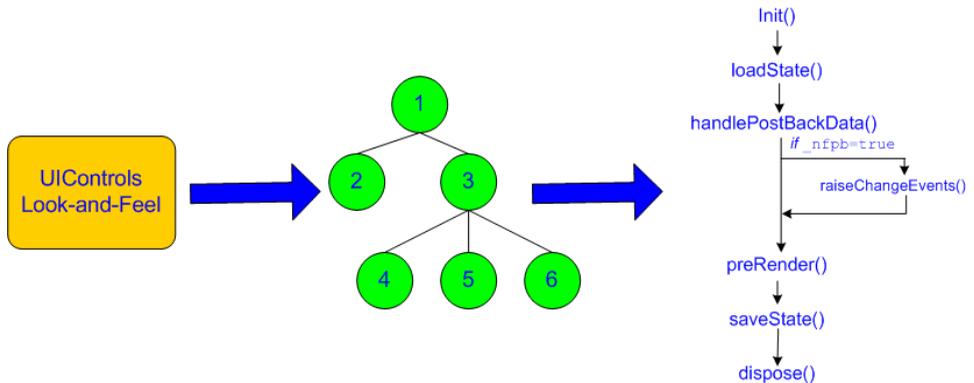
How Backing Files are Executed

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 `HttpServletRequest` object; Oracle 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

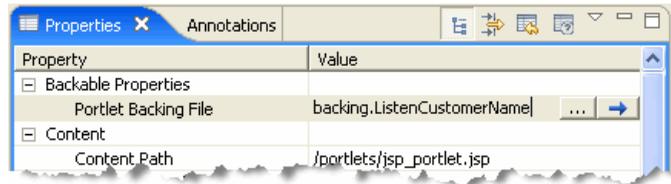
You can add a backing file to a portlet either from within Workshop for WebLogic or by coding it directly into the file to which you are attaching it. Simply specify the backing file in the **Backing File** field of the Properties view, as shown in [Figure 5-26](#). You need to specify the backing directory 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](#)
- [Creating Custom 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 concepts related to skin and skeleton resource dependencies are 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](#)
- [Creating and Editing a Dependency File](#)
- [Example Dependency Files](#)
- [Considerations and Limitations](#)

- [Scoping JavaScript Variables and CSS Styles](#)
- [Rewriting Resource URLs](#)

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 5-82](#).

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.

Creating and Editing a Dependency File

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

Tip: For example dependency files, see [Listing 5-12](#), [Listing 5-13](#), and [Listing 5-14](#).

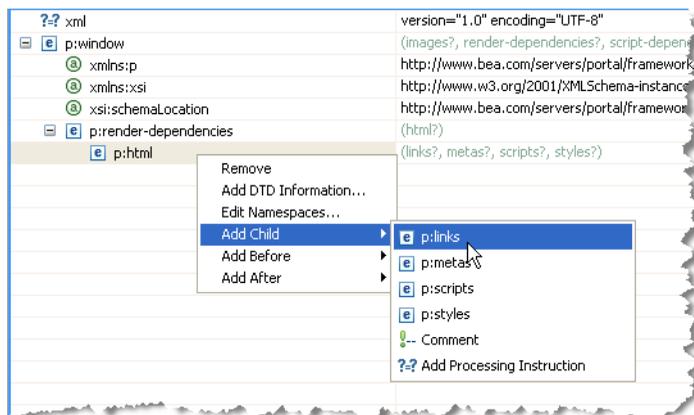
The simplest way to create a dependency file is to select **File > New > Other > Markup Files > Render Dependencies**. The `.dependencies` file must reside in a WebLogic Portal framework project, within the web content folder (typically named `WebContent`).

You can also create a dependency file as follows:

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`.

You can use the Workshop for WebLogic XML editor to add elements and attributes to the dependency file. Right-click on an element and use the menu to select child elements and add attributes. As shown in [Figure 5-27](#), valid choices based on the schema file are automatically populated in the menu.

Figure 5-27 Editing a Dependencies File



Tip: The standard Eclipse hover-window help is available for XML schema elements in the **Source** view of the XML editor. Simply hover the mouse pointer over the element and a

help pop-up appears. Also, in the Source view, you can click in an element and press F2 to display the help pop-up.

Example Dependency Files

This section includes the following examples:

- [Including JavaScript in a Render Dependencies File](#)
- [Including Meta and Style Elements in a Render Dependencies File](#)

Including JavaScript in a Render Dependencies File

[Listing 5-13](#) illustrates how to include both an external JavaScript file as well as an embedded script.

Listing 5-13 Including JavaScript

```
<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:scripts>
      <p:search-path>
        <p:path-element>.</p:path-element>
      </p:search-path>
      <p:script type='text/javascript' src='my.js' />
      <p:script type='text/javascript'>
        alert('hello world');
      </p:script>
    </p:scripts>
  </p:html>
</p:render-dependencies>
</p:window>
```

Including Meta and Style Elements in a Render Dependencies File

[Listing 5-14](#) shows the use of both the `metas` and `styles` elements. The `metas` element lets you specify HTML meta tags, and the `styles` element lets you embed HTML style tags.

Listing 5-14 Use of Meta and Styling Elements

```
<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:metas>
      <p:meta name='keywords' content='pirate, ninja' />
    </p:metas>
    <p:styles>
      <p:style type='text/css'>
        div.myClass {
          background-color: red;
        }
      </p:style>
    </p:styles>
  </p:html>
</p:render-dependencies>
</p:window>
```

Considerations and Limitations

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

Oracle 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.

Scoping JavaScript Variables and CSS Styles

Whenever you place multiple instances of a portlet on a page, you can encounter scoping problems with JavaScript variables and CSS styles. For example, if a portlet includes inlined JavaScript and you place two instances of that portlet on a page, it is possible that changing a JavaScript variable in one portlet will affect the other portlet.

To ensure that JavaScript and CSS styles are scoped to a specific portlet instance, add the token `wlp_rewrite_` to the front of the variable or style class name. When the portlet is rendered, this token is replaced by the portlet instance label, which is unique for each portlet instance.

For example, to ensure portlet instance-level scoping of a JavaScript variable called `stockQuote` that is defined in a `.js` file that is referenced from a `.dependencies` file, you need to append `wlp_rewrite_` to the front of the variable name:

```
var wlp_rewrite_stockQuote
```

To ensure portlet instance-level scoping of a CSS class name called `portlet_bg` that is defined in a `.css` file that is referenced from a `.dependencies` file, you need to append `wlp_rewrite_` to the front of the class name. For example:

```
.wlp_rewrite_portlet_bg { background_color:white; }
```

In both of these cases, the `wlp_rewrite_` token is replaced by the portlet's instance label, which is a unique identifier.

Note: The scoping mechanism described in this section only works for `.css` and `.js` files that are referenced with the `content-uri` dependency file attribute. Files linked with the `src` attribute or the `link` tag will not be rewritten.

Rewriting Resource URLs

WebLogic Portal also has the ability to rewrite URLs contained in the *content* of files referenced in a `<script>` or `<style>` section of a `.dependencies` file. The URLs are rewritten based on the standard Look and Feel URL Templating mechanism using the `window-resource-url-template-ref`, as described in the section [Optional Look And Feel URL Templates](#) in the *WebLogic Portal Development Guide*.

For example, if your `.dependencies` file specifies:

```
...
    <p:styles>
        <p:search-path>
```

```
<p:path-element>styles</p:path-element>
</p:search-path>
<p:style content-uri='my-style.css' type='text/css'/>
</p:styles>
```

...

and the contents of `my-style.css` look like this:

```
.my_portlet_bg
{
    background-image: url('wlp_rewrite?/images/picture.gif/wlp_rewrite');
}
```

then when the portlet is rendered, the value of `/images/picture.gif` will be templated based on the standard Look and Feel URL Templating mechanism.

In a non-WSRP case, this value can take one of two forms:

- If a `window-resource url-template-ref` is found, the value returned will be templated accordingly. Using the example from the section [Optional Look And Feel URL Templates](#) in the *WebLogic Portal Development Guide*, this value would be `http://my.domain.com/resources/laf/images/picture.gif`.
- If no `window-resource url-template-ref` is found, the value returned in this case will be `http://host:port/context/images/picture.gif`.

When the portlet that specifies the `.dependencies` file is being accessed remotely through a WLP remote (WSRP) portlet, then this rewritten URL also takes one of two forms:

- If a `window-resource url-template-ref` is found on the producer, the value returned will be templated accordingly. Using the example from the section [Optional Look And Feel URL Templates](#) in the *WebLogic Portal Development Guide*, this value would be `http://my.domain.com/resources/laf/images/picture.gif`. In this case, the resulting URL may or may not be rewritten to be proxied via the `ResourceProxyServlet`; it is up to the URL template developer to ensure that resources are serviced appropriately.
- If no `window-resource url-template-ref` is found on the producer, the value returned in this case will be templated based on the `resource url-template-ref` (see the section [URL Templates and Web Services for Remote Portlets \(WSRP\)](#) in the *WebLogic Portal Development Guide*. In this way, the resource URL will be wrapped in a `ProxyResourceServlet` URL, so that it can be served by the producer from the consumer.

Note: The rewriting mechanism described in this section only works for files that are referenced with the `content-uri` dependency file attribute. Files linked with the `src` attribute or the `link` tag will not be rewritten.

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 5-28](#) shows an example of the default buttons for the portlet modes when displayed in the editor; [Figure 5-29](#) shows the appearance of the mode icons in a running portlet.

Figure 5-28 Portlet Mode and State Buttons in Editor

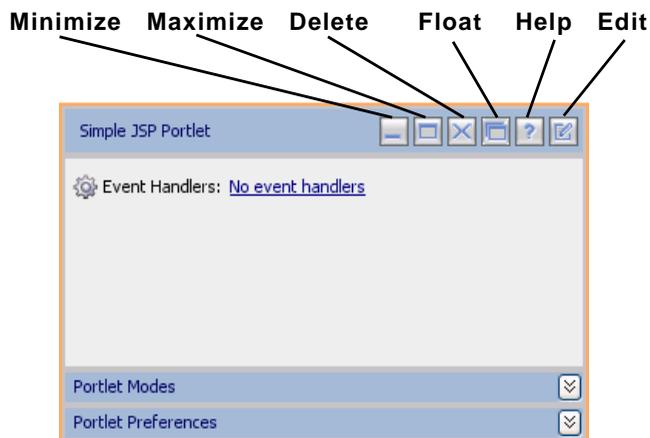
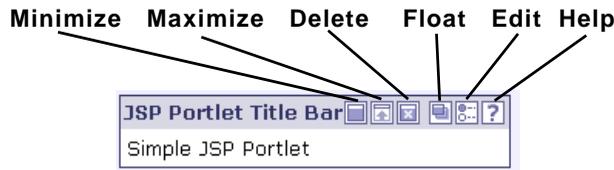


Figure 5-29 Portlet Mode and State Buttons in a Running Portlet



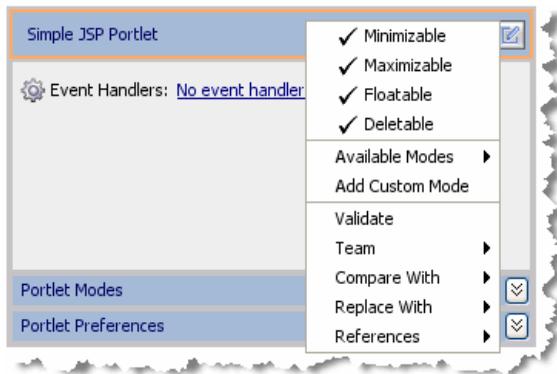
When you use the Portlet Wizard to create a portlet, mode and state settings are available on the Portlet Details dialog. These settings can also be edited in the portlet’s Properties view: The following sections describe possible methods of performing these tasks.

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:

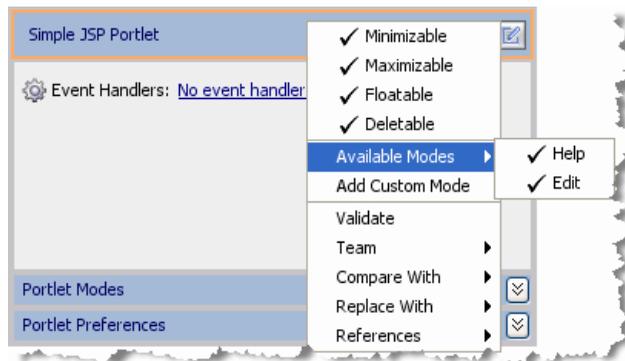
1. Display the portlet for which you want to add or remove a mode.
2. Right-click the title bar of the displayed portlet to display the context menu. [Figure 5-30](#) shows an example of the title bar context menu.

Figure 5-30 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 5-31](#) shows an example of the submenu.

Figure 5-31 Portlet Mode - Available Modes Submenu

4. Click the mode for which you want to change the availability status. For example, in [Figure 5-31](#), 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.

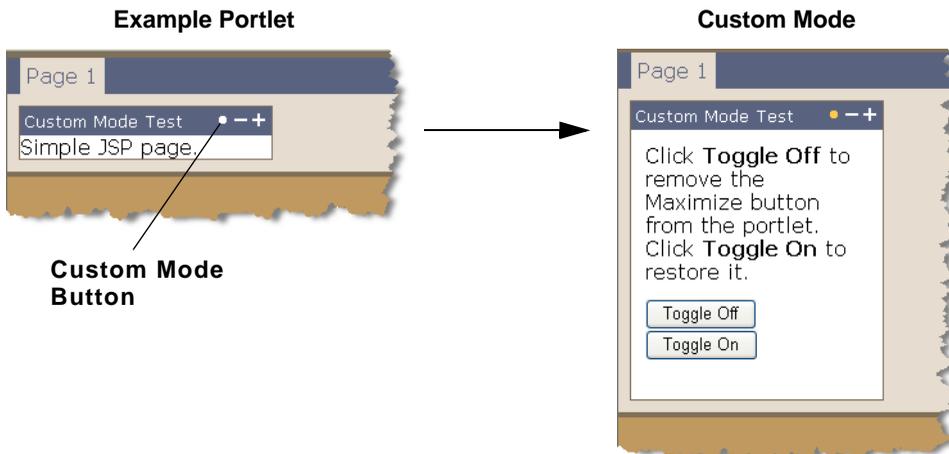
For descriptions of the mode properties, refer to [Table 5-7](#).

Creating Custom Modes

A custom mode is a portlet mode that you implement. Like with the help and edit modes, a custom mode is activated with a button that appears in the portlet's title bar. To implement a custom mode, you need to supply a display part, typically a JSP, and a backing file. This section includes an example that explains how to create a simple custom mode that lets a user add or remove the Maximize button from a portlet. Once you understand the basic principles involved in writing a custom mode, you can create a custom mode to perform the specific tasks you want.

Figure 5-32 shows the example portlet and the portlet’s custom mode view. When the user clicks the custom mode button in the example portlet on the left, the portlet display changes to the custom mode view on the right. In this example, the custom mode offers a way for the user to add or remove the portlet’s Maximize button.

Figure 5-32 Selecting a Custom Mode



1. Create a JSP portlet in which to embed the custom mode. For information on JSP portlets, see “JSP and HTML Portlets” on page 5-11. For this example, any JSP portlet will suffice.
2. Create a JSP page to display the custom mode view when a user clicks the custom mode button. For example, Listing 5-15 shows a JSP for a custom mode that lets a user add or remove the Maximize button from a portlet. The code to execute this action is in a backing file, which is discussed next. In this example, the JSP is called `togglebutton.jsp`.

Listing 5-15 Sample Custom Mode JSP

```
<%@ page import="com.bea.portlet.PostbackURL"%>
<%
    PostbackURL url = PostbackURL.createPostbackURL(request, response);
%>
<TABLE CELLSPACING="10" ID="toggleButtonsTable">
    <TH>Using a Button and Backing File</TH>
    <TR>
```

```

<TD>
Click <b>Toggle</b> Off to remove the Maximize button from the portlet.<br>
Click <b>Toggle On</b> to restore it.
</TD>
</TR>
<TR>
<TD>
<FORM method="post" name="Toggle" action="%url.toString()">
<INPUT ID="toggle_off" TYPE="SUBMIT" NAME="toggle_off" VALUE="Toggle
Off">
<INPUT ID="do_nothing" TYPE="SUBMIT" NAME="do_nothing" VALUE="Toggle
On">
</FORM>
</TD>
</TR>
</TABLE>

```

3. Create a backing file for the custom mode. [Listing 5-16](#) implements the `JspBacking` interface and implements the `preRender()` method of that interface. In this example, the `preRender()` method removes the Maximize button from the portlet in response to a request. Refer to [Javadoc](#) for details on the API used in this example.

Listing 5-16 Sample Backing File

```

package modes;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.bea.netuix.servlets.controls.content.backing.JspBacking;
import com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext;
import com.bea.netuix.servlets.controls.window.WindowCapabilities;
import com.bea.pl3n.util.debug.Debug;

public class MyMode implements JspBacking {

    public void dispose() {
    }
}

```

Building Portlets

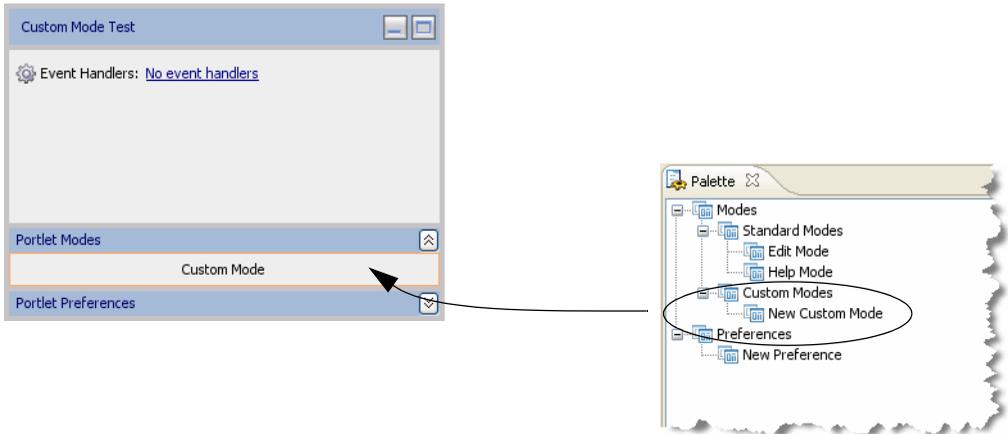
```
public boolean handlePostBackData(HttpServletRequest request,
    HttpServletResponse response) {
    return true;
}

public void init(HttpServletRequest request, HttpServletResponse response) {
}

public boolean preRender(HttpServletRequest request, HttpServletResponse
    response) {
    PortletBackingContext pbc =
    PortletBackingContext.getPortletBackingContext(request);
    if (request.getParameter("toggle_off") != null)
    {
        try
        {
            pbc.setCapabilityVisible(WindowCapabilities.MAXIMIZED.getName(),
                false);
        }
        catch (NullPointerException npe)
        {
            //
        }
    }
    return true;
}
}
```

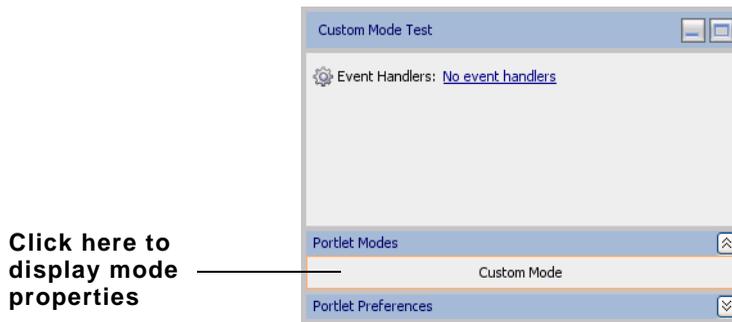
4. Add a new custom mode to the portlet by dragging the New Custom Mode icon from the Design Palette to the portlet, as shown in [Figure 5-33](#). You will be prompted to enter a name for the mode. You can enter a name now, or accept the default and change the name later.

Figure 5-33 Adding a New Custom Mode



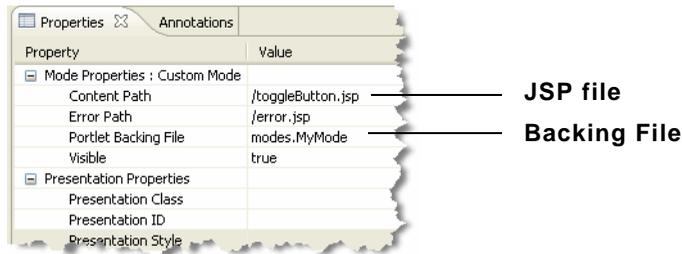
5. Open the Properties view for the custom mode. To do this, click in the Custom Mode region of the portlet in the portlet editor, as shown in [Figure 5-34](#). The properties for the custom mode appear in the Properties view.

Figure 5-34 Displaying Mode Properties



6. In the Properties view, enter the path of the custom mode JSP in the Content Path field. This is the JSP that is displayed when the mode is activated. You can find the Content Path field in the Mode Properties section of the Properties view, as shown in [Figure 5-35](#).
7. In the Properties view, enter the name of the backing file class, including the full package name. You can find the Portlet Backing File field in the Mode Properties section of the Properties view, as shown in [Figure 5-35](#).

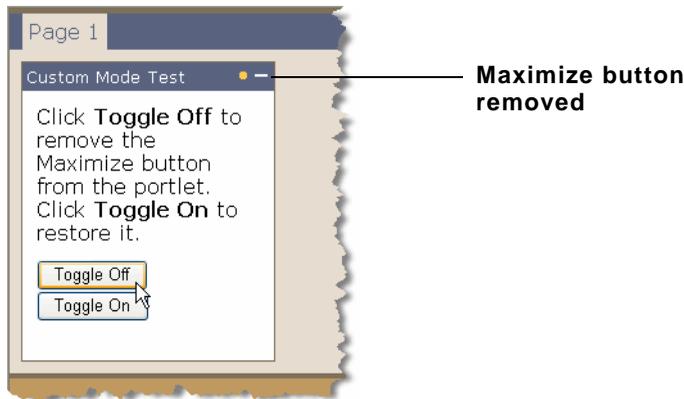
Figure 5-35 Specifying a Content File and a Backing File



Tip: The Properties view lets you set many other custom mode properties, such as an image for the custom mode button, a rollover image, button text, alternate text, and others. Refer to [Table 5-12](#) at the end of this section for information on each of the custom mode properties.

8. Test the custom mode by placing the example portlet in a portal and running it on the server. Select the portlet's custom mode button, as shown previously in [Figure 5-32](#), to display the custom mode view. Click **Toggle Off** to remove the Maximize button, as shown in [Figure 5-36](#).

Figure 5-36 Testing the Example



[Table 5-12](#) briefly describes each of the custom mode properties:

Table 5-12 Custom Mode Properties

Property	Value
Mode Properties	
Content Path	Required. The path (relative to the project) to the file/class to be used for the custom mode 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 <code>Project/myportlets/my.jsp</code> , the Content URI is <code>/myportlets/my.jsp</code> .
Error Path	Optional. The path (relative to the project) to the JSP, HTML, or page flow file to be used for the error message if the portlet's mode page cannot be rendered. For example, if the error page is in <code>project/myportlets/errorPortletEdit.jsp</code> , the Content URI is <code>/myportlets/errorPortletEdit.jsp</code> .
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 <code>JspBacking</code> interface or extend <code>AbstractJspBacking</code> . From the data field you can choose to browse to a class or open the currently displayed class.
Visible	Optional. Makes the mode icon in the title bar or menu invisible (<code>false</code>) or visible (<code>true</code>). Set <code>Visible</code> to <code>false</code> when, for example, you want to provide a custom mode URL in a desktop header.
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 .

Table 5-12 Custom Mode Properties

Property	Value
Toggle Button Properties	
Activate Alternate Text	Popup text that appears when the mouse pointer hovers over the custom mode button.
Activate Image	An image for the button that activates the custom mode. Place the image in the images directory of the skin that your portal uses.
Activate Rollover Image URI	Provides a rollover image for the custom mode button. Place the image in the images directory of the skin that your portal uses.
Active	Not generally used, but available for use by custom skeletons.
Alternate Text	Not generally used, but available for use by custom skeletons.
Deactivate Alternate Text	Popup text that appears when the mouse pointer hovers over the custom mode button.
Deactivate Image URI	An image for the button that deactivates the custom mode. Place the image in the images directory of the skin that your portal uses.
Deactivate Rollover Image UI	Provides a rollover image for the button that deactivates the custom mode. Place the image in the images directory of the skin that your portal uses.
Image	Not generally used, but available for use by custom skeletons.
Name	The name of the custom mode. If specified, the name appears in the Portlet editor view, Outline view, and Properties view. If no name is supplied, a default name is used.
Rollover Image	Not generally used, but available for use by custom skeletons.

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.

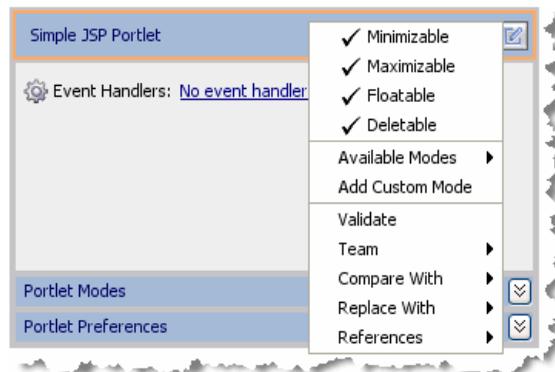
Modifying Portlet States in Workshop for WebLogic

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 5-37](#) shows an example of the title bar context menu showing all states as available.

Figure 5-37 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 5-37](#), 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 J2EE Shared Library`; 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:

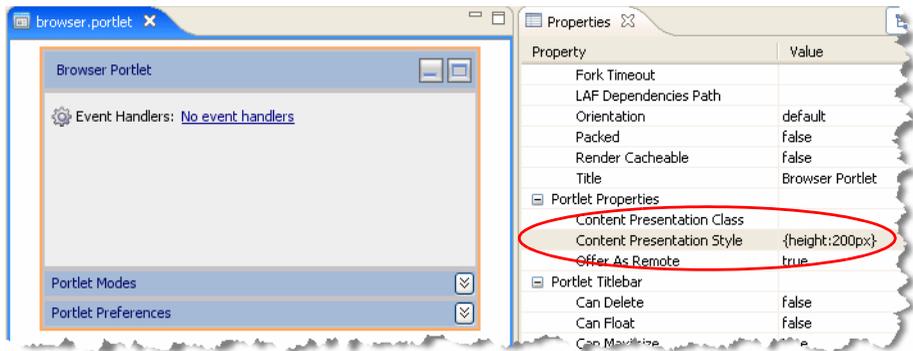
1. Open a portlet in the workbench editor.
2. Click the outer border of the portlet to display the portlet properties in the Properties view.

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.

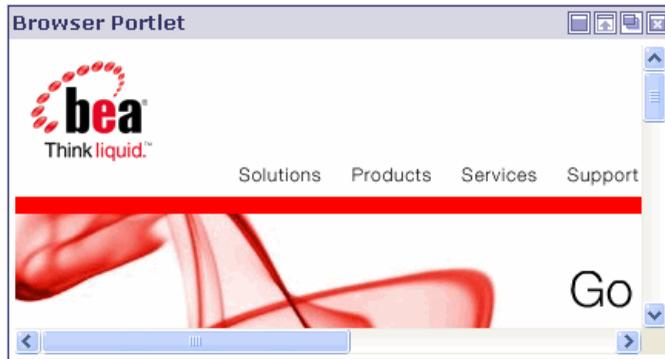
Figure 5-38 shows an example of a height property, set using Content Presentation Style.

Figure 5-38 Portlet Height and Scrolling Presentation Properties Example



Based on the entries shown in Figure 5-38, the result looks similar to the example in Figure 5-39.

Figure 5-39 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

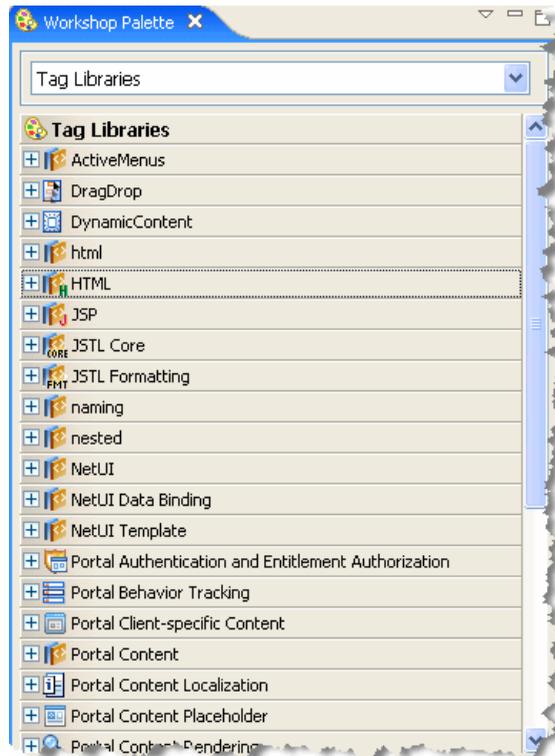
WebLogic Portal provides JSP tags that you can use within JSPs. You can view available JSP tags in the Design Palette and then drag them into the Source View of your JSP, and use the Properties view to edit elements of the code.

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*.

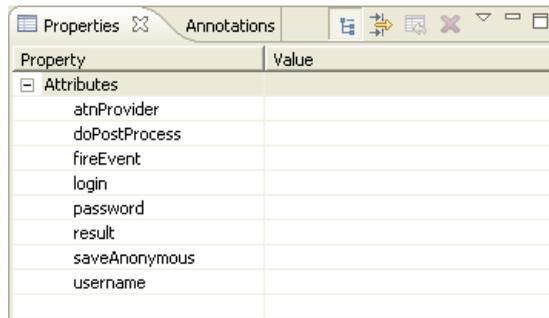
Viewing Available JSP Tags

When you open a JSP in Workshop for WebLogic, you can use the Design Palette to display all the JSP tags currently loaded and available; [Figure 5-40](#) shows a portion of the display.

Figure 5-40 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 5-41](#):

Figure 5-41 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:

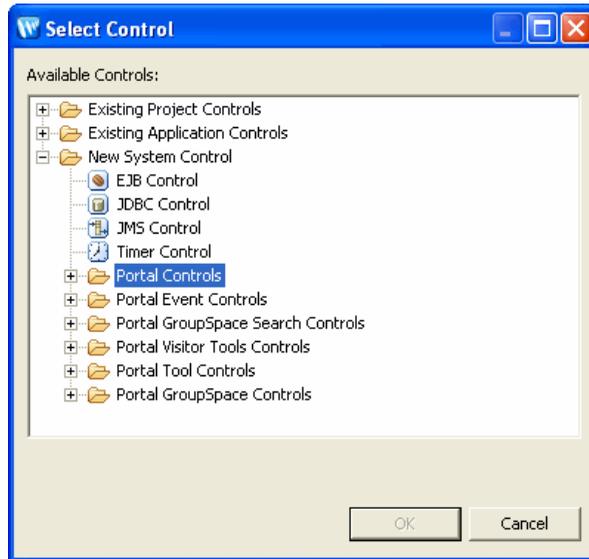
1. Open an existing page flow (. java file) or create a new page flow.
For information about creating page flows using Workshop for WebLogic, refer to the [Oracle Workshop for WebLogic User's Guide](#).
2. If you are not already using the Page Flow Perspective, Workshop for WebLogic asks if you want 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 5-42](#).

Figure 5-42 Insert > Control Menu Selection



The Select Control dialog box displays, as shown in [Figure 5-43](#).

Figure 5-43 Select Control Dialog



- 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, all 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 [Javadoc](#). (Links to the Javadoc for each of the controls packages are conveniently listed in the Javadoc Overview frame.)

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.

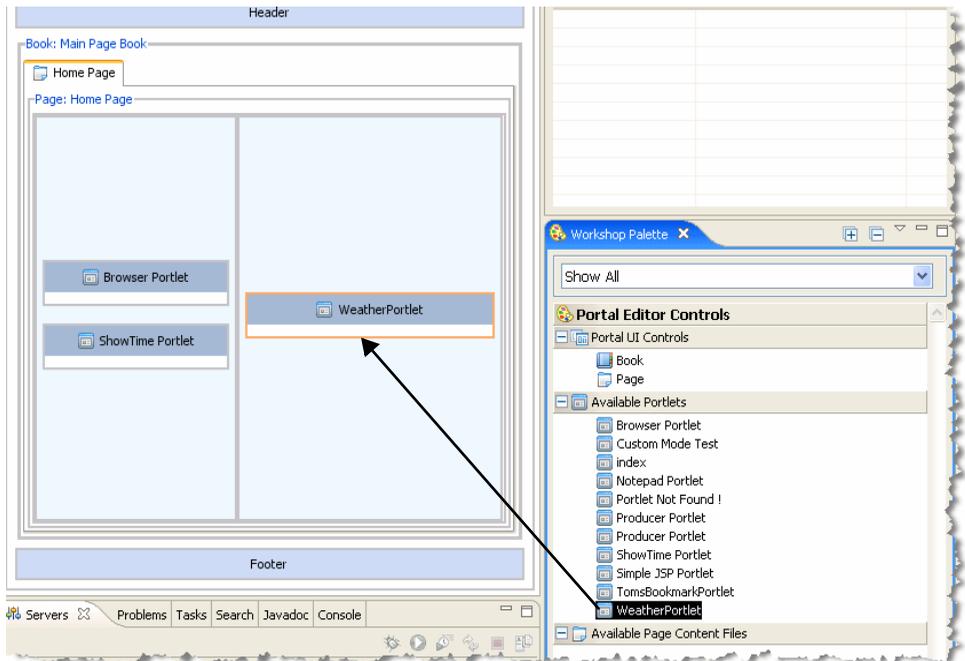
Follow these steps:

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

[Figure 5-44](#) shows an example of this step.

Figure 5-44 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 5-40](#).

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 13-5](#).

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 13-5](#).

Advanced Portlet Development with Tag Libraries

During the Development phase, you can use tag libraries to add features to a GroupSpace Community, a custom Community, or a portal web application. This section discusses the following tag libraries:

- The `ActiveMenus` JSP tag library
- The `DragDrop` JSP tag library
- The `DynamicContent` JSP tag library
- The `UserPicker` 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 5-109](#)). See [Figure 5-45](#) for an example of the `ActiveMenus` tag in a GroupSpace Community.

Figure 5-45 ActiveMenus in the GS Issue Portlet

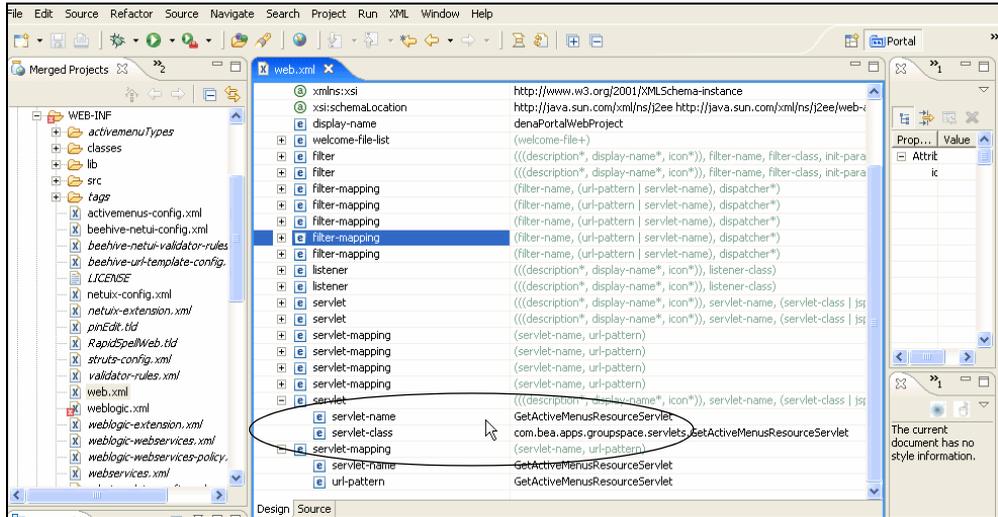
You can tie a user's capability to the ActiveMenu that you see when you hover your mouse over an item (an Issue, for example) and hover over the arrow that appears. In this example, if your assigned capabilities include the ability to delete items, you will see the **Delete** choice, as shown in [Figure 5-45](#).

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 ActiveMenu 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 5-46](#) to view the edited file in Workshop for WebLogic.

Figure 5-46 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 ActiveMenu Tag

To use the ActiveMenu tag, you must set up the `activemenu-config.xml` file (the XSD that defines this config file is located in the `activemenu_taglib.jar` file as `activemenu-config.xsd`). This `activemenu-config.xml` 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 `activemenu-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 activemenu-config.xml File

```
<typeInclude xhref="/WEB-INF/activemenuTypes/username.xml"/>
<type>
  <menuItem>
    <param name="linkId"/>
    <action action="editLink">
      <il8nNamebundleName="com.bea.apps.groupspace.links.
        LinksPopupMenu" key="edit.link"/>
    </action>
    
  </menuItem>
</type>
```

```
</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/
    activemenu-config/9.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.bea.com/servers/apps/groupspace/ui/
    activemenu-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 `ActiveMenu` 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 `ActiveMenu` 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 `ActiveMenu` 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 `ActiveMenu` 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 `ActiveMenu` tag: `<activemenu 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 `ActiveMenu` 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 `activemenu_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="background-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:activemenu display="Foo Link" type="link">
        <param name="linkId" value="{fooLink.id}"/>
        <param name="linkParent" value="{fooLink.parent}"/>
    </activeMenus:activemenu>
...

```

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 `activemenu-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 `il8nName` 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 `activemenu-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 `activemenu-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 [Listing 5-17](#) into your `web.xml` file

Listing 5-17 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}" >
    
    ${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">
    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.

Using the User Picker

During the Development phase, you can use the `UserPicker` tag library to add a form button to a JSP page in a GroupSpace Community, a custom Community, or a portal web application.

The `UserPicker:popupButton` tag provides the developer with the ability to add a form button to a JSP page which opens a popup window that displays a list of current users. You can select a user from this list. The name of the selected user is populated into a specified form field on the parent window.

Using the UserPicker Tags

This section describes the `UserPicker:popupButton` tag in a custom Community and how to use the following attributes:

- The `inputId` Tag – The id of the HTML form input element that is populated with the selected user's name. This tag is optional.
- The `inputTagId` Tag – The `tagId` of the netui-based form input element that is populated with the selected user's name. If the `inputId` tag is provided, the `inputTagId` tag is ignored. This tag is optional.
- The `buttonImage` Tag – The `src` path to the image for the popup button. This tag is required.

- The `atnProviderName` Tag – The Authentication Provider name. If an `atnProviderName` is supplied, there is no provider drop-down box in the popup window. If an `atnProviderName` is not supplied, the default provider is used. If you have configured multiple Authentication Providers, a drop-down box appears in the popup window to allow you to specify a provider. This tag is optional.

Tip: When the `UserPicker:popupButton` tag is used in a Community, the Community members are listed, rather than users.

Importing and Exporting Java Portlets

Workshop for WebLogic lets you import and export Java (JSR168) portlets. You can import Java portlets from a Web Archive (WAR), Java Archive (JAR), or ZIP file directly into your workspace. Workshop for WebLogic automatically creates `.portlet` files for all imported Java portlets, making them available for immediate use in your portal. You can also export Java portlets from your workspace to a supported archive file.

Note: Throughout this section, supported archive files refer to WAR, JAR, and ZIP files.

By default, Workshop for WebLogic imports and exports the `portlet.xml`, any Java class files required by the portlet, and any Java source files. Also, if any class or source Java files are found within a JAR or ZIP archive, that archive is also imported or exported. You can optionally specify additional files to be imported or exported. Once exported, a Java portlet contained in a supported archive file can be used with any compatible web server.

Tip: You can use the JSR168 Import utility to deploy a Java portlet contained in a supported archive file directly to a WebLogic Server instance. Once deployed with this utility, the portlets can be available to consumers through WSRP. For more information on this utility, see [“Using the JSR168 Import Utility” on page 5-131](#).

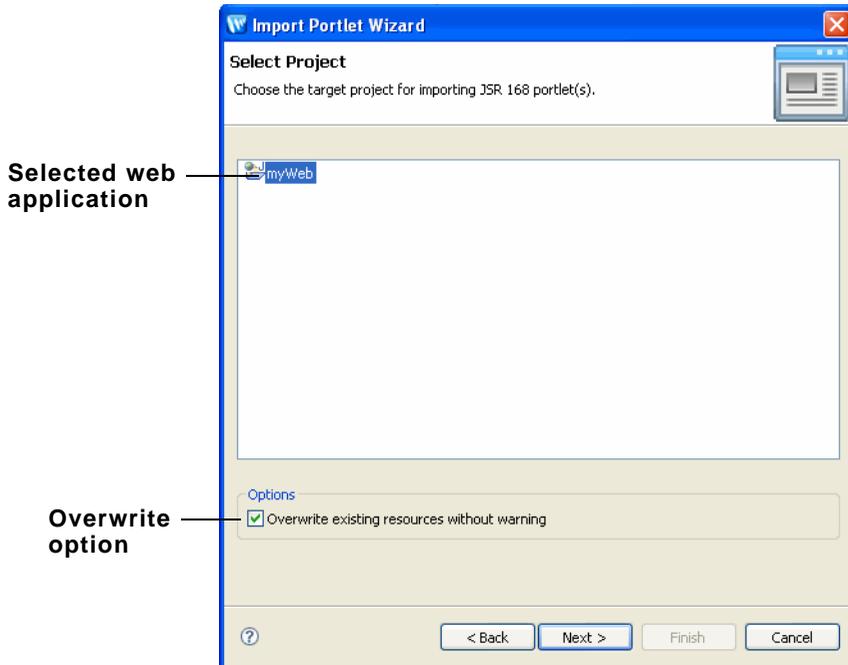
Importing Java Portlets

To import Java portlets packaged in a supported archive file into your Workshop for WebLogic workspace:

1. Select **File > Import**. You can also right-click in the Project Explorer and select **Import > Import...**
2. In the Import dialog, open the Other folder and select **Portlet(s) from Archive**.

3. In the Select Project dialog, select the web project in which to place the imported Java portlets. Select the **Overwrite existing resources without warning** checkbox to force the import tool to overwrite duplicate files automatically, as shown in [Figure 5-47](#).

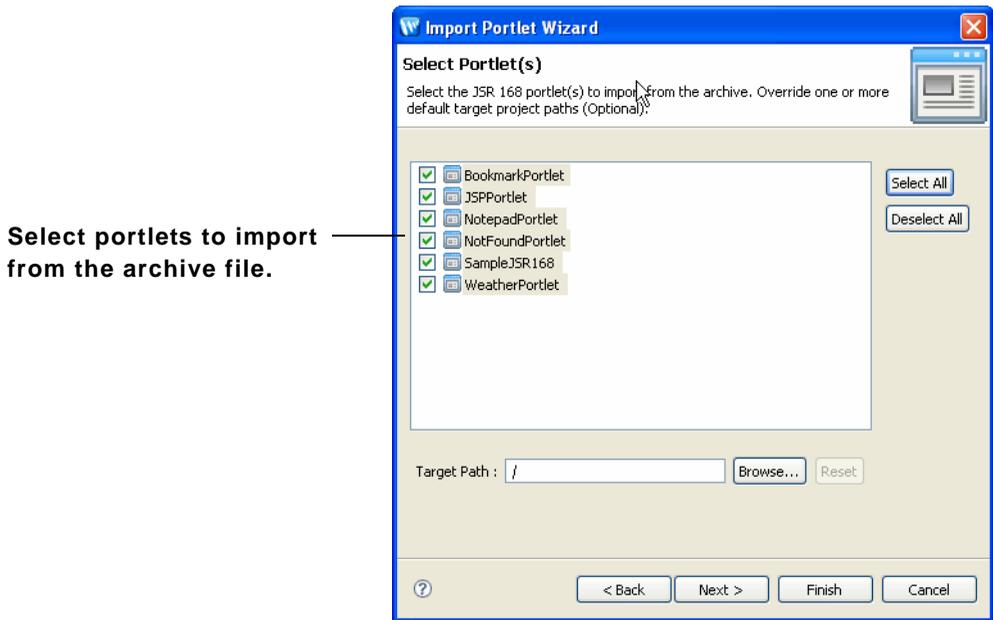
Figure 5-47 Select Project Dialog



4. In the Select Archive dialog, select from your system a supported archive file (WAR, JAR, or ZIP) containing Java portlets, and click **Next**. The Select Portlet(s) dialog appears, as shown in [Figure 5-48](#).

Note: If the selected archive does not contain any Java portlets, then no standard artifacts will be selected. This is because the default archive format plugin does not recognize the archive as a Java portlet archive. In this case, the wizard allows you to select an archive format and allows you to select files manually to import. This is not a typical use case.

Figure 5-48 Select Portlet(s) to Import Dialog



5. If you have more than one archive format plugin. If you have multiple plugins installed that recognize the archive file as a Java portlet archive, then the Select Format dialog appears. Use this dialog to pick the archive format plugin you wish to use.
6. In the Select Portlet(s) dialog, select the portlets to import. You must select at least one portlet. The Target Path specifies where, relative to the WebContent root, to place the portlet(s).

Tip: You can create a new folder for your imported portlets simply by typing the folder name in the Target Path field. For example, if you enter `/portlets`, then the folder `WebContent/portlets` will be created and the portlet(s) will be placed in that folder. You can assign a specific target path to one or more portlets simply by selecting the portlet(s) and entering a target path. The Reset button restores the original path. You can multi-select a group of portlets and assign a target path to the selected group.

7. Click **Next**.

- (Optional) If the supported archive file contains any optional files to that you want to import, select them and specify a target path relative to the WebContent root.

Tip: Optional files are any files in the supported archive file that were not specified as required in the Import Template. The default Import Template requires the archive to contain a `portlet.xml` file and any class files required by the portlet.

Tip: If you develop your own custom import plugin, it will also show up in the dropdown list of formats in the Select Format dialog. You can create a custom portlet import plugin if you want to specify the format of the imported archive and the default format is not sufficient. To create the plugin, you need to implement the `PortletImporterPlugin` interface. For more information, refer to the [Javadoc](#) on this interface.

- Click **Finish**. The portlet files are imported from the supported archive file, and `.portlet` files are created automatically in your workspace.

Tip: If you encounter problems with the import, it is possible that the a conflict exists with artifacts that already exist in your project workspace. In this event, use the **Back** button to change the target paths of conflicting artifacts, or select the wizard option on the Select Project dialog that forces the wizard to overwrite existing resources (see [Figure 5-47](#)).

Exporting Java Portlets

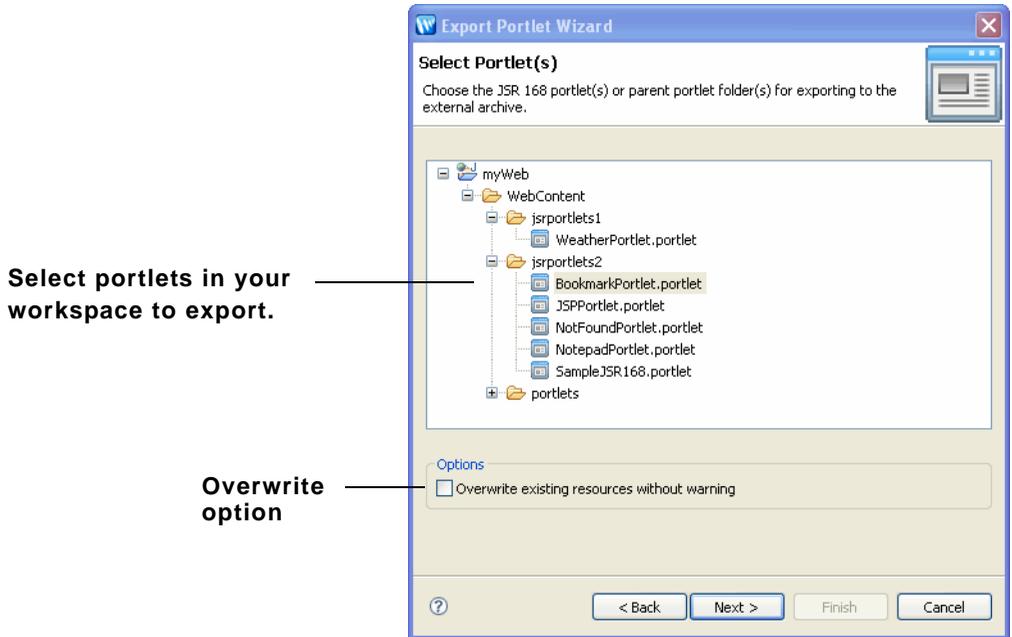
You can export Java portlets to a new or existing supported archive file (WAR, JAR, or ZIP). To export Java portlets to a supported archive file:

- Select **File > Export**. You can also right-click in the Project Explorer and select **Export > Export...**
- In the Export dialog, open the Other folder and select **Portlet(s) to Archive**.
- In the Select Portlets dialog, select the web project that contains the Java portlet(s) you want to export. You can select the parent folder that contains the portlet(s) or drill down to select individual portlets, as shown in [Figure 5-49](#). Any portlets and/or parent folders that were selected in the Project Explorer will be pre-selected by default.

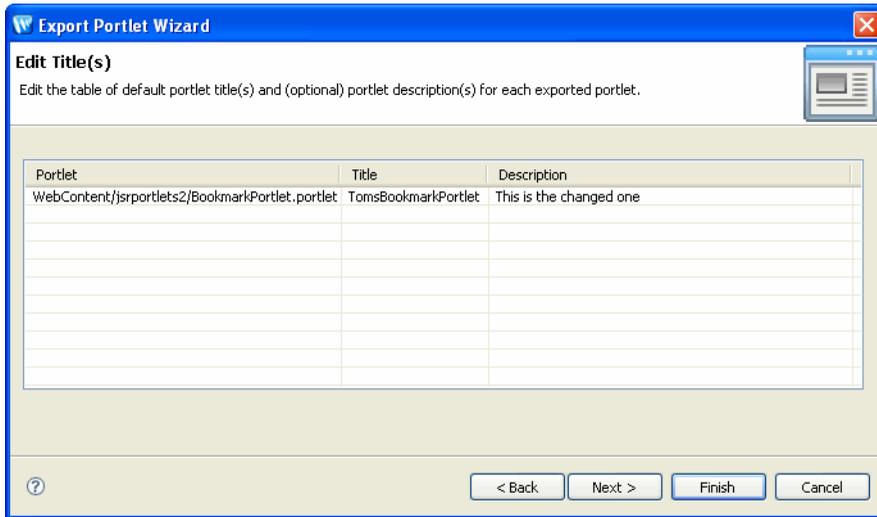
Select the **Overwrite existing resources without warning checkbox** to force the export tool to overwrite duplicate files automatically.

Note: All selected portlets must exist within the same Web project. You cannot select portlets for export across different Web projects.

Figure 5-49 Select Portlet(s) to Export Dialog



4. Click **Next**. The Edit Title(s) dialog appears, as shown in [Figure 5-50](#).

Figure 5-50 Edit Title(s) Dialog

- In the Edit Title(s) dialog, you can add or modify the Title and/or Description of an exported portlet. These fields are written to the exported portlet's `portlet.xml` file. Click **Next** to continue.

Note: A Title is required for each portlet. If any Title field is blank, the **Next** button is disabled until you supply a title.

- In the Select Archive dialog enter a full path and name for the archive file, or use the **Browse** button to specify the path, and click **Next**. If the archive does not exist, the wizard will prompt you to create it.
- In the Select Format dialog, pick the archive format that you want to use and click **Next**. A default format is provided.

Tip: If you develop your own custom export plugin, it will also show up in the dropdown list of formats in the Select Format dialog. You can create a custom portlet export plugin if you want to specify the format of the exported archive and the default format is not sufficient. To create the plugin, you need to implement the `PortletExporterPlugin` interface. For more information, refer to the [Javadoc](#) on this interface.

8. In the Select Files dialog, select any optional supporting files, such as JSPs, that you wish to include in the supported archive file. Any files that are included in the selected archive format (such as `portlet.xml`) are automatically selected in the dialog. You can associate a Target Path path with any selected files. Those files will be placed in the specified target path within the archive file. By default, all files are stored relative to the root directory of the archive.
9. Click **Finish**. The archive file is created in the location you specified.

Using the JSR168 Import Utility

WebLogic Portal provides a utility for automatically deploying JSR-168 portlets that are packaged in JSR-168 WAR files. This utility lets you import JSR-168 WAR files containing JSR-168 portlets, and expose the portlets in WSRP producers. For detailed information on this utility, see the chapter “Deploying Portal Applications” in the *Production Operations Guide*.

Building Portlets

Creating Clipper Portlets

A clipper portlet is a portlet that renders content from another web site. A clipper portlet can include all or a subset of another web site's content using a process called "web clipping." This chapter explains how to create and configure clipper portlets.

This chapter includes these topics:

- [Introduction](#)
- [Creating a Clipper Portlet](#)
- [Modifying Clipper Portlet Properties](#)
- [Modifying the Appearance of a Clipper Portlet](#)
- [Authenticating a Clipper Portlet](#)
- [Configuring URL Rewriting](#)
- [Clipper Portlets and HTTPS](#)
- [Certificates and WebLogic Server](#)
- [Resetting the Clipper Portlet](#)
- [Using Backing Files with Clipper Portlets](#)
- [Updating Portlet Preferences While the Server is Running](#)
- [Clipper Portlet Limitations](#)

Introduction

Clipping is an easy technique for including content in your portal. You can clip all or part of another web site. Users can effectively view and interact with content from another web site without leaving the portal.

Note that another WLP feature, the browser portlet, also lets you include remote web page contents in a portal. For information on browser portlets, see [“Browser Portlets” on page 5-28](#). A clipper portlet differs from a browser portlet in the following ways:

- A browser portlet uses an IFrame, while a clipper portlet includes the content of the clipped web page into the same page as the rest of the portal. An advantage of using an IFrame is that it isolates the remote content, preventing it from overlapping other parts of the portal. Disadvantages are that the portal cannot access the IFrame’s content and portal and IFrame sessions are maintained separately.
- A browser portlet returns the entire remote page, while a clipper portlet lets you subset or modify the contents of a remote web page.

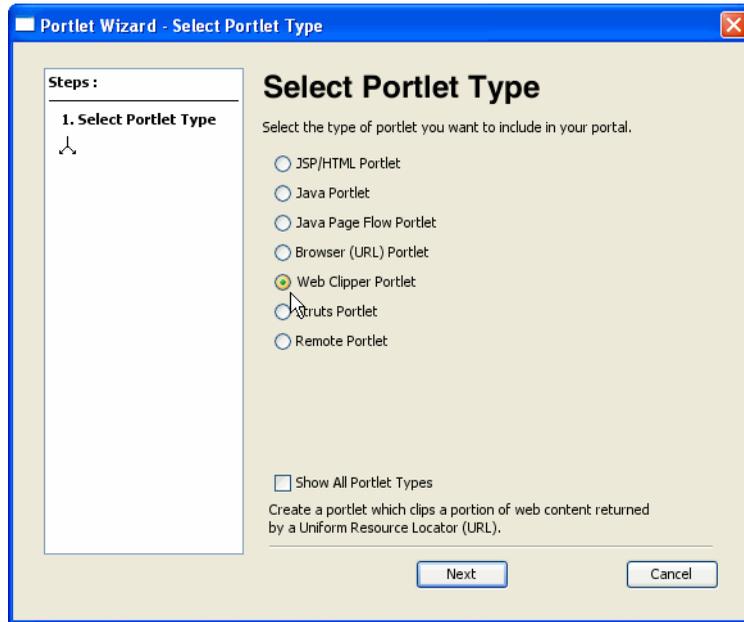
Creating a Clipper Portlet

You create a clipper portlet using the Portlet Wizard. The steps are similar to those of creating other types of portlets.

Note: No post-processing is performed on the text of a clipped web page, unless a `clipCustomClass` preference is specified as described in [“Modifying the Appearance of a Clipper Portlet” on page 6-6](#). Clipped text is written verbatim to the response. If the original web page contains syntax errors, the errors may also appear in the consumer browser when the clipper portlet is rendered.

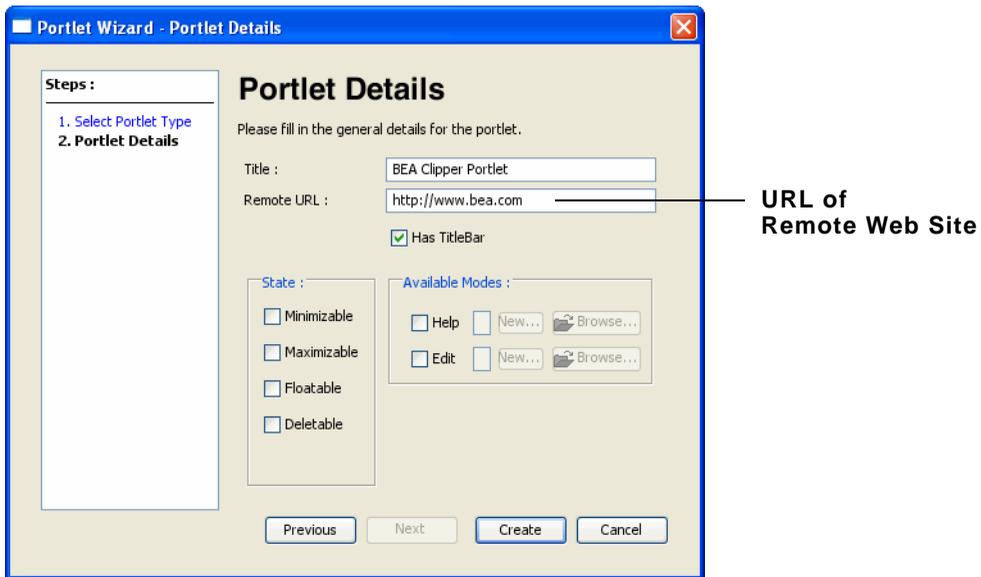
1. If it is not currently open, open the Portal Perspective.
2. Select File > New > Portlet.
3. In the New Portlet dialog, enter a name for the portlet, and click Finish. The Portlet Wizard opens.
4. In the Portlet Wizard, select Web Clipper Portlet, as shown in [Figure 6-1](#) and click Next.

Figure 6-1 Selecting Web Clipper Portlet



5. In the Portlet Details dialog, enter a title for the portlet and enter the URL of the web site you want to clip in the Remote URL field, as shown in [Figure 6-2](#).

Figure 6-2 Specifying the URL of a Remote Web Site



6. Click Create to create the new clipper portlet.
7. Modify the clipper portlet, if you want, by adding and editing preferences, as explained in [“Modifying Clipper Portlet Properties”](#) on page 6-4.

Note: By default, the entire web site is included in the clipper portlet’s contents, including the <HEAD> element of the remote site.

Modifying Clipper Portlet Properties

By setting certain portlet properties, you can change the appearance of a clipper portlet, subset the content of a web page that appears in a clipper portlet, and provide authentication. There are two primary ways to modify a clipper portlet’s properties: through the Properties editor and manually.

This section includes these topics:

- [Using the Properties Editor](#)
- [Setting Clipper Properties Manually as Preferences](#)

Using the Properties Editor

You can use the Properties editor to edit the common set of portlet properties, such as the title bar and presentation properties. Clipper portlets share most of these properties with other types of portlets, and the procedure for changing them is the same. See [“Portlet Properties” on page 5-40](#) for detailed information on editing portlet properties through the Properties editor.

Setting Clipper Properties Manually as Preferences

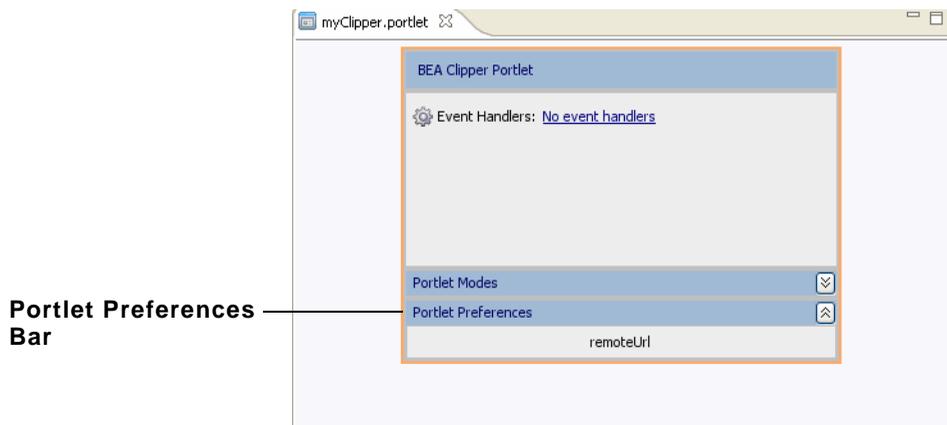
Clipper portlets also include a set of properties that do not appear in the Properties editor and which must be set manually. The easiest way to modify clipper portlet properties manually is to add and set them as portlet preferences.

Tip: WLP provides preferences for controlling the extent of a clipped page and for authentication. See [“Modifying the Appearance of a Clipper Portlet” on page 6-6](#) and [“Authenticating a Clipper Portlet” on page 6-8](#) for specific information on those tasks.

To set portlet preferences, do the following:

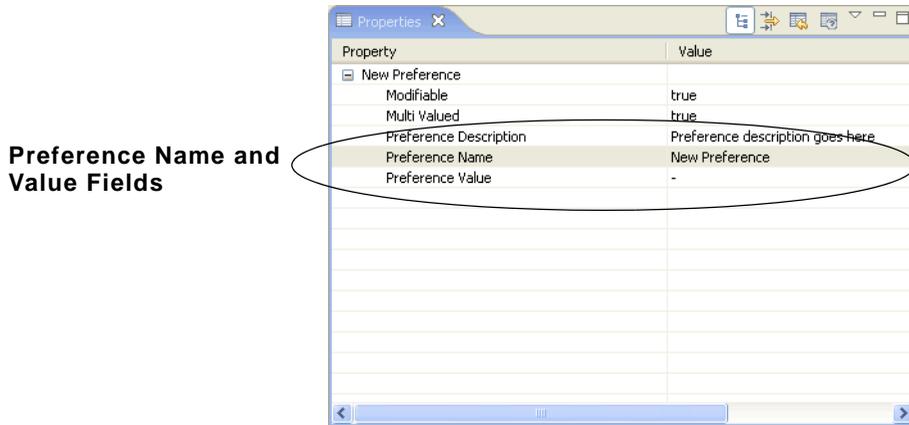
1. Open the Portlet editor for the clipper portlet. To do this, right click the portlet name in the Project Explorer and select Open With > Portlet Editor.
2. Right click the Portlet Preferences bar in the portlet editor and select Add Preference. The Portlet Preferences bar is shown in [Figure 6-3](#).

Figure 6-3 Portlet Preferences Bar



3. In the Properties editor, enter the Preference Name and Preference Value in the appropriate fields, as shown in [Figure 6-4](#).

Figure 6-4 Preference Name and Value Fields



For more information on setting portlet preferences, see [“Portlet Preferences”](#) on page 5-56

Modifying the Appearance of a Clipper Portlet

You can set portlet preferences to specify which portion of a web page to clip. You can also modify the text that is clipped by implementing a Java class and specifying it as a portlet preference. [Table 6-1](#) lists and describes the set of clipper portlet preferences that you may set manually to accomplish these tasks. For details on how to set preferences, see [“Modifying Clipper Portlet Properties”](#) on page 6-4.

Note: The preferences clipXPath, clipStartText/clipEndText, and clipCustomClass (listed in [Table 6-1](#)) are exclusive. The system looks for clipCustomClass first. If that class is not present, the system looks for clipXPath. If clipXPath is not present, the system looks for clipStartText/clipEndText.

Table 6-1 Preferences for Determining the Text to Clip

Property Name	Property Value
clipXPath	<p>(Optional) An XPath that is applied to the remote page. The remote page is required to be well-formed XML. If you set this option, the system will apply the XPath to the remote page and put the text of the first node found in the clipper portlet output.</p> <p>This option provides a convenient way to clip a specific chunk of text. For example, suppose this preference value is <code>html/body[1]</code>, which is an XPath expression that selects the <code><body></code> element of the web page's text.</p> <p>You can also use this option to specify div elements to clip. For example, <code>//div[@id="barracuda"]</code> clips out a <code><div id="barracuda"></code> element.</p>
clipStartText, clipEndText	<p>(Optional) Specifying regular expressions that are used to locate the beginning and the end of the web page text to clip. For example, if the page you want to clip looks like:</p> <p>Some web site text... <code><abc></code> text to clip <code></abc></code> Some more web site text...</p> <p>and you want to clip the text between <code><abc></code> and <code></abc></code> inclusive, enter the following properties and values:</p> <p><code>clipStartText = <abc></code> <code>clipEndText = </abc></code></p> <p>The left angle bracket needs to be escaped if you enter the values directly in the XML .portlet file. For example:</p> <pre><netuix:preference name="clipStartText" value="&lt;abc;" modifiable="false"/> <netuix:preference name="clipEndText" value="&lt;/abc;" modifiable="false"/></pre>
clipCustomClass	<p>(Optional) This preference specifies the name of a class that implements <code>com.bea.netuix.clipper.IClipStrategy</code>. This interface lets you define your own clipping logic. The interface has one method to implement:</p> <pre>String clip(String markup);</pre> <p>Your implementation must have a no-argument constructor. The <code>clipCustomClass</code> preference registers your implementation with the portlet.</p> <p>An <code>IClipStrategy</code> class can be used to selectively rewrite a web page; for example, you can substitute text in the page, or suppress certain elements.</p>

Authenticating a Clipper Portlet

This section explains how to configure authentication for a clipper portlet. Once configured, clipper portlet authentication is automatic. WebLogic Portal supports two kinds of clipper portlet authentication:

- Form-based authentication
- Basic HTTP authentication

Both of these methods are described in this section.

Form-Based Authentication

Form-based authentication is performed through a server-side form request on the remote site. You configure this type of authentication by setting preferences on the portlet. The procedure for setting preferences is described in [“Modifying Clipper Portlet Properties” on page 6-4](#).

Note: There are current security limitations associated with form-based authentication. See [“Clipper Portlet Limitations” on page 6-16](#).

To set up form-based authentication:

1. Set the preference: authenticationType = Form. This preference enables form-based authentication.
2. Tell the server how to build up the HTTP request that performs the authentication by setting the preferences listed in [Table 6-2](#).
3. Provide the authentication credentials by setting the preferences listed in [Table 6-3](#).

Table 6-2 HTTP Request Preferences

Preference Name	Preference Value
loginFormUrl	(Required) The ACTION attribute in the HTML <FORM> element. This is the URL to which the authentication request is made.
loginFormMethod	(Required) The METHOD attribute in the HTML <FORM> element. The value must be either GET or POST.
loginFormUserParam	(Required) The name of the request parameter that holds the login name.

Table 6-2 HTTP Request Preferences

Preference Name	Preference Value
loginFormPasswordParam	(Required) The name of the request parameter that holds the login password.
loginFormExtraParams	(Optional) A string to append to the request query. Use this string to specify custom parameters that might need to be set. For example, if the form also has COLOR and SHAPE parameters, you can set them with: loginFormExtraParams= COLOR=PURPLE&SHAPE=DIAMOND

The following preferences are used to provide the authentication credentials.

Table 6-3 Authentication Credential Preferences

Preference Name	Preference Value
groupUsername	Specifies shared user name.
groupPassword	Specifies the password for the shared user name.
personalUsername	Specifies a username on a per-user basis. Ignored if groupUsername is set.
personalPassword	Specifies the password on a per-user basis. Ignored if groupUsername is set.

[Listing 6-1](#) shows example preferences for form-based authentication.

Listing 6-1 Example Form-Based Authentication Preferences

```
<netuix:preference name="remoteUrl" value="http://some.site.com"
modifiable="false"/>
<netuix:preference name="loginFormUrl" value="http://some.site.com/login.action"
modifiable="false"/>
<netuix:preference name="authenticationType" value="Form" modifiable="false"/>
<netuix:preference name="loginFormMethod" value="POST" modifiable="false"/>
```

```
<netuix:preference name="loginFormUserParam" value="os_username"
modifiable="false" />

<netuix:preference name="loginFormPasswordParam" value="os_password"
modifiable="false" />

<netuix:preference name="loginFormExtraParams" value="os_destination=abc"
modifiable="false" />

<netuix:preference name="groupUsername" value="your_username"
modifiable="false" />

<netuix:preference name="groupPassword" value="your_password"
modifiable="false" />
```

Basic HTTP Authentication

To set up basic HTTP authentication:

1. Set the preference: `authenticationType = BasicHTTP`. This preference enables form-based authentication.
2. Provide the authentication credentials by setting the preferences listed in [Table 6-3](#).

Table 6-4 Authentication Credential Preferences

Preference Name	Preference Value
<code>groupUsername</code>	Specifies shared user name.
<code>groupPassword</code>	Specifies the password for the shared user name.
<code>personalUsername</code>	Specifies a username on a per-user basis. Ignored if <code>groupUsername</code> is set.
<code>personalPassword</code>	Specifies the password on a per-user basis. Ignored if <code>groupUsername</code> is set.

[Listing 6-2](#) shows example basic HTTP authentication preferences.

Listing 6-2 Example Basic HTTP Authentication Preferences

```
<netuix:preference name="authenticationType" value="BasicHTTP" modifiable="false"/>
<netuix:preference name="groupUsername" value="your_username" modifiable="false"/>
<netuix:preference name="groupPassword" value="your_password" modifiable="false"/>
```

Configuring URL Rewriting

This section explains how to configure the way clipper portlets rewrite navigable links and resource URLs.

This section includes these topics:

- [Navigable Link Configurations](#)
- [Resource URL Configurations](#)
- [URL Rewriting Configuration Techniques](#)

Navigable Link Configurations

Navigable links, such as anchor links, can be configured as follows:

- Rewrite the link so that the resulting page displays in the portlet. This is the default.
- Do not rewrite the link. In this case, if you click the link, the linked page opens in another browser window, not in the portlet.
- Block the link. Because a clipper portlet embeds the URL that defines a page to clip in the portal request, it is possible to manually change the URL so the portlet clips an arbitrary web page. This presents a security risk, because a user could browse web pages from the WLP server, which may be behind a firewall and thus allow access to pages that aren't authorized for the given user. Clipper portlets can be configured to block this security risk.

See [“URL Rewriting Configuration Techniques” on page 6-12](#) for more information.

Resource URL Configurations

Resource URLs point to images, stylesheets, scripts, and so on. You can configure a clipper portlet so that it either does or does not rewrite resource links so that they are proxied through the WLP server.

By default, resources are proxied, because cookies for clipped pages are stored on the WLP server. For example, if you clip a page behind a firewall, your browser will not have access to resources on the remote page. In this case, it is necessary to route resource requests through the WLP server. However, this proxying can affect WLP server performance; therefore, you have the option to turn proxying off if you don't need it.

See [“URL Rewriting Configuration Techniques” on page 6-12](#) for more information.

URL Rewriting Configuration Techniques

You can configure the way URLs are rewritten by implementing a Java class called `IClipperUrlFilter` or by setting portlet preferences.

Implementing `IClipperUrlFilter`

The SPI interface `com.bea.netuix.clipper.IClipperUrlFilter` is available for you to define your link rewriting rules. This interface has three methods, listed in [Listing 6-3](#). Your implementation must have a no-argument constructor. You register your implementation with the portlet by using the `urlFilter` portlet preference. For example:

```
<netuix:preference name="urlFilter" value="my.package.MyUrlFilterImpl"
modifiable="false"/>
```

For more information on setting portlet preferences, see [“Setting Clipper Properties Manually as Preferences” on page 6-5](#) and see [Portlet Preferences](#) in the *Portlet Development Guide*.

Listing 6-3 `IClipperUrlFilter` Methods

```
/** Should the url be reachable from the clipper portlet?
 * If this method returns false, rewritten links containing this url will have
 * empty values (for example, a link <a href="forbidden.site.com">
 * would be rewritten to <a href="" >, and a request to clip this url would
 * receive a 404 response.
 */
boolean allowUrl(String url);

/**
 * Should the url be rewritten to stay within portal context? If this methods
 * returns false, clicking on a link
 * to this url will take the user straight to the target url, which will render
 * the new page in the full browser, and not inside the clipper portlet.
```

```

*
* This method applies to navigable urls only: links in anchors, form actions,
* etc.
*/
boolean rewriteClickableUrl(String url);

/**
* For resource urls only, e.g. image, script, and style tags.
*
* Should the resource be proxied through the wlp server, or should the resource
* link point
* directly to the original resource in the remote page?
*/
boolean rewriteResourceUrl(String url);

```

Using Portlet Preferences

If you don't want to define your own class to control link rewriting, you can use these portlet preferences. For more information on setting portlet preferences, see [“Setting Clipper Properties Manually as Preferences” on page 6-5](#) and see [Portlet Preferences](#) in the *Portlet Development Guide*.

- **allowedUrlRegex** – Set the value of this portlet preference to a regular expression. WLP tries to match URLs against this expression, and if the match fails, the link will be blocked. For example:

```
<netuix:preference name="allowedUrlRegex" value=".*allowedlink.*"
modifiable="false"/>
```

- **proxyResourceUrls** – If this portlet preference is set to false, resource URLs will not be rewritten to run through the WLP server. The default value is true. The following example turns off rewriting for resource links:

```
<netuix:preference name="proxyResourceUrls" value="false*"
modifiable="false"/>
```

Suppose the clipped page has an image tag ``. If the `proxyResourceUrls` preference value is false, then the clipper page will have the exact same image link:

```

```

But if the value is set to true, the link will look like this:

```

```

Clipper Portlets and HTTPS

This section discusses how to handle clipper portlets with HTTPS URLs.

When an HTTPS link is clipped, the link shows up as an HTTPS link in the portal page.

If you click on a clipped link that causes a redirect on the remote site from an HTTP URL to an HTTPS URL, the portal request is redirected from an HTTP URL to an HTTPS URL, as expected. However, note the following exception to this case: the initial request to the portal for a given browser session is never redirected to HTTPS. The following cases illustrate this exception:

The page `www.xyz.com` has a link to `xyz.com/mail`. This link points to `http://mail.google.com/mail`, and clicking that link redirects you to `https://www.google.com/accounts/ServiceLogin`.

If you clip `http://www.xyz.com` into your portal at `http://myportal.com`, start your browser, and click the mail link in your clipped portal, the portal request will be redirected to `https://myportal.com`. The clipped page will be redirected to, for example, `https://www.xyz.com/accounts/ServiceLogin`, and you will see that page.

If you clip `http://mail.xyz.com/mail`, and you start your browser and open the portal, then you will not be redirected to HTTPS. The clipped page will still follow the redirect to, for example, `https://www.zyz.com/accounts/ServiceLogin`, so the page contents will look fine, but the route from the browser to the WLP server will not use HTTPS. Note that the “Sign In” form on that page has an HTTPS action URL, so the action for the clipped form will point to `https://myportal.com`.

Likewise, if you clip `https://www.xyz.com/accounts/ServiceLogin`, and you start your browser and go to `http://myportal.com`, then you will not be redirected to HTTPS on that initial request.

Certificates and WebLogic Server

For WLS to make an HTTPS request to a site, it must have a certificate for that site in its keystore. If the certificate is not available, you will see exceptions such as:

```
[Security:090477]Certificate chain received from aaa.bbb.com - 10.123.45.67  
was not trusted causing SSL handshake failure.
```

For detailed information on configuring SSL in WLS, see the WLS document [Configuring Identity and Trust](#) on e-docs. The basic steps to configure a clipper portlet to use HTTPS correctly are:

1. Obtain a security certificate for the site you are trying to clip.

Tip: One way to obtain the certificate is to use the Firefox plugin called “Cert Viewer Plus.” This plugin lets you view and save the security certificate.

2. Open a command shell and navigate to the root directory of your domain.
3. Locate the trust keystore; for example, DemoTrust.jks.
4. Obtain the password for the keystore.
5. Use the Java keytool program to import the key. For example:

```
keytool -import -file my_certificate_file -keystore DemoTrust.jks -alias some_unique_alias
```

The alias value is an alias unique to that .jks file. You can view the aliases with the command:

```
keytool -list -keystore DemoTrust.jks
```

Resetting the Clipper Portlet

If you click on links in a clipped web page, the only way to restore the original clipped URL is to restart your browser.

Using Backing Files with Clipper Portlets

The clipper portlet comes with its own backing file. For detailed information on backing files, see [Backing Files](#) in the *Portlet Development Guide*.

To add your own backing file to a clipper portlet, do the following:

- Create your own backing class that extends `com.bea.netuix.clipper.ClipperBacking`.
- For any of the backing file methods you override, call the super class on that method.
- Set the `backingFile` attribute in your `.portlet` file to your backing class.

Updating Portlet Preferences While the Server is Running

If you change the preferences for a clipper portlet in the `.portlet` file, the changes are not picked up at runtime unless you set the following attribute in the `WEB-INF/netuix-config.xml` file:

```
<propagate-preferences-on-deploy propagate-to-instances='true'  
master='file' />
```

Preference changes that are made in the Administration Console are picked up automatically.

Clipper Portlet Limitations

The following are known limitations on the clipper portlet feature. These limitations may or may not apply to future releases.

- Authentication preferences are not encrypted.
- JavaScript in remote pages is not fully supported. Sites with that use JavaScript may work, but it is not guaranteed.
- Persistent cookies are not supported. Remote cookies only last as long as the main portal session.
- The current remote URL for a clipper portlet is stored in the session. This means that to reset your clipper window, you need to close and restart your browser.

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](#)
- [Portlet Caching](#)
- [Remote Portlets](#)
- [Portlet Forking](#)
- [Asynchronous Portlet Content Rendering](#)

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 5-40](#) 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 article [Portlet Caching](#) by Gerald Nunn, available the Oracle Technology Network.

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](#)
- [Architectural Details of Forked Portlets](#)
- [Best Practices for Developing Forked Portlets](#)

Configuring Portlets for Forking

Forking is easy to enable – you just set properties using the portlet Properties editor in Workshop for WebLogic, as shown in [Figure 7-1](#). The available forking properties are described in this section. For detailed information on the Portlet Properties editor, see “[Portlet Properties](#)” on [page 5-40](#).

Figure 7-1 Forking Properties

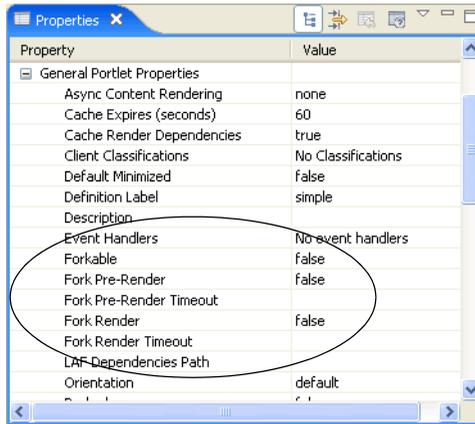


Table 7-1 Portlet Forking Properties

Property	Value
Forkable	<p>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 7-10 for tips on developing forked portlets.</p> <p>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.</p>
Fork Pre-Render	<p>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 7-6. See also “How the Control Tree Affects Performance” in the Portal Development Guide for more information about the control tree life cycle.</p> <p>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 7-9 for more information on the pre-render phase.</p>

Table 7-1 Portlet Forking Properties (Continued)

Property	Value
Fork Pre-Render Timeout (seconds)	<p>If Fork Pre-Render is set to <code>true</code>, 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.</p> <pre><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.></pre>
Fork Render	<p>Setting to <code>true</code> tells the framework that it should attempt to multi-thread render the portlet. This property can be set to <code>true</code> only if the <code>Forkable</code> property is set to <code>true</code>. See “Dispatching Render Forked Portlets to Threads” on page 7-9 for more information on the render phase.</p>
Fork Render Timeout (seconds)	<p>If Fork Render is set to <code>true</code>, 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.</p> <p>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.</p>

The forking properties, if set, appear as XML elements a `.portlet` file. [Listing 7-9](#) shows a sample of a portlet configured for both pre-render and render forking:

Listing 7-9 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 7-10](#).

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 7-2](#) illustrates the general structure of a portal control tree.

Figure 7-2 Simple Portal Schematic Example

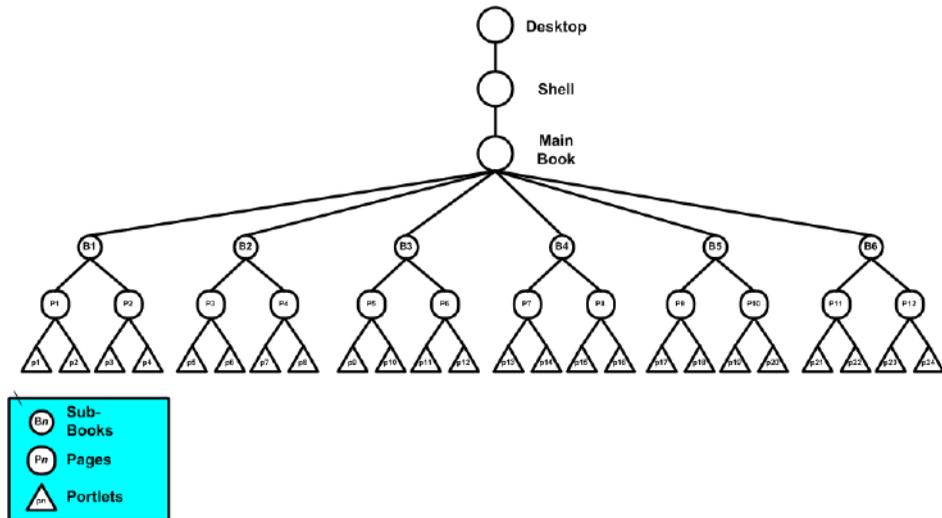
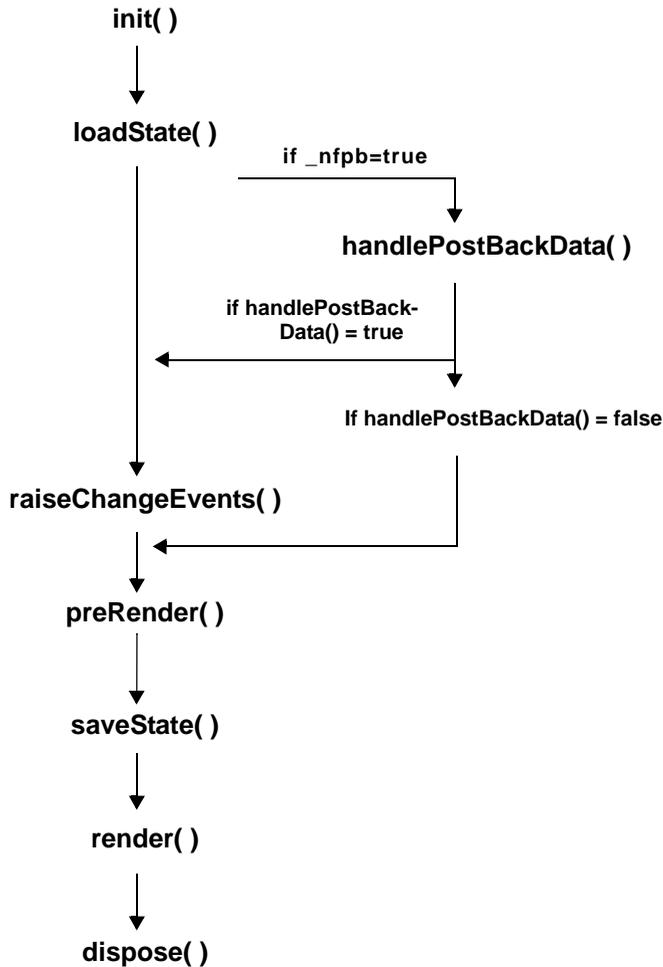


Figure 7-3 illustrates the successive phases of the portal rendering life cycle. During the first traversal of the control tree, the `init()` method is called on each control. On the second traversal, `loadState()` is called, and so on, until every control is processed.

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 7-3 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 7-3](#) 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 7-9](#).

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 7-11](#).

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 7-13](#). For a comparison of portlet forking and asynchronous rendering, see [“Comparison of Asynchronous and Conventional or Forked Rendering” on page 7-18](#).

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.

Note: The Collaboration portlets, such as the Calendar portlet, will not operate correctly when the desktop or portlet asynchronous mode is enabled. Async mode is not supported for Collaboration portlets. For information on Collaboration Portlets, see [“Using the Collaboration Portlets”](#) on page 11-1.

Tip: You can also enable asynchronous rendering for an entire portal desktop. Both portlet-specific (as discussed in this section) and desktop asynchronous rendering offer quicker response times than synchronous rendering. Note that the portlet-specific and desktop options for asynchronous rendering are mutually exclusive features. For more information on asynchronous desktop rendering and tips on deciding which method to choose, see the chapter “Designing Portals for Optimal Performance” in the *WebLogic Portal Development Guide*.

You can use either AJAX technology or IFRAME technology to implement asynchronous rendering for individual portlets. 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](#)
- [Considerations for IFRAME-based Asynchronous Rendering](#)
- [Considerations for AJAX-based Asynchronous Rendering](#)
- [Comparison of IFRAME- and AJAX-based Asynchronous Rendering](#)
- [Comparison of Asynchronous and Conventional or Forked Rendering](#)
- [Asynchronous Content Rendering and IPC](#)

Implementing Asynchronous Portlet Content Rendering

The 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 drop-down 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.

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 7-20](#). If you require interportlet communication, consider using asynchronous desktop rendering, as described in the chapter “[Designing Portals for Optimal Performance](#)” in the [Portal Development Guide](#).
- 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 7-20](#).
- 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 7-20](#), 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. See [“Portlet Forking” on page 7-3](#).

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.
- 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 7-20](#); 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 7-2](#) summarizes the advantages and disadvantages of IFRAME- and AJAX-based asynchronous rendering. Oracle recommends AJAX as a more robust implementation.

Table 7-2 IFRAME-based and AJAX-based Asynchronous Portlet Summary Table

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

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 7-3 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.	<p>Requires $n + 1$ requests (where n is the number of asynchronous portlets)</p> <p>True only for page requests; when interacting with an individual portlet, only one request is required.</p>

Table 7-3 Comparison of Behaviors - Forked/Conventional Rendering and Asynchronous Rendering

Behavior/Feature	Forked or Conventional Rendering	Asynchronous Rendering
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.

Tip: If you enable asynchronous rendering at the portal/desktop level, you can use IPC without restrictions. For more information on asynchronous portal/desktop rendering, see the [WebLogic Portal Development Guide](#).

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, use the `<render:context>` tag as follows. You can find this tag in the Design Palette under Tag Libraries > Portal Framework Rendering > Context.

```
<render:context asyncContentDisabled="true">
```

```

    Form, anchor, etc. would appear here
    (that is, <netui:form action="submit"...) 
</render:context>

```

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](#).

Optimizing Portlet Performance

Monitoring and Determining Portlet Performance

Oracle WebLogic Portal supports the collection of analytics. You can use these analytics to deliver information to other products such as Oracle WebCenter Analytics or they can be consumed locally via custom code. These analytics are exposed through a public WebLogic Portal API in the `com.bea.netuix.servlets.controls.analytics` package. For more information, see the WebLogic Portal [Javadoc](#).

This chapter focusses on deriving and using these metrics for uses such as monitoring Service Level Agreements (SLAs), triggering a warning if a specific portlet's response time is excessive, and replacing a misbehaving portlet with an alternate portlet.

This appendix contains the following sections:

- [Introduction](#)
- [Use Case](#)
- [Detecting a Misbehaving Portlet](#)
- [Disabling the Bad Portlet and Enabling an Alternative Portlet](#)

Introduction

Portals aggregate content and applications. Portals encapsulate these applications or content into subcomponents called portlets. Portlets are then brought together into a unified view that can be managed in one place. What happens when one of these portlets misbehaves or becomes unavailable? WebLogic Portal has many options for dealing with these scenarios. A few common ones are Web Service for Remote Portlets (WSRP) timeouts, interceptors, caching, threading,

and AJAX enabled portlets. Each of these solutions deals with the problem in a different way and each has their own set of advantages and disadvantages. Using the public API `com.bea.netuix.servlets.controls.analytics` package, you can provide a more comprehensive way of dealing with this problem.

Use Case

Supposed that you have a portal that brings together different applications (portlets) into a single portal. Each of these applications (portlets) is mission critical in their own right and if one goes down it is mandatory that it not disrupt the other applications. Additionally, if one application goes down, an alternate (backup) application must be displayed in its place. After the original application resumes normal service levels, it should be brought back online to replace the temporary application.

The first challenge is determining when an a portlet is not meeting a particular Service Level Agreement (SLA). The second challenge is providing the ability to bring certain portlets online or offline based on logic applicable the SLA.

Note: The other previously mentioned methods for dealing with these types of issues can be used in conjunction with the method described here. For simplicity, this example does not include any other mechanisms.

The solution to portlet failure and replacement uses two features available in WebLogic Portal. The first feature deals with detecting when a portlet is not meeting a particular SLA. The second feature deals with disabling the poorly behaving portlet and enabling a temporary portlet in its place. Each of these features are useful by themselves but when combined provide a more comprehensive solution to the problem.

Detecting a Misbehaving Portlet

To capture analytic events, you use the same hook that Oracle WebCenter Analytics uses to capture all of its events. However, rather than using that hook to provide the rich set of reporting capabilities provided by Oracle WebCenter Analytics, you can use the hook to simply determine when a portlet is misbehaving.

The first step is to implement the `AnalyticEventHandler` interface. One ore more implementations of this interface are called by the server whenever a portlet or page completes its processing. This means any implementation must be extremely efficient, as it may get called 50 or more times per request.

Listing 8-1 Portlet Rendering Time Detection

```

public class MyAnalyticEventHandlerImpl implements AnalyticEventHandler
{
    private final static long SLA = 5000000000L; // Five seconds (nano time)

    /**
     * <p>Implementation class may perform any one-time initializations here. If
     * this method fails (throws an exception) the event handler will not be
     * registered and no event handling will take place.</p>
     */
    public void init() {
        System.out.println("My Analytic Event Handler Initialized");
    }

    /**
     * <p>This method is called by the container at the end of each page's and
     * portlet's run. It is invoked for every page and portlet on every request.
     * Since this method is called so frequently the implementation must be
     * extremely efficient, or the entire portal's performance will suffer.</p>
     * @param analyticEvent the event to be logged.
     */
    public void log(AnalyticEvent analyticEvent)
    {
        // Ignore all but portlet events.
        if (analyticEvent.getAnalyticEventObject().equals
            (AnalyticEvent.AnalyticEventObject.PORTLET))
        {
            if (analyticEvent.getTotalTime() > SLA)
            {
                System.out.println("WARNING: portlet " +
                    analyticEvent.getDefinitionLabel() + " is exceeding SLA of " +
                    String.valueOf(SLA / 1000000000L) + " seconds.");
            }
        }
    }

    /**
     * <p>Implementation class may perform cleanup operations here.
     * Note: there is no guarantee this method will be called.
     * </p>
     */
    public void dispose() {
    }
}

```

The main method of interest is the `log(AnalyticEvent analyticEvent)` method. This method is invoked for every page and portlet on every request. The `AnalyticEvent` class contains a variety of information including the times for various lifecycle phases of the portlet.

The `AnalyticEventObject` has three attributes or methods that provide the functionality to detect a misbehaving portlet:

- `getAnalyticEventObject()` – Reports what type of object this event is for, such as a portlet or a page.
- `getDefinitionLabel()` or `getInstanceLabel()` – Provides the unique identifiers for the portlet.
- `getTotalTime()` – Provides the total time it takes for the portlet to run through all of its lifecycles.

If the total time for a portlet to render exceeds 5 seconds, an error message is logged to the console. Note that times returned by these methods are in nanoseconds. As illustrated in the next section, you can use the `ServiceLevelManager` service to disable the portlet and enable another portlet.

Generally, you should package the service provider in a JAR file. The JAR should consist of any necessary classes (including the `AnalyticEventHandler`), and file named `META-INF/services/com.bea.netuix.servlets.controls.analytics.AnalyticEventHandler`. This services file must contain one and only one class name and must be a concrete `AnalyticEventHandler` implementation.

The service provider JAR or JARs should then be deployed with the application by including the JAR in the web application's `WEB-INF/lib` directory. Provider JARs may also be included in the application or system classpath, although this changes the scoping of the provider class objects, and causes the provider implementations to be shared by multiple web applications.

Disabling the Bad Portlet and Enabling an Alternative Portlet

The `ServiceLevelManager` service allows you to enable and disable portlets based on a variety of identifiers. You can either disable a specific instance of a portlet or all instances of the portlet definition. In this example, all instances of the portlet definition are disabled. This choice is based on the assumption that if a portlet definition is poorly behaving, then all instances of that portlet are behaving poorly. You can just as easily disable one offending instance.

Alternatively, you can call the `ServiceLevelManger` interface from an administration JSP. If you are using this approach, the administration JSP would list all the poorly behaving portlets and an administrator would have to manually disable the misbehaving portlets and re-enable the good ones.

In the scenario described in “Use Case” on page 8-2, you create an alternate portlet that could display a message saying the service is down, or provide a cached set of results, or get information from an alternate source. Regardless of what the alternate portlet does, it goes in the same placeholder as the misbehaving portlet.

As illustrated in Listing 8-2, the alternate portlet is disabled on startup. When a portlet goes amiss, the alternate portlet is enabled and the bad portlet is disabled.

Listing 8-2 Enabling an Alternate Portlet and Disabling a Misbehaving Portlet

```

/** Service-provider interface for {@link AnalyticEventHandler}
implementations.
 * <p/>
 * An analytic event handler is a concrete subclass of this interface that
 * has a public no-argument constructor and implements the interface methods
 * specified below.
 * <p/>
 * An AnalyticEventHandler implementer should generally package their
 * provider in a jar. That jar should consist of any necessary classes
 * (including, of course, an implementation of AnalyticEventHandler), as well
 * as a file named <tt>META-INF/services/com.bea.netuix.servlets.controls.
 * analytics.AnalyticEventHandler<tt>. That services file must contain one
 * and only one class name which must be a concrete AnalyticEventHandler
 * implementation.
 * <p/>
 * The provider jar(s) should then be deployed with the application by
 * including the jar in the webapp's <tt>WEB-INF/lib</tt> directory.
 * Provider jars may also be included in the application or system classpath,
 * although this changes the scoping of the provider class objects, and
 * causes the provider implementations to be shared by multiple web apps.
 * <p/>
 * On initialization, the {@link com.bea.netuix.servlets.controls.
 * analytics.AnalyticEventDispatcher} will load such provider. Id the

```

Monitoring and Determining Portlet Performance

```
* Dispatcher fails to load or initialize the event handler a error message
* will be logged and no event handling will take place.
* <p/>
* NOTE: Implementations of the interface methods must be safe for use by
* multiple concurrent threads.
* <p/>
*/
public class MyAnalyticEventHandlerImpl implements AnalyticEventHandler
{
    private final static long SLA = 5000000000L; // Five seconds (nano time)
    /**
     * <p>Implementation class may perform any one-time initializations here.
     * If this method fails (throws an exception) the event handler will not
     * be registered and no event handling will take place.</p>
     */
    public void init() {
        System.out.println("My Analytic Event Handler Initialized");

        // Disable alternate portlet
        ServiceLevelManagerFactory serviceLevelManagerFactory =
            ServiceLevelManagerFactory.getInstance();
        ServiceLevelManager serviceLevelManager =
            serviceLevelManagerFactory.getServiceLevelManager("/portal_1");

        serviceLevelManager.setServiceLevelForDefinitionLabel(PortletService
Level.suspended, "alternate_pdl");
    }
    /**
     * <p>This method is called by the container at the end of each page's
     * and portlet's run. It is invoked for every page and portlet on every
     * request. Since this method is called so frequently the implementation
     * must be extremely efficient, or the entire portal's performace will
     * suffer.</p>
     * @param analyticEvent the event to be logged.
     */
    public void log(AnalyticEvent analyticEvent)
    {
        // Ignore all but portlet events.
    }
}
```

Disabling the Bad Portlet and Enabling an Alternative Portlet

```
if (analyticEvent.getAnalyticEventObject().equals
    (AnalyticEvent.AnalyticEventObject.PORTLET))
{
    // This will disable any portlet that has a response time > then
    // the SLA
    if (analyticEvent.getTotalTime() > SLA)
    {
        System.out.println("WARNING: portlet " +
            analyticEvent.getDefinitionLabel() + " is exceeding SLA of "
            + String.valueOf(SLA / 1000000000L) + " seconds.");

        ServiceLevelManagerFactory serviceLevelManagerFactory =
            ServiceLevelManagerFactory.getInstance();

        System.out.println("Servlet context path: " +
            analyticEvent.getServletContextName());

        // Note: service level manager is scoped to the context path
        // (request.getContextPath()). Before 10.2 this has to be known,
        // in 10.2, you can retrieve it from the AnalyticEvent using
        // String getWebappContextPath();

        ServiceLevelManager serviceLevelManager =
            serviceLevelManagerFactory.getServiceLevelManager("/portal_1");

        PortletServiceLevel portletServiceLevel =
            serviceLevelManager.getServiceLevelForDefinitionLabel
            (analyticEvent.getDefinitionLabel());

        System.out.println("Suspending Portlet: " +
            analyticEvent.getDefinitionLabel());
        serviceLevelManager.setServiceLevelForDefinitionLabel
            (PortletServiceLevel.suspended,
            analyticEvent.getDefinitionLabel());

        // Activating alternate portlet
        System.out.println("Activating Alternate Portlet ");

        serviceLevelManager.setServiceLevelForDefinitionLabel(PortletServiceLevel.
            active, "alternate_pdl");
    }
}
```

Monitoring and Determining Portlet Performance

```
    }  
    /**  
     * <p>Implementation class may perform cleanup operations here.  
     * Note: there is no guarantee this method will be called.  
     * </p>  
     */  
    public void dispose() {  
    }  
}
```

To get a reference to the `ServiceLevelManager`, you need to access it through the service level factory. `ServiceLevelManagers` are scoped to the web application. As such the factory requires the webapp context path. The method of interest is:

```
serviceLevelManagerFactory.getServiceLevelManager("/mywebapp");
```

To disable (or enable) a particular portlet, set the `PortletServiceLevel` on the selected portlet. In [Listing 8-2](#), the following code disables the alternate portlet:

```
serviceLevelManager.setServiceLevelForDefinitionLabel(PortletServiceLevel.  
    suspended, analyticEvent.getDefinitionLabel());
```

The alternate portlet is enabled by the same method:

```
serviceLevelManager.setServiceLevelForDefinitionLabel(PortletServiceLevel.  
    active, "alternate_pdl");
```

In this example, the bad portlet is not disabled until it has actually run because you do not know it is bad until it takes more than 5 seconds to run. In an actual application, you would probably use timeouts for a better implementation. Also, the alternate portlet does not go online until the next request, as this request has already finished. If you want to have the alternate portlet run as part of this request, you could perform a redirect that picks up the new portlet.

In WebLogic Portal 10.0, you had to hard code the web application context path:

```
ServiceLevelManager serviceLevelManager =  
    serviceLevelManagerFactory.getServiceLevelManager("/portal_1");
```

In WebLogic Portal 10.2 and later versions, you can get this path from the analytic event using:

```
String getWebappContextPath();
```

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 9-14). 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: If you want to use asynchronous rendering with IPC, consider using asynchronous desktop rendering, as discussed in the chapter “Designing Portals for Optimal Performance” in the *Portal Development Guide*. Note that IPC is not compatible with asynchronous

portlet rendering, as discussed in [“Asynchronous Content Rendering and IPC” on page 7-20](#), but workarounds exist for some use cases.

This chapter includes the following sections:

- [Definition Labels and Interportlet Communication](#)
- [Portlet Events](#)
- [IPC Example](#)
- [IPC Special Considerations and Limitations](#)

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 9-8](#).

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 Types](#)
- [Event Actions](#)
- [Portlet Event Handlers Wizard Reference](#)
- [JSF Events](#)

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. [Table 9-1](#) lists the event handlers that are available on the Add Handler menu of the Portlet Event Handlers wizard:

Table 9-1 Event Handlers

Event	Description
Handle Generic Event	Allows you to set up an event that will fire in several possible situations. For details, see Generic Event Handlers below.
Handle Portal Event	Responds to a portal framework event on a portlet by firing an action.
Handle Custom Event	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.
Handle PageFlow Event	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.
Handle Struts Event	Responds to an event on a portlet by firing a struts action.
Handle Faces Event	Responds to an event on a JSF portlet by firing a JSF action.

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 9-6](#). Events available with the portal event handler are listed in [Table 9-2](#).

Table 9-2 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

Table 9-2 Events Available to a Portal Event Handler

<code>onInit</code>	The <code>onInit</code> event is broadcast once per portal request. Use this event if you want to define an event handler that is always executed on every portal request.
<code>onLookAndFeelReinit</code>	<p>This event is fired when a Look & Feel is re-initialized. This happens whenever the Look & Feel is dynamically changed, such as when any of the <code>reinit()</code> methods on a <code>PortalLookAndFeel</code> object are called. See the <code>PortalLookAndFeel</code> class Javadoc for more information about the <code>reinit()</code> methods.</p> <p>For instance, if you capture information about the Look & Feel, you need to know when the Look & Feel changes so that you can refresh it with the captured information. This event provides that notification.</p>
<code>onCustomEvent</code>	<p>Mode change to the custom mode <code>CustomEvent</code></p> <p>Refer to “Event Handlers” on page 9-2.</p>

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 9-3](#) lists the event actions available for portlets.

Table 9-3 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.

Table 9-3 Event Actions

This action...	Has this effect...
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 5-71 .

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 9-14](#).

Portlet Event Handlers Wizard Dialogs

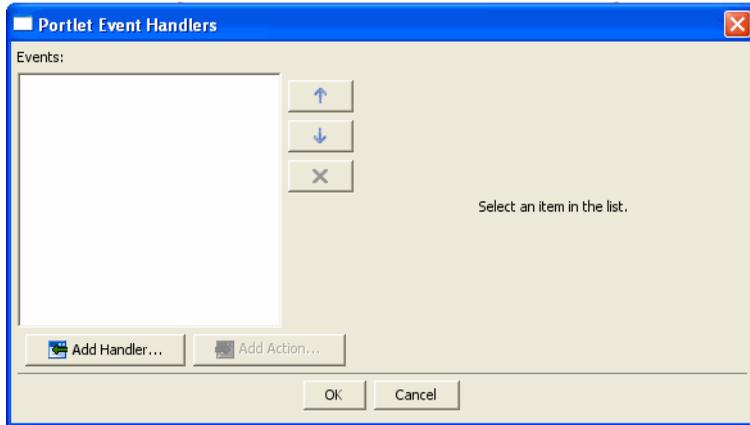
You can open the Portlet Event Handlers wizard in two ways:

- 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. Alternatively, you can click the Event Handlers link in the portlet editor.
- The wizard opens when you click the Event Handlers link in the portlet editor.

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 9-1](#).

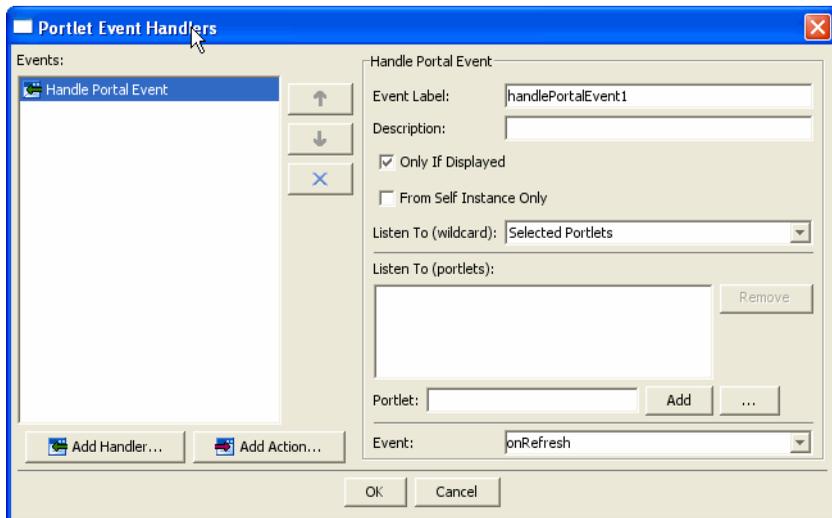
Figure 9-1 Portlet Event Handlers Wizard



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 9-2](#) shows an example of the expanded dialog for adding an event handler.

Figure 9-2 Expanded Event Handlers Dialog



Portlet Event Handlers Wizard - Add Handler Field Descriptions

Table 9-4 explains the fields in the Add Handler dialog and how your selections affect the behavior of the event.

Note: WebLogic Portal attempts to validate the settings of the Event Handlers dialog. You will receive an error message if any problems are detected. For detailed information on the WebLogic Portal validation framework, see the [WebLogic Portal Development Guide](#).

Table 9-4 Portlet Event Handlers Wizard - Add Handler

Field	Description
Event Label	Required. This identifier can be used by the <code><filterEvent></code> 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 <code>true</code> . Note: If the event is <code><handlePortalEvent event="onMinimize" fromSelfInstanceOnly="true"></code> then it is logically impossible for this event to fire if <code>onlyIfDisplayed="true"</code> .
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 <code>false</code> . If From Self instance Only is set to true, any Listen To values are ignored.
Listen To (wildcard)	Identifies the portlet(s) that this portlet can listen to. The values include: <ul style="list-style-type: none"> Any – Listens to events fired from any portlet in the portal. This – Listens to events fired from the currently selected portlet. Selected Portlets – (default) Listens to events fired from selected portlets only. Click the ... button in the Listen To (portlet(s)) part of the dialog to select portlets. This and Selected Portlets – Listens to events fired from the currently selected portlet and portlets selected in the Listen To (portlet(s)) part of the dialog.

Table 9-4 Portlet Event Handlers Wizard - Add Handler (Continued)

Field	Description
Listen To (portlets)	<p>Optional. Allows you to specify the portlets that this portlet can listen to. You can choose a <code>.portlet</code> file from the file system by clicking the <code>...</code> button). When you select a <code>.portlet</code> file and click OK, the portlet is added to the Listen To list. This part of the dialog is only enabled if you chose either the Selected Portlets or This and Selected Portlets option in the Listen To (wildcard) menu.</p> <p>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.</p>
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 <code>HandlePortalEvent</code> handler, you can use the Event drop-down menu to select portal events, such as the <code>onRefresh</code> event. If you choose a handler that exposes actions, type the name of the action in the Action field. For example, if you chose <code>HandlePageFlowEvent</code> , you could type <code>submitReport</code> . The <code>submitReport</code> 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 9-5](#) explains the possible fields in the expanded Add Action dialog and how your selections affect the behavior of the action.

Table 9-5 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.

Table 9-5 Portlet Event Handlers Wizard - Add Action

Field	Description
Activate Page	This action activates the page on which the portlet <code><portlet_def_id></code> currently resides. This action will fire only when triggered during the <code>handlePostBack</code> 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	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. This action is only available on the menu for struts portlets.
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 action is only available on the menu if a backing file is configured for the portlet.
Invoke Page Flow Action	Use this selection to cause a page flow action to be raised. This action is only available on the menu for page flow portlets.
Invoke Faces Action	Use this selection to cause a JSF action to be raised. This action is only available on the menu for JSF portlets.

JSF Events

This section explains how to add JSF events and listeners to portlets.

To add a faces event to a JSF portlet:

1. In the Property view of the portlet, select **Java Server Faces (JSF) Content > Faces Events**.
2. In the Faces Events dialog, add an name/action pair. The Event Name identifies the event to event listeners. Note that events are referenced by name, not by path. The action specifies the Faces viewroot ID of the action. Workshop for WebLogic adds tags to the portlet similar to those shown in [Listing 9-1](#).

Listing 9-1 Sample facesEvent Tag

```

<netuix:content>
  <netuix:facesContent contentUri="/portlets/myJsf.faces"
    requestAttrPersistence="none">
    <netuix:facesEvents>
      <netuix:facesEvent action="/portlets/bar.faces" eventName="bar"/>
    </netuix:facesEvents>
  </netuix:facesContent>
</netuix:content>

```

To handle a faces event and invoke an action, use the Portlet Event Handlers wizard as discussed in [“Portlet Event Handlers Wizard Reference”](#) on page 9-6. When specifying the Event to listen to, use the event name specified in the facesEvent tag in the JSF portlet (the eventName attribute of the facesEvent tag). For example, in [Listing 9-1](#) the eventName is bar.

[Listing 9-2](#) shows the handleFacesEvent tag that is created when you add a Faces event handler to a JSF portlet. The event handler is listening for an event called bar. (Note that you can use eventName="*" to handle any Faces event that is fired.) The event handler then invokes a JSF action.

Listing 9-2 Faces handleFacesEvent Tag

```

<netuix:handleFacesEvent eventLabel="handleFacesEvent1" eventName="bar"
  fromSelfInstanceOnly="false" onlyIfDisplayed="true"
  sourceDefinitionLabels="myJsf">
  <netuix:invokeFacesAction action="/portlets/myportlet.faces"/>
</netuix:handleFacesEvent>

```

When working with JSF events, note the following:

- Faces events are named with an alias.
- Faces event handlers reference the event alias.
- Faces actions are invoked by the viewroot ID, as shown in [Listing 9-2](#).

- In a Faces event handler, you can use `eventName="*"` to handle any Faces event that is fired.

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 9-6](#) 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 9-6 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	<p>In the Generate a domain configured automatically to support the following Oracle products list, select WebLogic Portal.</p> <p>When you do this, other components are selected automatically; <i>keep all of them selected</i>.</p>

Table 9-6 IPC Example - Environment Setup Values (Continued)

Setup Information	Notes/Values
Domain Configuration Wizard - Configure Administrator Username and Password	User name: <code>weblogic</code> User password: <code>weblogic</code> Confirm user password: <code>weblogic</code>
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: <code>ipcDomain</code> Domain location: Accept the default, or specify another directory on your system.
Portal EAR Project Wizard	EAR Project Name: <code>ipcEAR</code> 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 <code>ipcEAR</code> project with the portal domain <code>ipcDomain</code> .
Portal Web Project Wizard	Web Project Name: <code>ipcTestWebProject</code> In the Add project to an EAR checkbox: Check the box and add to <code>ipcEAR</code>
Portal Wizard	Right-click the <code>ipcWebProject/WebContent</code> folder and select New > Portal Portal Name: <code>ipcPortal</code>

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 9-6](#). For more information on backing files, refer to [“Backing Files” on page 5-71](#).

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.

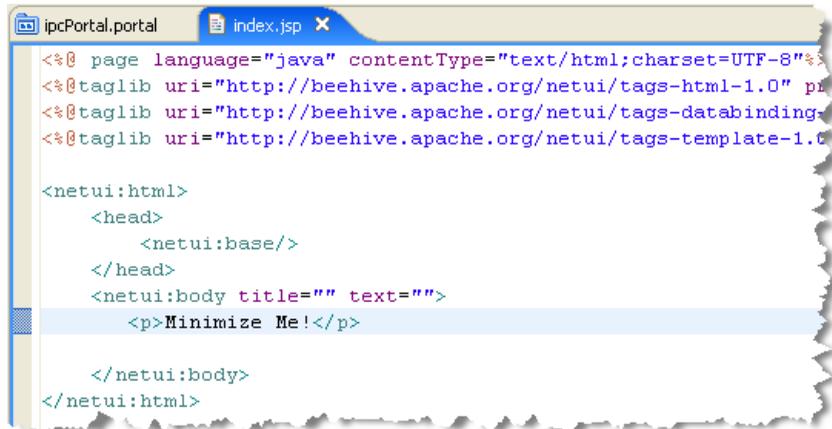
Create the JSP Files and Portlets

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 9-3 index.jsp after Editing the Body Text in the Workbench Editor



```

<? page language="java" contentType="text/html; charset=UTF-8" ?>
<? taglib uri="http://beehive.apache.org/netui/tags-html-1.0" ?>
<? taglib uri="http://beehive.apache.org/netui/tags-databinding-1.0" ?>
<? taglib uri="http://beehive.apache.org/netui/tags-template-1.0" ?>

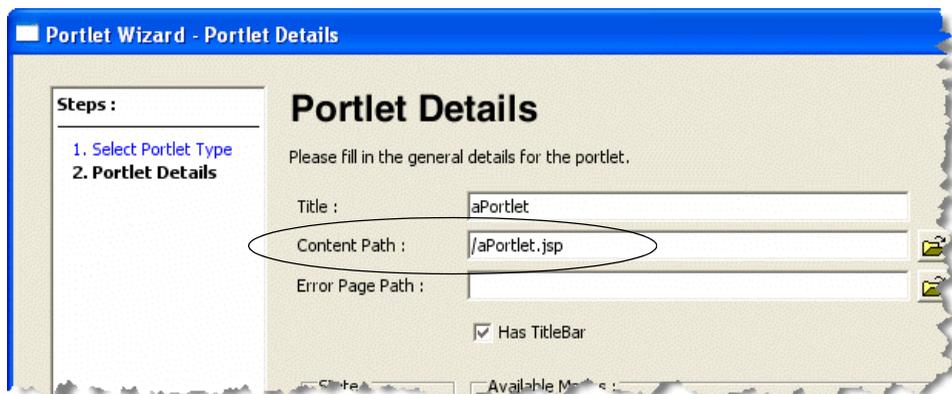
<netui:html>
  <head>
    <netui:base/>
  </head>
  <netui:body title="" text="">
    <p>Minimize Me!</p>
  </netui:body>
</netui:html>

```

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 9-4), with `aPortlet.jsp` in the Content Path field.

Figure 9-4 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 9-10](#) 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 9-10 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 9-5](#).

Figure 9-5 Updated bPortlet JSP Source

```

<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://beehive.apache.org/netui/tags-html-1.0" prefix="netui"%>
<%@ taglib uri="http://beehive.apache.org/netui/tags-databinding-1.0" prefix="netui-data"%>
<%@ taglib uri="http://beehive.apache.org/netui/tags-template-1.0" prefix="netui-template"%>

<netui:html>
  <% String event = (String)request.getAttribute("minimizeEvent");%>
  <head>
    <title>
      Web Application Page
    </title>
  </head>
  <body>
    <p>
      Listening for portlet & minimize event:<%=event%>
    </p>
  </body>
</netui:html>

```

9. Save the file.
10. Following the same steps you used previously, generate a portlet from the `bPortlet.jsp` file.

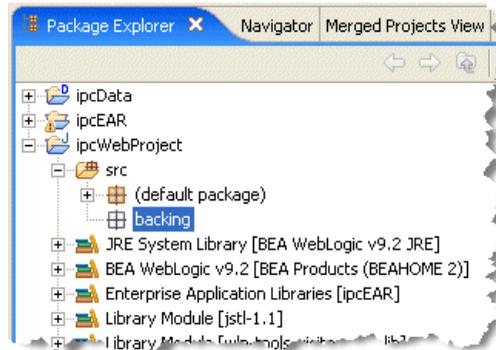
Checkpoint: At this point the `ipcWebProject/WebContent` folder contains these files: `aPortlet.jsp`, `aPortlet.portlet`, `bPortlet.jsp`, and `bPortlet.portlet`.

Create the Backing File

To create the backing file, do the following:

1. In `ipcTestWebProject`, select the `Java Resources/src` folder and select **File > New > Folder** from the main menu.
The Create New Folder dialog box appears.
2. Create a folder called `backing`.
The folder `backing` will appear under `ipcTestWebProject/src`, as shown in [Figure 9-6](#).

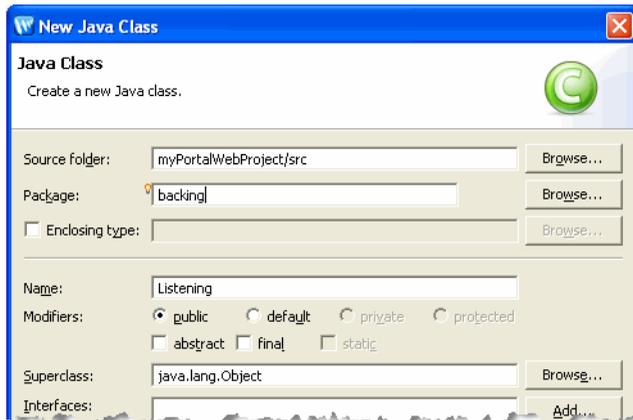
Figure 9-6 New Backing File Folder in Package Explorer View



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 9-7. 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 9-7 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 9-3](#) into the file. [Figure 9-8](#) shows the top portion of the `Listening.java` file as it should look after you paste the code into it.
7. When you're finished, save the file.

Listing 9-3 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)
        {

```

Local Interportlet Communication

```
        // 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 9-8 Listening.java with Updated Backing File Code

```

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
        // (i.e. to the pre-render phase). Passing data between backingfile
        // 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, "minimised!");
    }

    public boolean preRender(HttpServletRequest request, HttpServletResponse
        response)
    {
        String attributeId= this.getPortletInstanceLabel(request) + "_minimizeEventHandled";
    }
}

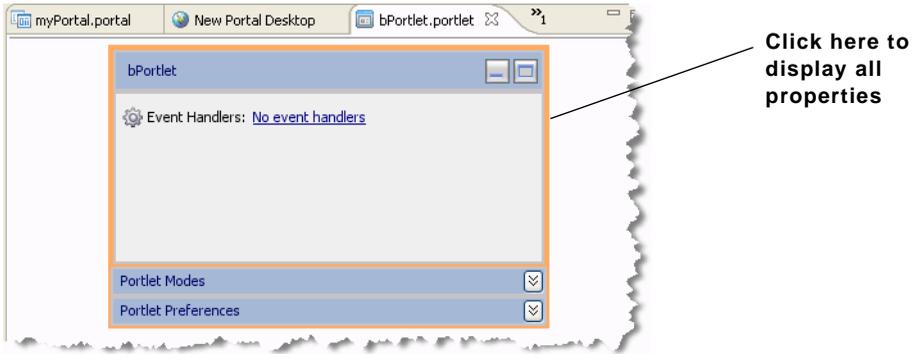
```

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 9-9](#).

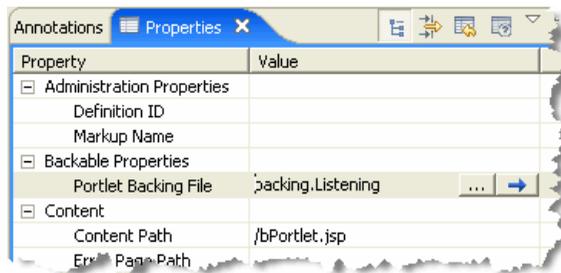
Figure 9-9 bPortlet with Outer Border Selected to Display Properties



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 9-10](#).

Figure 9-10 Attaching the Backing File in the Properties View



4. Save the portlet file.

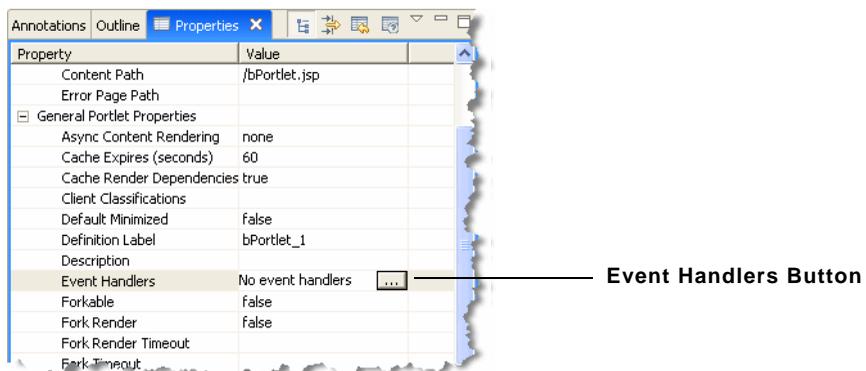
Add the Event Handler to bPortlet

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 `ipcTestWebProject/WebContent` folder in the application panel and double-click it.

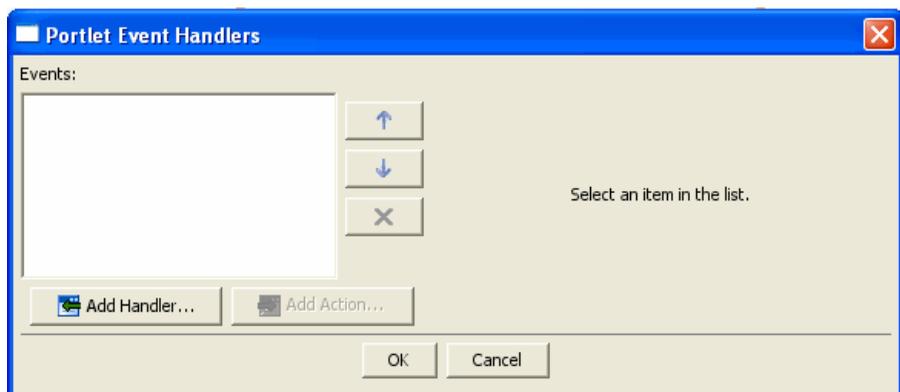
1. Click on the portlet in the editor if needed to display its properties.
1. In the Properties view, click in the **Value** column of the **Event Handlers** property. A browse button `...` appears, as shown in [Figure 9-11](#).

Figure 9-11 Event Handlers Button



2. Click the ellipsis button `...` to display the Portlet Event Handlers dialog, as shown in [Figure 9-12](#).

Figure 9-12 Portlet Event Handlers Dialog Box

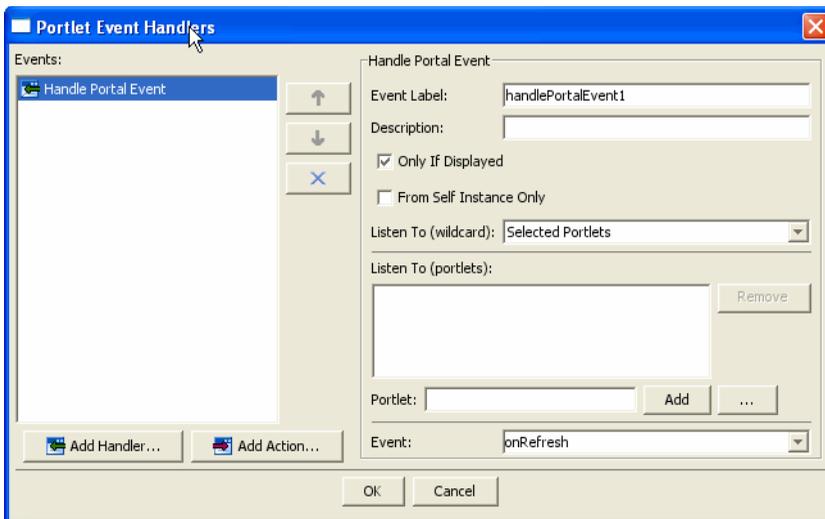


3. Click **Add Handler** to open the **Event Handler** drop-down list.

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 9-13](#).

Figure 9-13 Event Handler Dialog Box Expanded



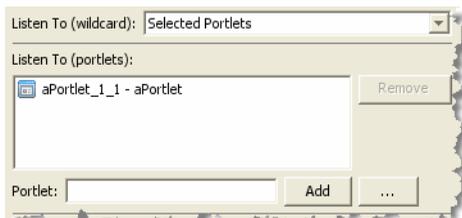
5. Accept the defaults for all fields except **Portlet**.
6. In the **Portlet** field, click the ellipsis button .

The **Please Choose a File** dialog appears.

7. Click `aPortlet.portlet` and click **OK**.

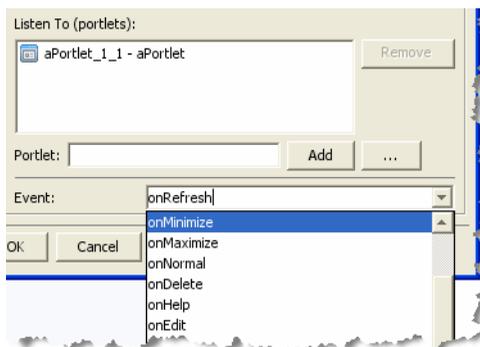
The dialog box closes and `aPortlet_1` appears in the **Listen to (portlets):** list and in the **Portlet** field, as shown in [Figure 9-14](#). The label `aPortlet_1` is the definition label of the portlet to which the event handler will listen.

Figure 9-14 Adding portlet_1



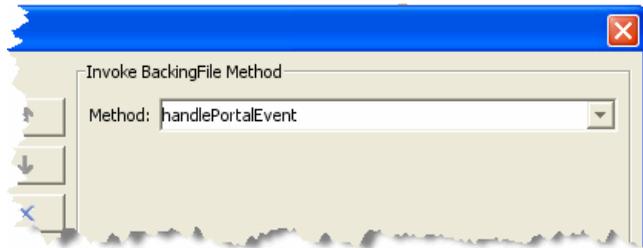
8. Click the **Event** drop-down control to open the list of portal events that the handler can listen for and select **onMinimize**, as shown in [Figure 9-15](#).

Figure 9-15 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 9-16](#).
The dropdown menu for this field displays the last several values that you entered, if applicable.

Figure 9-16 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.

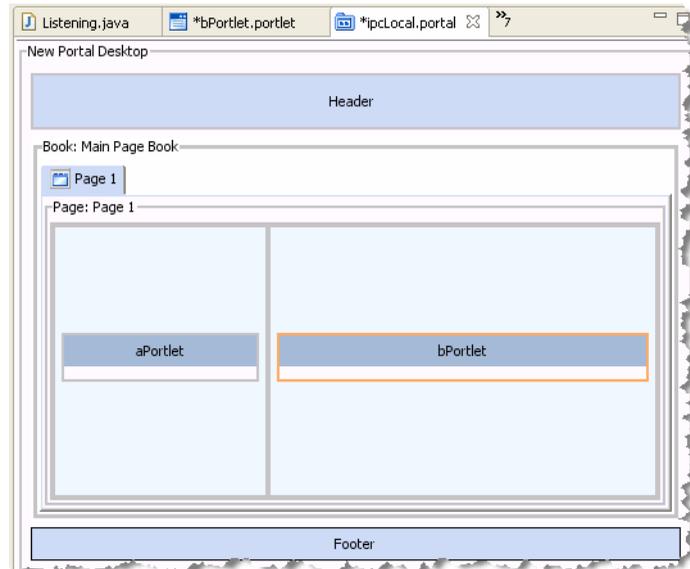
Test the Project

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 9-17](#).

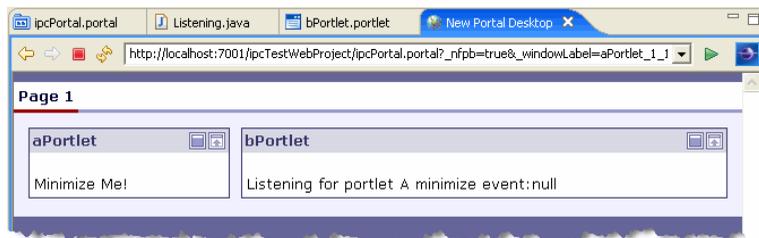
Figure 9-17 Portal Layout with aPortlet and bPortlet Added



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 9-18](#)).

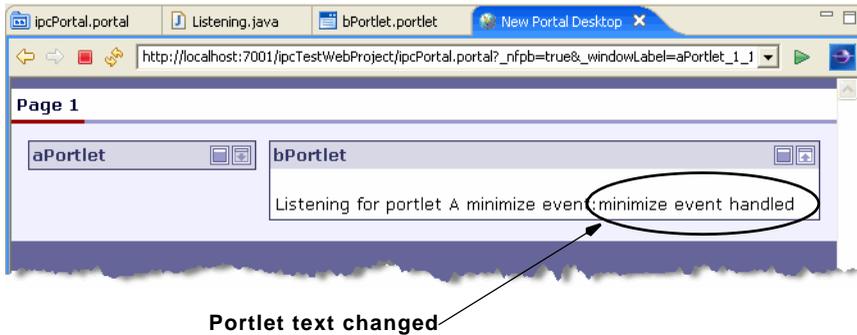
Figure 9-18 ipcLocal Portal in Browser



6. Click the minimize button to minimize aPortlet.

Note the content change in bPortlet, as shown in [Figure 9-19](#).

Figure 9-19 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 for specific portlets 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 7-20.

Tip: If you enable asynchronous rendering at the portal/desktop level, you can use IPC without restrictions. For more information on asynchronous portal/desktop rendering, see the [WebLogic Portal Development Guide](#).

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](#).

Local Interportlet Communication

Adding the Content Presenter Portlet

The Content Presenter portlet allows users to retrieve and display different kinds of content in a portal in real time, without assistance from your IT Department or software developers. For example, you might want to display a list of the most recent Press Releases so users can browse them and click one to read the entire Press Release. You can place images (a photograph or a chart, for example) or add textual content on a portal page. You can also segregate content by subject matter to target different audiences.

In the Content Presenter Example, you can perform inline editing to quickly change the content that displays in the portlet.

Tip: The Content Presenter portlet works only with streamed portals.

This chapter includes these sections:

- [Using the Content Presenter Example](#)
- [Configuring the Content Presenter Portlet in Your Portal](#)

Using the Content Presenter Example

WebLogic Portal includes a Content Presenter Example, and allows you to perform inline editing on a Content Presenter portlet to modify the portlet's content. Editing the portlet's content also changes the content in the content repository. The Example's Content Presenter portlet uses the public Dojo rich text editor.

This section contains the following topics:

- [Starting the Content Presenter Example](#)
- [Performing Inline Editing in the Content Presenter Example](#)
- [Enabling Inline Editing in Your Portlets](#)
- [Configuring the Content Presenter Portlet](#)

Starting the Content Presenter Example

Note: You must install the Portal Examples before you perform these steps. See “Installing the Sample Applications and Domain” in the [WebLogic Portal Release Notes](#) for details.

To start the Content Presenter sample:

1. From the Windows Start Menu, start the WebLogic sample server. (You can also double-click the `startWebLogic.cmd` file located in the `<WLPORTAL_HOME>/samples/domains/portal/bin` directory.)
2. After the server starts, from the Windows Start Menu choose **Oracle Products > WebLogic Portal > Examples > Visit Portal Examples**.
3. On the WebLogic Portal Sample Domain, select **Go to the Content Presenter demo**. (You can also launch the Content Presenter Example in a browser at `http://localhost:7041/contentpresenter/`.)
4. Enter your username and password (for example, `weblogic/weblogic`) and click **Login**.

See [Performing Inline Editing in the Content Presenter Example](#) for instructions on how to perform inline edits to the content in a portlet in the Content Presenter Example.

Performing Inline Editing in the Content Presenter Example

The Content Presenter Example is the only portal where you can perform inline HTML content editing without doing additional setup tasks. By default, the Content Presenter Example lets you immediately edit the content in the Letter from the CEO Portlet, or you can enable the Training Announcement Portlet for inline editing. Inline editing in the Content Presenter Example works only on single-item portlets.

The Letter from the CEO portlet in the Content Presenter Example is already configured for inline editing because it uses the template view with inline editing enabled and has the appropriate entitlement rights. To configure the Training Announcement portlet for inline editing, see “[Enabling Inline Editing for the Training Announcement Portlet](#)” on page 10-4.)

This section contains the following topics:

- [Entering Inline Edits](#)
- [Enabling Inline Editing for the Training Announcement Portlet](#)

Entering Inline Edits

To enter inline edits to the Letter from the CEO portlet:

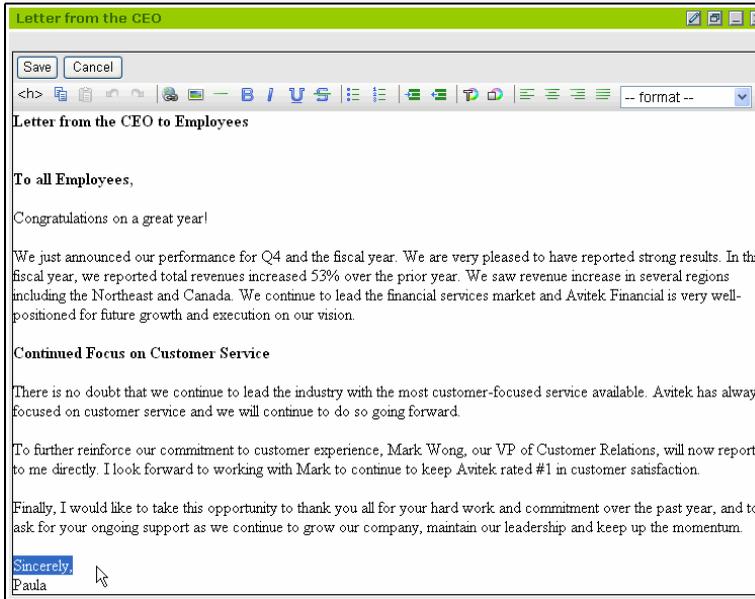
1. Follow the steps in [Starting the Content Presenter Example](#) to start the Content Presenter Example and log in.
2. After you log in, click **Edit HTML** in the Letter from the CEO portlet, as shown in [Figure 10-1](#). The Content Presenter Configuration Wizard appears.

Figure 10-1 Click the Edit HTML Button that Appears in the Text of the Portlet



3. Enter your edits or insert a link to other content or graphics outside your content management system. For example, before the signature line, type **Sincerely**, as shown in [Figure 10-2](#).

Figure 10-2 Enter Text in the Signature Line



4. Click **Save**. Your changes appear in the portlet and are saved to the CM Repository.

Tip: You can perform inline editing on two portlets in the Content Presenter Example. Inline editing is not available for the Content Presenter portlets in your portal.

Enabling Inline Editing for the Training Announcement Portlet

One other portlet in the Content Presenter Example, the Training Announcement portlet, allows inline editing. Inline editing is turned off by default for this portlet, so you must first enable the inline editing capability by choosing a different content display template view.

To enable inline editing for the Training Announcement portlet:

1. Follow the instructions in “[Starting the Content Presenter Example](#)” on page 10-2 to log into the Content Presenter Example.
2. In the Training Announcement portlet in the Content Presenter Example, click **Edit** in the portlet’s title bar, as shown in [Figure 10-3](#).

Figure 10-3 Click Edit to Enable Inline Editing for this Portlet

3. In the Content Presenter Configuration Wizard's Single or Multiple Items window, click **Next**. (If you use the wizard to change this portlet to allow multiple content items, rather than a single item, you cannot enable inline editing. For more information on configuring the portlet, see [“Configuring the Content Presenter Portlet” on page 10-7.](#))
4. In the wizard's Select Content window, click **Next**.
5. In the wizard's Select Template & View window, select **Single Item View with Inline Edit** and click **Next**.
6. In the wizard's Portlet Properties window, click **Next**.
7. In the wizard's Finish window, click **Save**. The Edit HTML button appears in the Training Announcement portlet. If you do not see the Edit Content button, you might not have the correct entitlement. See the [Security Guide](#) for instructions on setting entitlements.
8. Click **Edit HTML** to change the content in the portlet or insert a link to other content or graphics outside your content management system.
9. Click **Save**. Your changes are saved to the CM Repository.

Enabling Inline Editing in Your Portlets

You can use the three sample JSP files that ship with WLP to enable inline HTML editing in your own Content Presenter portlets. Inline editing does not work with library services enabled, because library services support versioning.

Edit the files in the order listed below. The files are located in the following directories:

1. Display Template (Outer Template) –


```
<WLPORTAL_HOME>\samples\applications\portalApp\
contentPresenterSampleWeb\samplePresenterTemplates\
inlineEditExamplePresenterTemplate.jsp
```

2. CM Display Template (Inner Template) That Displays the Content –

```
<WLPORTAL_HOME>\samples\applications\portalApp\contentPresenterSampleWeb\sampleCMTemplates\inlineEditExampleCMTemplate.jsp
```

3. JSP File that Performs Other Work –

```
<WLPORTAL_HOME>\samples\applications\portalApp\contentPresenterSampleWeb\sampleCMTemplates\saveNode.jsp
```

The files include detailed comments to help you customize them for your portlets. For example, you might want to replace the DOJO rich text editor with your own rich text editor. You might want to change the entitlements on the portlets or their look and feel.

Configuring the Content Presenter Portlet in Your Portal

The Content Presenter portlet ships with WebLogic Portal. You must configure the portlet before you can use it.

Tip: To see a sample web application built with the Content Presenter portlet, download a demo at <http://wlp.bea.com>.

The Content Presenter portlet uses a portlet framework that is based on Content Management, metadata, and templates that let business users step through a wizard to quickly retrieve and display content that is appropriate to the audience. The framework allows WebLogic Portal customers to easily publish content in a variety of ways to almost any site.

The Content Presenter portlet can read content from any configured content provider. Content providers can be any repository that implements the WebLogic Portal Content Management Service Provider Interface, including third-party Content Management vendor products, file systems, or other database systems.

When you plan your Content Presenter portlet, determine who will view your content and if you plan to re-use the portlet later for a different audience. Plan entitlements to determine who can choose content to display in the Content Presenter portlet. If a logged-in user has Delegated Administration rights, the user can edit the content in the Content Presenter portlet.

You must be a member of the Portal System Administrators role or the Content Presenter Administrators role to configure the Content Presenter portlet. Portal System Administrators and Content Presenter Administrators have edit and delete capabilities on the Content Presenter portlet itself, and edit capabilities on any page where the portlet is placed.

If you plan to have a group of users (for example, a subset of the Portal System Administrators role) edit the Content Presenter portlet, this subgroup must have edit and delete capabilities for

the Content Presenter portlet itself and edit capabilities for the page that contains the portlet. If the group does not have edit and delete capabilities to the portlet, the group's members will not be able to see the Edit icon in the portlet.

See the [Security Guide](#) for more information on setting entitlements and creating roles and groups. Portal System Administrators and Content Presenter Administrators can also turn a portlet off by disabling the activation flag in the portlet's preferences in the Administration Console.

WARNING: If you add or modify your Content Presenter portlet using the Visitor Tools (rather than the Administration Console), the portlet is no longer configurable in your desktop. You should add or modify your Content Presenter portlet in the Administration Console.

If you use Portlet Publishing to configure a Content Presenter portlet, some features are not available. See [“Using Portlet Publishing to Expose a Content Presenter Portlet”](#) on page 10-19 for more information.

This section contains the following topic:

- [Configuring the Content Presenter Portlet](#)

Configuring the Content Presenter Portlet

The Content Presenter portlet ships with WebLogic Portal. You must add the Content Presenter facet in Workshop for WebLogic when you set up your portal web project. By default, your portal administrator has the ability to administer the Content Presenter portlet (to move portlets, turn a portlet off, and so on) in the Administration Console.

In a desktop, use the Content Presenter's Configuration Wizard to determine the content you want to display, and how to display it (through templates and template views). The Content Presenter portlet stores those choices as portlet preferences for each portlet instance.

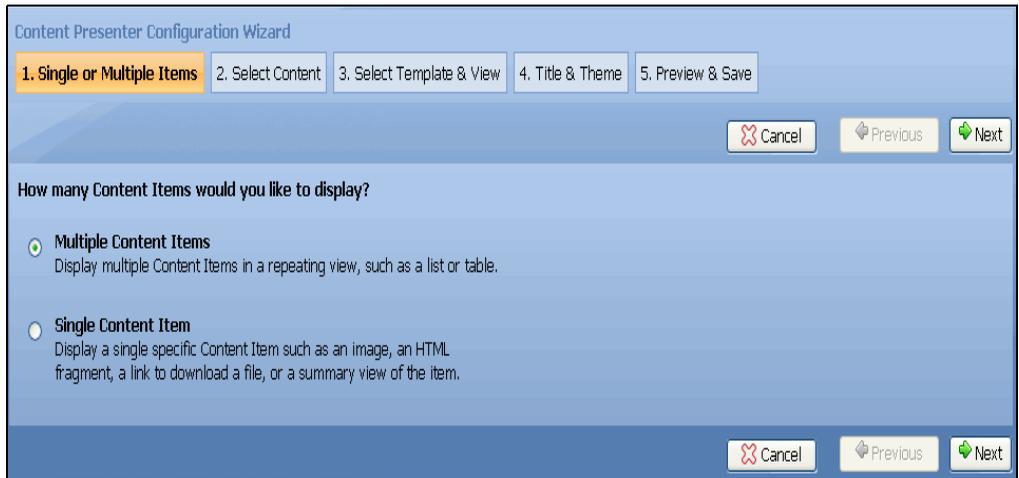
Perform the following steps to configure the Content Presenter portlet:

1. In Workshop for WebLogic, create and deploy a Portal EAR project, web project, a portal, and a method to authenticate users (such as a login portlet) according to the instructions in the [Getting Started With WebLogic Portal](#) tutorial. You should also create a Datasync project if you plan to use Content Selectors to display content in the Content Presenter portlet.

Tip: When you create your Portal Web Project, you must select the **Content Presenter Framework** facet in the **WebLogic Portal (Optional)** directory in order to view and use the Content Presenter portlet.

2. Start the Administration Console and choose **Portal > Portal Management**.
3. Create a page and a desktop (you can choose to create a desktop from a desktop template, library resources, or a `.portal` file) according to the instructions in [Getting Started With WebLogic Portal](#) tutorial.
4. Select the page you created in [step 3](#) in the Portal Management tree in the `Library/Pages` directory (or wherever you saved it). Add the Content Presenter portlet to the page by clicking **Add Page Contents**. Click **Add Contents** in the appropriate column, select the check boxes next to the Content Presenter portlet and a login portlet (if that is the method you are using to authenticate users), and click **Save**. (If you do not see a list of portlets in the Add Books and Portlets to Placeholder page, click the drop-down box and select **Portlets**, and click **Show all**.)
5. In the Portal Management tree, select the **Portals** directory, the portal you created, and your desktop.
6. With the desktop selected, click **View Desktop**.
7. In the new browser window, log into the desktop and click **Open Configuration Wizard** in the **Content Presenter** portlet. A new unconfigured Content Presenter portlet does not appear until you log in, and only if you have rights to edit it.
8. In the Content Presenter Configuration Wizard, select one of the following in the **Single or Multiple Items** window:
 - **Multiple Content Items** – Pick more than one content item and display them. You can create a custom list of content, choose all content in a specific folder, use the results of a Content Selector, or run a search to find content. The result is a list of content items in the portlet.
 - **Single Content Item** – Browse or search for one item. The item can be an image, article, link to a file, link to a URL, and so on. The result is the content item displaying inline in the portlet.

See [Figure 10-4](#). Click **Next** after you select an option.

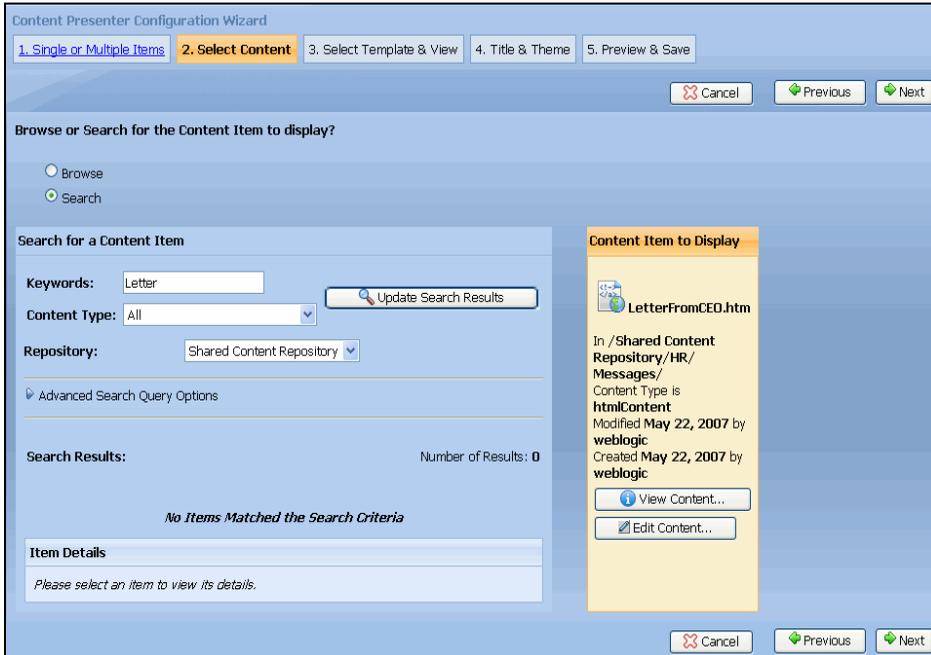
Figure 10-4 Determine How Much Content You Want to Display

9. The choice you made in [step 8](#) in this section determines which wizard pages you see and what you choose next.
- If you chose **Single Content Item** in [step 8](#), you can select **Browse** in the wizard's **Select Content** window to navigate through the repository tree to locate content, select the item, and click **Next**.

You can also select **Search** to find content by keyword or content type. (If you have multiple content repositories configured, you can also search by repository.) By default, all content types are listed in the Content Type field. If you want to control which content types appear in the Content Type field, you can set entitlements by user. In order to have these entitlements by user evaluated, override the `com.bea.content.ui.framework.AllowObjectClassViewRights` context parameter in the `web.xml` file. If you change the setting to **false**, the entitlements set on the content types determine the content types that are listed in the Content Type field. If you change the setting to **true**, or you omit the context parameter altogether, all content types appear for all users.

See the [Security Guide](#) for instructions on setting up users and roles in the Administration Console, and adding view capabilities so users can see the object classes in the Content Type field. Click **Update Search Results** to view the results. Adjust the **Items per Page** field to determine how many search results to display in the wizard. Select an item; see [Figure 10-6](#). Click **Next**.

Figure 10-5 Narrow Your Search with a Keyword



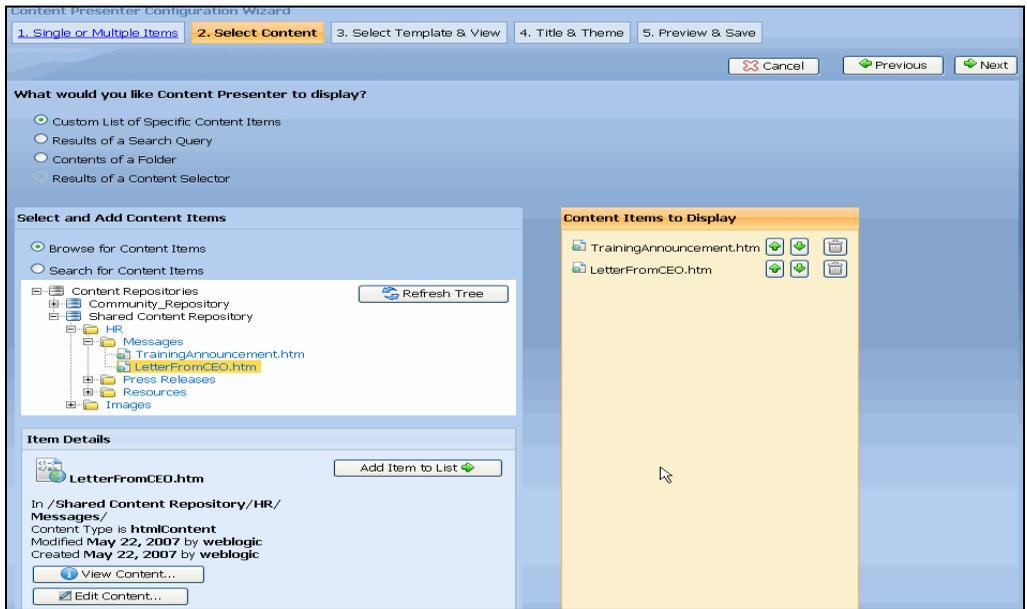
b. If you chose **Multiple Content Items** in [step 8](#) of this section, select one of the following in the wizard’s **Select Content** window to specify your content items:

- **Custom List of Specific Content Items** – This type of content retrieval is not a live search; it displays content from a list that you create. Click **Browse for Content Items** to navigate through the repository tree to locate content.

You can also click **Search for Content Items**, enter a keyword, and click **Update Search Results** to retrieve content by keyword. If you have multiple content repositories configured, you can also search within a specific repository. To search your repository, you must configure it and make it searchable. The repository that ships with a WebLogic Portal for a new domain is not automatically indexed for searching. See the [Content Management Guide](#) for instructions on indexing your repository.

[Figure 10-6](#) shows that after you retrieve multiple items and click **Browse for Content Items**, you can select an item and click **Add Item to List**. The custom list lets you control the order in which the content displays by selecting the check box next to the item and clicking **Move Up** or **Move Down**. Moving content to the top of the list ensures that a certain content item displays first. See [Figure 10-7](#).

Figure 10-6 Custom List of Specific Content Items



You can also verify that you selected the correct content by clicking **View Content** to preview the content you selected. The actual content and other details appear in a separate window. If you selected an image, the image also appears. You can also click **Edit Content** to change the property values of the item. If the item is a binary file, you can download the item, upload new values, or change the property values.

After you determine the order of the content, click **Next**.

- **Results of a Search Query** – Locate content by entering a keyword or selecting a content type (such as an image or a book) and clicking **Preview Query Results**. You can enter multiple keywords (separated by a space) and the *or* connector is assumed. For example, if you search for *IRA retirement*, results will include the keyword *IRA* or *retirement*. If you have multiple content repositories configured and you are entitled to view them, you can also search by repository.

You can click **Advanced Search Query Options** to create a query filter or a sort filter. Clicking **Create New Query Filter** applies a filter to your query, such as property name, an operator, and a value to narrow down your search results. For example, you can search for content created after 1/1/07 by selecting the **Creation Date** property, selecting **After**, and then entering a date. Click **Add Filter** and the new filter appears in the **Query Filters** section, as shown in [Figure 10-7](#).

Tip: The *Similar* operator retrieves results that contain words that are similar to the keyword (for example, the word might be misspelled).

You can click **Create New Sort Filter** to determine how to display the search results. For example, you can display the most recently-modified content by selecting the **Last Modified Date** property, selecting **Descending**, and clicking **Add Sort Filter**. The new sort filter appears in the **Sort Filters** section. See [Figure 10-7](#) to see how you can display the most recently-modified content first (descending order). You can change the order in which the results are sorted by clicking the up or down arrow to move the items in the sort filter up or down in the list (from ascending to descending, for example.)

Figure 10-7 Query Filter and Sort Filter

The screenshot shows a web interface titled "Create a Search Query". It includes the following elements:

- Keywords:** A text input field containing "investors".
- Content Type:** A dropdown menu set to "PressRelease".
- Repository:** A dropdown menu set to "Shared Content Repository".
- Advanced Search Query Options:** A section header with a collapsed icon.
- Query Filters:** A section containing a filter "Creation Date Before 2007-6-13:0:0:0" with a "Remove Filter" button and a "Create New Query Filter" button.
- Sort Filters:** A section with the text "There are no Sort Filters currently applied." Below it are two columns: "Sort Property" with a dropdown set to "Creation Date", and "Sort Direction" with radio buttons for "Ascending" (selected) and "Descending". There are "Add Sort Filter" and "Cancel" buttons.
- Results Preview:** A section with a "Preview Query Results" button.
- Results List:** A table with the following entries:

Title
AvitekFutureInvestorsProgram.htm
BoardMemberAnnouncement.htm
DividendsAnnouncement.htm
NewRewardsCreditCard.htm
- Summary:** "Number of Results: 4" and "Items per Page: 500" with a dropdown arrow.

Click **Preview Query Results** to view the results of the search query. The search query searches both property values and binary content. Adjust the **Items per page** field to determine how many search results to display in the wizard (it does not affect the final display). Each time someone visits the Content Presenter portlet, the search is re-run. Since content in the repository can change, the results might be different the next time the portlet renders. Click **Next**.

Tip: Running a search query is more resource intensive than retrieving content from a specific node or retrieving the contents of a specific folder. When possible, try to retrieve specific content, rather than running a search.

- **Contents of a Folder** – Navigate through the repository tree to locate a content folder, click **Select this Item**, and click **Next**. Each time a user views this portlet, the most current content items in the selected folder are retrieved. Any content can behave like a folder; therefore, you can select any child items under that content.
- **Results of a Content Selector** – Select a Content Selector that you created in WebLogic for Workshop, and click **Select this Item**. Content Selectors use rules to target specific groups of people with content items from the WLP Virtual Content Repository. See the *Interaction Management Guide* for more information. Each time the portlet renders, the most current Content Selectors are retrieved. Click **Next** after you locate the Content Selector you want to display.

10. In the wizard's **Select Template & View** window, select an item from the **Template Category** field, and then select a template. A template category helps you organize your templates and template views. You can have as many template categories as you need, but you should plan your organization strategy early, so that you do not have to update the preference values of existing Content Presenter portlet instances. A template is similar to a folder and is used to organize content. The default template that appears is based on the single or multiple content choice you made in [step 8](#) in this section.

WebLogic Portal ships with the following two default templates:

- **Default Single Item Template** – Lets you view a single item and a single property. See [Figure 10-8](#). This template appears because you chose **Single Content Item** in [step 8](#).
- **Default Multiple Items Template** – Lets you view multiple items and properties in a bulleted list. This template appears because you chose **Multiple Content Items** in [step 8](#).

You should create your own custom template based on the content you want to display. The custom templates you create appear in the wizard. See the [Content Management Guide](#) for instructions on creating custom templates and views.

Select an item in the **Template Views** field. A template view controls the layout and formatting of the content in the portlet. For example, you might create a template for your company's Press Releases. You could then create several custom template views that display the content in the following ways: a summary of the Press Release, the full text of the press release, details of the Press Release, and so on.

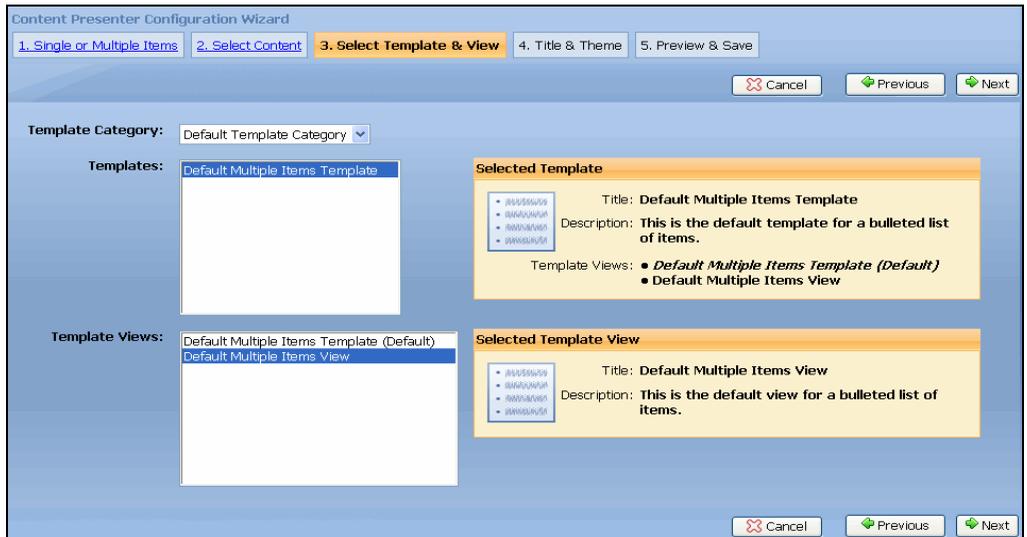
If you chose the Default Single Item Template in [step 10](#) in this section (which displays a single piece of content), or a custom template that you created that does not contain views, select **Default Single Item View** for the template view. If you chose the Default Multiple Items Template in [step 10](#) in this section, select **Default Multiple Items View** for the template view. If you created a custom template view as described in the [Content Management Guide](#), that template view also appears here.

For templates or views that support pagination of multiple content items, set the **Items per Page** value to the maximum number of items you want to display at one time on the template. With the appropriate pagination JSP tags on the template or view, users can navigate large numbers of items while viewing only a manageable few.

Note: You can customize how templates and views paginate in the Content Presenter Configuration Wizard. For more information on the pagination tags, see the CM JSP Tag [Javadoc](#).

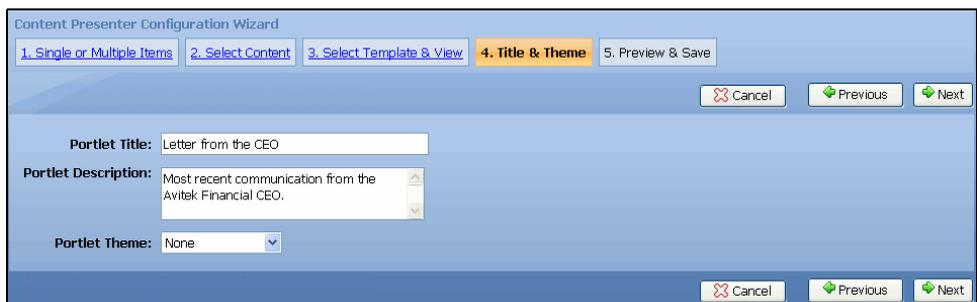
Click **Next** after you select a template and view. See [Figure 10-8](#).

Figure 10-8 Select a Template and View



11. In the wizard's **Title & Theme** window, enter a title (which appears as the portlet's title in the titlebar and the Administration Console) and description for the portlet. Select a theme from the **Portlet Theme** field and click **Next**. See Figure 10-9. The borderless theme presents your content with no background or border, so the content looks as if it is inline. If you choose the borderless theme, the Edit icon (discussed in step 12) appears only when you mouse over the top of the portlet.

Figure 10-9 Enter a Description for the Portlet and Pick a Theme



12. In the portlet's **Preview & Save** window, you can click **View Content** or **Edit Content**. Click **Save** to save your changes to this page and to shared portlets in the library. Click **Preview Changes in Portlet** to view the changes in the portlet. Only you can see this version of this

portlet. When previewing the portlet, you can click **Edit** to make changes to it. (If someone is authorized to make changes to your portlet and does so while you are previewing the portlet, you can no longer preview your version of the portlet. If this occurs, click **Continue** to see the newer version of the portlet.)

Tip: Clicking **Preview Changes in Portlet** puts the portlet into a "preview state". The portlet will remain in its preview state until you click **Cancel** or **Save** in the wizard. The preview state is maintained even if you log out or close your browser window. If you make changes in other steps of the wizard, you must click **Preview Changes in Portlet** to update the preview with your changes, or click **Save** to make the changes public.

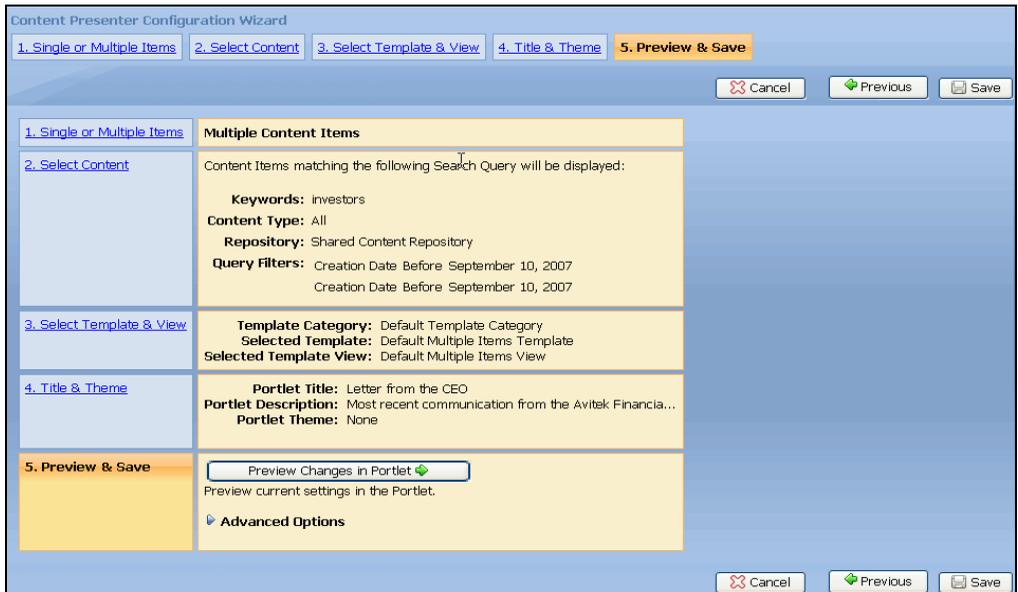
You can also click **Advanced Options** and select **Save to Current Page** (to save your changes to the current page only) or select **Save to Shared Portlet in Library**, to save changes to this shared portlet in the library. Saving changes at the library level globally affects everywhere the shared portlet was placed. You see this option enabled only if you clicked **Create New Shared Portlet**. When you click **Create New Shared Portlet**, you use these settings to create a new portlet in the library. The new portlet is available to all entitled users of this web application and can be placed on any page. The current page is updated to use this new portlet, rather than the current portlet. The new portlet also appears in the Administration Console in the `\Library\Portlets\` directory. Inheritance rules apply to shared portlets in the library. See the [Portal Development Guide](#) for more information on inheritance. [Figure 10-10](#) shows the Finish window.

Note: Content Presenter portlets that are configured on pages in the library will remain on the page in the library, even if the page has been customized on the desktop. The only way to make Content Presenter configurations local to a desktop is to add a configured or unconfigured Content Presenter portlet directly to the desktop, or customize the portlet on the desktop by changing or adding localizations to the portlet on the desktop using the Administration Console.

When you finish previewing the portlet, click **Return to Wizard** or click the **Edit** icon to return to this step in the wizard and make any additional changes. If you want to save your changes, click **Save**.

Tip: To be able to use the Create New Shared Portlet button in the Advanced Options, you must have edit rights on the page and delete rights on the portlet, in addition to the rights needed to configure a Content Presenter portlet.

Figure 10-10 Preview or Save the Portlet



The new content displays in the Content Presenter portlet. [Figure 10-11](#) shows an example of content that is an advertisement for a college savings account.

Figure 10-11 A Single Content Item Displayed in the Content Presenter Portlet



You can use a custom template or template view to change the look of the Content Presenter portlet. See [Displaying Content with the Content Presenter Portlet](#) in the *Content Management Guide* for instructions.

Tip: The Content Presenter portlet uses error logging to catch exceptions and displays an error message in the portlet if you have rights to configure the portlet. For example, you might receive an error message if content your portlet is referencing was deleted, or a template or view the portlet is using was removed. The error message instructs you to reconfigure the Content Presenter portlet to fix the errors. You must have edit and delete capabilities in order to configure the portlet.

This section also contains the following topics:

- [Changing How Much Content Appears in the Portlet](#)
- [Using Portlet Publishing to Expose a Content Presenter Portlet](#)

Changing How Much Content Appears in the Portlet

You can change the amount of content that a portlet displays. Use the Content Presenter Example to edit the Avitek - In the News portlet to display three specific press releases, rather than a list of all press releases.

Perform the following steps to change how much content appears in a portlet:

1. Follow the instructions in [“Starting the Content Presenter Example” on page 10-2](#) to log into the Content Presenter Example.
2. In the Avitek - In the News portlet in the Content Presenter Example, click **Edit** in the portlet’s title bar.
3. In the Content Presenter Configuration Wizard, click **2. Select Content**.
4. Select **Custom List of Specific Content Items** to display content from a list that you create.
5. Navigate through the repository tree to locate content in the Shared Content Repository folder and select the **HR > Press Releases** folder.
6. Select the AvitekFutureInvestorsProgram.htm press release and click **Add Item to List**. Repeat these steps for two additional press releases: DividendsAnnouncement.htm and NewRewardsCreditCard.htm. This step will control what displays in the portlet— rather than all press releases appearing, only these three will appear.
7. If you want user to see the DividendsAnnouncement press release first in the list, select the check box next to it and click **Move Up**.
8. Click **Next** to save your changes.

9. In the Select Template & View window, click **Next**.
10. In the Portlet Properties window, click **Next**.
11. In the Finish window, click **Preview Changes in Portlet**.
12. The three press releases now appear. Click **Return to Wizard**.
13. Click **Save**.

Using Portlet Publishing to Expose a Content Presenter Portlet

You must be entitled to use Portlet Publishing to expose a Content Presenter portlet. If you use Portlet Publishing to expose your Content Presenter portlet, some features are not available.

The following list describes features that are not part of a Content Presenter portlet exposed with Portlet Publishing:

- Advanced options are not available when you expose a Content Presenter portlet through Portlet Publishing. You can save the portlet's configuration only at its current location (for example, a desktop or a page in a library).
- There is no Title Bar, so you cannot click **Edit** in the portlet's title bar. You can edit the portlet configuration only by rolling your mouse over the top portion of the portlet, which enables the edit button on the top right of the portlet.
- Themes do not apply in portlets exposed through Portlet Publishing. Therefore, you cannot change the currently selected theme of the Content Presenter portlet.

Tip: When you use Portlet Publishing, work with only one Content Presenter portlet on a page at a time. If you work on more than one Content Presenter portlet on a page, an error appears. Close one of the Content Presenter wizards to continue working.

For more information on Portlet Publishing, see the [Portlet Guide](#).

Adding the Content Presenter Portlet

Adding a Third-Party Portlet

This chapter discusses special-purpose portlets that are provided by WebLogic Portal partner companies that you can easily incorporate into your portal.

This chapter includes these sections:

- [Using the Collaboration Portlets](#)
- [Third-Party Portlets](#)

Using the Collaboration Portlets

WebLogic Portal provides a set of portlets for adding collaborative features to your portal. You can use these collaboration portlets in any WebLogic Portal desktop.

Note: The Collaboration portlets will not operate correctly when desktop or portlet asynchronous mode is enabled. Async mode is not supported for Collaboration portlets. For information on portlet async mode, see [“Asynchronous Portlet Content Rendering” on page 7-13](#). For information on desktop async mode, see the *WebLogic Portal Portal Development Guide*.

This section includes these topics:

- [What Are Collaboration Portlets?](#)
- [Adding Collaboration Portlets To Your Portal](#)
- [Configuring Collaboration Portlets for a Shared View](#)
- [Using the Collaboration Portlets](#)

- [Using the Collaboration Portlet Source Code](#)

What Are Collaboration Portlets?

WebLogic Portal provides the following collaboration portlets that you can use in any WebLogic Portal desktop.

Note: *User portlets* are portlets that store data on a per-user basis. *Common area portlets* store data in a common location that can be viewed by all users. The Calendar, Address Book, and Tasks portlets are user portlets by default. In some cases, you might want to reconfigure them to be common area portlets. For example, you might want to configure a corporate events calendar where all users see the same data. See [“Configuring Collaboration Portlets for a Shared View”](#) on page 11-7 for details.

- **Calendar Portlet** – (User portlet) Lets you create and schedule appointments.
- **Mail Portlet** – (User portlet) Allows you send and receive personal e-mail. This portlet supports IMAP and POP.
- **Address Book Portlet** – (User portlet) Lets you view and manage names, addresses, phone numbers, e-mail addresses, and other information in a personal address book.
- **Tasks Portlet** – (User portlet) Allows you to create and track Community items or personal items on a To Do list.
- **Discussion Portlet** – (Common area portlet) Lets you post and monitor topics of interest.

Tip: The collaboration portlets are also available for use in communities, such as a GroupSpace Community. For detailed information on creating communities, see the [WebLogic Portal Communities Guide](#). A GroupSpace Community is a community created using the Workshop for WebLogic GroupSpace Template. For information on the WebLogic Portal GroupSpace Community, see the [WebLogic Portal GroupSpace Guide](#). The instructions in this chapter are intended for use of the collaboration portlets outside of a groupspace enabled community.

Adding Collaboration Portlets To Your Portal

This section explains how to add collaboration portlets to your portal and configure them properly. The basic steps are:

- [Step 1. Add Collaboration Facets](#)

- [Step 2: Add Collaboration Repository to Your Domain](#)
- [Step 3: Create a Role for Collaboration Portlet Users](#)
- [Step 4. \(Optional\) Configure a Repository](#)
- [Step 5. Entitle the Collaboration Data Repository](#)
- [Step 6. Add Users to the New Role](#)
- [Step 7. Configure the Collaboration Portlets](#)
- [Step 8. Add Collaboration Portlets to Your Desktop](#)

Step 1. Add Collaboration Facets

You must add the appropriate facets to both the portal EAR projects and the portal Web projects in which the collaboration portlets will be used.

1. Add the relevant collaboration portlet facets to your portal EAR project.
 - a. In the Navigator view, right-click your portal EAR project and choose **Properties**.
 - b. In the Properties view, select **Project Facets**, and click **Add/Remove Project Facets**.
 - c. In the Add/Remove window, expand **WebLogic Portal Collaboration** and select both **Collaboration Portlets Application Libraries** and **Collaboration API**.
 - d. Click **Finish**, then **OK**.
2. Add the Collaboration Portlets facet to your portal web project.
 - a. Perform the same sub-steps above, selecting the **WebLogic Portal Collaboration > Collaboration Portlets** facet.

After you add the collaboration portlet facets, collaboration portlets themselves must be configured properly, as explained in the following steps. After configuration, they are available to add to a portal desktop.

Step 2: Add Collaboration Repository to Your Domain

If you have not done so, you need to create or extend a domain to includes the Collaboration Repository components.

1. If you have an existing server and it is running, stop the server.

2. Start the Configuration Wizard. From the Windows Start menu, choose **Oracle Products > WebLogic Server 10.x > Tools > Configuration Wizard**.
3. In the Configuration Wizard, select **Create** for a new domain or **Extend** for an existing domain, and click **Next**.
4. If you selected **Create** in [step 3](#), select **WebLogic Portal Collaboration Repository** check box and click **Next**. If you are extending an existing domain, select the domain root directory, and click **Next**.
5. Complete the remaining wizard windows.
6. Restart the server.

Step 3: Create a Role for Collaboration Portlet Users

Users of the collaboration portlets must be entitled to use the repository in which collaboration data is stored. This section explains how to create a new user role.

1. Start the WebLogic Portal Administration Console and log in.
2. Create a new enterprise application-scoped visitor entitlement. To do this, select **Users, Groups, & Roles > Visitor Entitlement > Browse Roles**.
3. Set the role scope. In the **Browse Roles from** panel, click **Update** to bring up the Update Role Scope dialog. In the dialog, select **Enterprise Application Scope**, and click **Update**.
4. Select **Visitor Roles > Browse Roles > Create New Role**. Enter a name for the new role and save it.

Step 4. (Optional) Configure a Repository

Data generated by collaboration portlets is stored in a content repository. By default, collaboration portlets are configured to store data in the repository subfolder `/Communities_Repository/Collaboration`.

If you wish, you can use any WLP content repository for storing collaboration portlet data. Note that library services must be *disabled* for the repository. Collaboration portlet data is not supported for third party repositories, such as Documentum repositories. See the [Content Management Guide](#) for detailed information on content repositories. It is a good practice to create a subfolder in the repository in which to store the data, as explained in this section.

Tip: A general best practice is to create a custom repository for collaboration data. See [Configuring Additional WLP Repositories](#) in the *Content Management Guide* for details.

To create a subfolder in which to store collaboration portlet data, do the following:

1. Select **Content > Content Management**. In the Repository View, select your repository.
2. Click **Add Folder** and add a new folder to the repository of your choice.

In a later step, you will configure individual collaboration portlets to point to the repository folder of your choice, which is an option if the default location is not desirable.

Step 5. Entitle the Collaboration Data Repository

You must properly entitle the repository folder in which collaboration portlet data will be stored. Only entitled users can use the collaboration portlets.

1. Select the subfolder you created or targeted to store collaboration data.
2. Select the **Entitlements** tab for the subfolder.
3. Click **Add Role** and add the new user role you created for the collaboration portlets. Entitle the role with the capabilities Create, View, Update, and Delete.

Step 6. Add Users to the New Role

You must add any users who will use the collaboration portlets to the new role. Select the role and click **Add Users to Role**. Use the dialog to add users to the role.

Tip: Be sure to add any new users to the role if you want them to use the collaboration portlets. To create new users, select **Users, Groups, & Roles > User Management**. After you create a new user, add it to the role.

Step 7. Configure the Collaboration Portlets

Configure the collaboration portlets so that they are aware of the repository that you configured. To do this, you edit certain portlet preferences.

1. Select **Portal > Portal Management**.
2. Expand the **Portal Resources > Library > Portlets** folder.

3. For each collaboration portlet that you wish to use, do the following:
 - a. Select a portlet to configure. For example, click **Discussion** to configure the Discussion portlet.
 - b. Click **Portlet Preferences**.
 - c. Edit **collaboration.personal_repository.path** and set its value to the designated collaboration data folder in your repository. For example, if you created a folder named MyCollaborationData in the repository called MyCollaborationRepository, set the value to: /MyCollaborationRepository/MyCollaborationData.
 - d. Edit **collaboration.personal_repository.name** and set its value to the name of the repository you are using for collaboration data. For example, if you are using a repository called MyCollaborationRepository, set the value to MyCollaborationRepository.

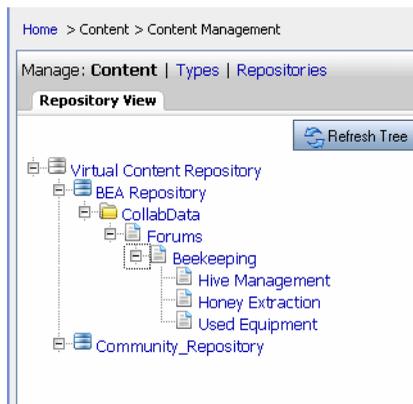
Step 8. Add Collaboration Portlets to Your Desktop

Now that you have configured your collaboration portlets, you can add them to a desktop.

Note: Collaboration portlets only work if the user is authenticated. Your desktop must include a login portlet. For more information on authentication, see the [Security Guide](#).

If you configured everything properly, authorized users can access the collaboration portlets after logging in. Folders will be created in the collaboration repository as they are needed. For example, [Figure 11-1](#) shows the repository structure for an example discussion forum on beekeeping.

Figure 11-1 Repository Structure for a Discussion Forum



Configuring Collaboration Portlets for a Shared View

This section explains how to reconfigure user portlets to be common area portlets. User portlets restrict the portlet's data to individual users, while common area portlets allow entitled users to share a the same view of the portlet's data.

Overview of User and Common Area Portlets

Collaboration portlets fall into two categories: *common area portlets* and *user portlets*: Typically, common area portlets are recommended for use cases where all users need to share the same view of the portlet's data. For example, you could create a calendar that displays corporate events to all users. In this case, you would need to configure the Calendar portlet (which is a user portlet by default) to be a common area portlet.

- Common area portlets write data to a shared repository that is accessible to all entitled users. By default, the Discussion portlet is a common area portlet.
- User portlets write data to a user-specific repository location that is accessible only to the currently logged in user. A user portlet's repository location is based on the root repository location specified by the portlet preference (see [“Step 7. Configure the Collaboration Portlets” on page 11-5](#)). For example, if the Calendar portlet preference is set to be `/MyRepository/Collaboration/calendar`, and if the Calendar portlet is configured as a user portlet, it will write to the location `/MyRepository/Collaboration/calendar/<username>`. The Calendar, Address Book, Tasks, and Mail portlets are user portlets by default.

Configuring a Common Area Portlet

This section explains how to reconfigure user portlets to be common area portlets. Note that by default, the Calendar, Address Book, Tasks, and Mail portlets are user portlets.

Tip: Because mail is usually intended to be used by specific users rather than shared among many users, it is typically not necessary to reconfigure the Mail portlet to be a common area portlet.

1. Copy the portlets you want to configure to your local project. To do this:
 - a. Open the Merged Projects View in Workshop for WebLogic. (To open the Merged Projects view, select **Window > Show View > Merged Projects**.)

- b. Right-click each portlet (located in the portlets/collaboration folder) and select **Copy to Project**. See “[Portlets in J2EE Shared Libraries](#)” on page 5-3 for information on the Copy to Project feature.
2. Rename each copied portlet. For example, change Tasks.portlet to Tasks-Team.portlet.
3. In a text editor, open each .portlet file and change the definition label and title. For example:

Before:

```
<netuix:portlet definitionLabel="task" title="My Task"
  lafDependenciesUri="/portlets/collaboration/collaboration.dependencies">
```

After:

```
<netuix:portlet definitionLabel="task_team" title="Team Task"
  lafDependenciesUri="/portlets/collaboration/collaboration.dependencies">
```

4. Also in the text editor, for each portlet, change the <netuix:meta> tag containing the AccountListenerImpl to use the CmAccountListener instead of the PersonalAccountListener. For example:

Before:

```
<netuix:meta name="collaboration.portlet.AccountListenerImpl"
  content="portlets.collaboration.common.c11n.PersonalAccountListener"/>
```

After:

```
<netuix:meta name="collaboration.portlet.AccountListenerImpl"
  content="portlets.collaboration.common.c11n.CmAccountListener"/>
```

5. Save each .portlet file.

Using the Collaboration Portlets

For detailed information on how to use the collaboration portlets, see the “Using the GroupSpace Portlets” chapter of the [WebLogic Portal GroupSpace Guide](#). All of the collaboration portlets listed previously in this section are described in that chapter.

Using the Collaboration Portlet Source Code

Source code for the collaboration portlets is available to WebLogic Portal developers, as explained in this section.

Copying the Source Code to Your Project

To use the source code, you must first copy it from a J2EE Shared Library to your workspace.

Source code for the collaboration portlets is located in the J2EE Shared Library `wlp-collab-portlets-web-lib`. To use this source code, you need to copy it from the shared library to your project workspace. See [“Portlets in J2EE Shared Libraries” on page 5-3](#) for information on the Copy to Project feature.

Java source code for the collaboration portlets is copied to `WEB-INF/src/portlets`. Javadoc for the collaboration portlet code is copied to `WEB-INF/src/javadoc.zip`.

Source Code Disclaimers

If you modify any of the source code for the collaboration portlets, be aware of the following disclaimers:

- If you modify the source code and discover a bug, you must either reproduce the problem using the *original* collaboration portlet code or provide a simple code sample that illustrates that a bug exists in the WebLogic Portal API bug.
- If you change the original copy of the collaboration portlet source code and later apply a software patch to WebLogic Portal, you must copy the updated source code from its library module to your workspace and reapply the changes you made to the original source code.

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 come 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 proprietary search APIs were deprecated; we recommend that you use Autonomy APIs to implement search capabilities.

For more information about Autonomy, see [Integrating Search](#).

Documentum Portlets

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

Oracle Communication and Mobility Server provides a standards-based, non-proprietary environment that extends Oracle 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 Oracle Communication and Mobility Server and how to use it with WebLogic Portal, see the product documentation on the [e-docs web site](#).

Adding a Third-Party Portlet

Working With JSF Portlets

This chapter provides an in-depth discussion on procedures and best practices for developing and configuring JSF portlets.

This chapter includes the following sections:

- [Overview](#)
- [Configuring JSF Within Weblogic Portal](#)
- [Creating JSF Portlets](#)
- [Native Bridge Architecture](#)
- [Understanding WLP and JSF Rendering Life Cycles](#)
- [Understanding Scopes and JSF Portlets](#)
- [State Sharing Patterns](#)
- [Using Common WLP Features With JSF Portlets](#)
- [Portal Container Features and JSF Portlets](#)
- [Understanding Navigation](#)
- [Navigation Within a Portal Environment](#)
- [Interportlet Communication with JSF Portlets](#)
- [Namespacing](#)

- [Using Custom JavaScript in JSF Portlets](#)
- [Ajax Enablement](#)
- [Localizing JSF Portlets](#)
- [Preparing JSF Portlets for Production](#)
- [Tips for Logging, Iterative Development, and Debugging of JSF Portlets](#)
- [Consolidated List of Best Practices](#)

Note: For information about JSF portlet development, see [Appendix B, “JSF Portlet Development.”](#)

Overview

Oracle WebLogic Portal has supported the use of JSF portlets starting with WLP 9.2, and this support has been enhanced through the current release. JSF is supported as a specific portlet type within WebLogic Portal. Portlets implemented with JSF can leverage all of the powerful features of WLP.

This chapter provides a developer with a comprehensive guide for building JSF portlets in Oracle WebLogic Portal 10.3.0. It covers a wide variety of portlet development topics necessary to know to build JSF portlets.

Configuring JSF Within Weblogic Portal

This section discusses how to configure JSF within WLP. These configuration settings are specified in files such as `web.xml` and `weblogic.xml`. All of these files are scoped to the entire Web Project. Therefore, Faces configuration is also scoped to the entire web application.

This section contains the following topics:

- [JSF Library Modules in WebLogic Server](#)
- [Installing the JSF Libraries into a Portal Web Project](#)
- [Configuring JSF 1.2 in WLP](#)
- [Creating JSF Portlets](#)
- [JSF Configuration Settings](#)

JSF Library Modules in WebLogic Server

WebLogic Server provides a packaging feature called library modules that package up one or more jar files as a deployable feature. JSF support for a web application is one such usage of the library module packaging feature.

[Table 12-1](#) lists the JSF library modules included in WebLogic Server 10gR3 (and therefore also WebLogic Portal 10gR3). This list is obtainable via the Manage WebLogic Shared Libraries link on the JSF library configuration dialog, or by inspecting the `config.xml` file for your domain.

Table 12-1 Supported JSF Implementations for WebLogic Server 10gR3

Library Module Name	Implementation	Version	Supported in WLP?
Jsf	Sun Reference	1.2.3.2	Yes, caveats
Jsf-ri	Sun Reference	1.1.1	Yes
Jsf-myfaces	MyFaces	1.1.3	No
Jsf-myfaces	MyFaces	1.1.1	No

For a Web Project, including WebLogic Portal Web Projects, you must choose just one version of JSF to use amongst the supported set. For WLP 10gR3, only the Sun RI implementations are officially supported. While it is possible to manually create a new JSF library module for a different JSF implementation, WebLogic Portal only officially supports the set provided in the installer. It is also possible to copy JSF implementation jar files directly into the `WEB-INF/lib` folder. This also is not officially supported with WLP.

The JSF portlet support in WebLogic Portal uses the same integration code regardless of the JSF implementation in use. However, Oracle has found the Sun RI to work best within WLP, and there are known issues with using MyFaces. Therefore, only the Sun RI is supported.

Caution: Workshop for WebLogic will by default configure an unsupported MyFaces JSF implementation. See the next section for instructions on changing it to Sun RI.

Installing the JSF Libraries into a Portal Web Project

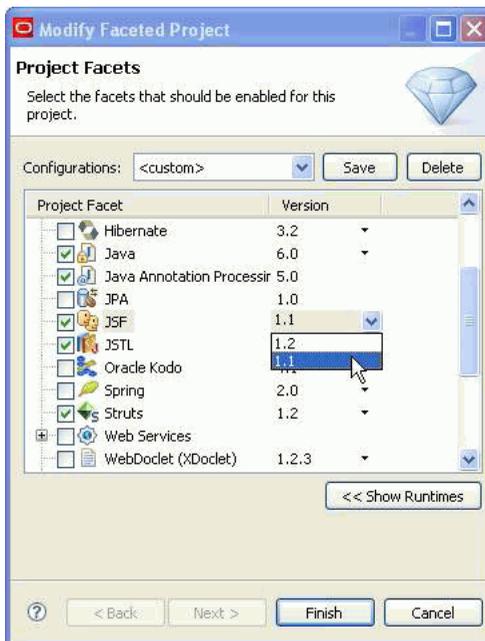
For a WebLogic Portal Web Project to support JSF portlets, JSF itself must be installed into the project. JSF support is not enabled by default.

A plugin called Web Tools Project (WTP) is available for the Eclipse IDE which aids in building web applications. To help developers manage their web application libraries, WTP provides a feature called facets. A facet is a feature that is provisioned in a web application, like JSF for example. When a facet is added to a Web Project, WTP will add in the necessary libraries, files, and configuration artifacts. When deploying to WebLogic Server, a Facet often will configure the Web Project to use one or more library modules.

There are several paths to installing JSF in a Portal Web Project:

- Enabling the JSF facet in Workshop for WebLogic when creating the project. The facet selection dialog appears during the web project creation process.(Figure 12-1)
- Adding the JSF facet in Workshop for WebLogic after creating the project. Access the dialog via **Project > Properties > Project Facets > Modify Project**.

Figure 12-1 Facet Selection Dialog



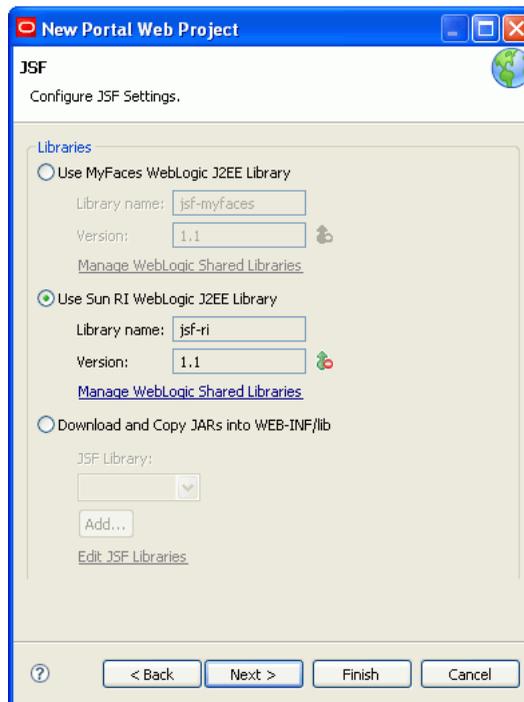
Once you have navigated to the facet selection dialog, follow these steps:

1. Click the JSF facet checkbox to select it.

2. Choose the desired version of JSF by clicking on the dropdown in the Version column. Selecting JSF 1.1 will eliminate the dependency errors. For the steps to configure JSF 1.2, see [Configuring JSF 1.2 in WLP](#).
3. Click **Next**. Do *not* click **Finish**. **Caution:** Clicking **Finish** on the facets selection dialog will result in an unsupported JSF implementation being deployed into the Web Project.
4. You will be prompted to provide the specific library module version for some of the facets, including JSF. Click **Next** until you reach the JSF panel. On the JSF panel, you must choose the Sun RI, as shown in [Figure 12-2](#).
5. Click **Finish**.

The wizard will update your web project descriptors to include the appropriate JSF library module (see `WEB-INF/weblogic.xml`). It will also update `web.xml` with default Faces configuration settings, and insert a new `faces-config.xml` into your project.

Figure 12-2 New Portal Web Project Dialog



Caution:

- It is not officially supported for WebLogic Portal to choose the first or third options in the library module configuration dialog. You must select Sun RI.
- Click on the **Manage WebLogic Shared Libraries** link to get a list of the installed library modules.

Configuring JSF 1.2 in WLP

Of the four JSF implementations installed with WebLogic Server, only one implements JSF 1.2. That implementation is the Sun RI. Normally, it would suffice to select the 1.2 version (which is actually the default) in the facet selection dialog. However, due to a dependency issue, configuring JSF 1.2 for a Portal Web Project using the facet selection dialog is not currently possible.

The core issue is that the Apache Beehive web framework has an integration with JSF (see [Integrating Apache Beehive Pageflow Controller](#)) which does not work correctly with JSF 1.2. In all releases through WLP 10gR3, Portal Web Projects have a hard dependency on Apache Beehive. Therefore, the facet selection dialog does not allow JSF 1.2 to be enabled.

However, for Portal Web Projects that do not use the Apache Beehive integration, this is an unnecessary limitation. Oracle offers limited support for JSF 1.2 when the Beehive Page Flow integration is not being used in the web application. Follow these instructions to work around the issue.

1. Launch Workshop for WebLogic.
2. Create a Portal Web Project. Add the JSF 1.1 facet, which will populate the project with the default JSF files.
3. Navigate to **Project > Properties > Java Build Path** and select the **Libraries** tab.
4. Remove the JSF 1.1 library from the list.
5. Click **Add Library** and then **WebLogic Shared Library**.
6. Choose the JSF 1.2 library from the list, and click **OK**.
7. Open `WEB-INF/weblogic.xml`.
8. Remove the entry for the JSF 1.1 library module, if it exists.
9. Add an entry for the JSF 1.2 library, as shown in [Listing 12-1](#).

10. Update the entry for JSTL to take version 1.2, not 1.1.
11. Remove the `PageFlowApplicationFactory` configuration artifact from `WEB-INF/facesconfig.xml` (see [Installing the JSF Libraries into a Portal Web Project](#)). This avoids a problem that occurs with using Apache Beehive with Sun RI 1.2.3.2.
12. Clean and rebuild, then redeploy.

Listing 12-1 The `weblogic.xml` Entry for Using JSF 1.2

```
<wls:library-ref>
    <wls:library-name>jsf</wls:library-name>
    <wls:specification-version>1.2</wls:specification-version>
    <wls:implementation-version>1.2.3.2</wls:implementation-version>
</wls:library-ref>
```

Creating JSF Portlets

Once Faces is configured in WLP, then a JSF portlet can be created by the Portlet wizard in Oracle Enterprise Pack for Eclipse. See [Chapter 5, “Building Portlets”](#) to learn how to create your first JSF portlet.

JSF Configuration Settings

In general, WebLogic Portal is agnostic to the various JSF configuration settings that can be manipulated. This section discusses how various JSF configuration choices are manifested when JSF is running within WebLogic Portal.

Client or Server State Storage

JSF is a stateful web framework - it maintains state for a user in between HTTP requests. JSF offers two state management options for JSF applications: `client` and `server`. It is configured in `web.xml` with the `javax.faces.STATE_SAVING_METHOD` context parameter.

While client state management with the JSF portlet native bridge works in most cases, Oracle only supports the server state management option. There are known issues, particularly when interportlet communication (IPC) is at work on the page, where client state management will cause the wrong portlet JSF view to be rendered. This is due to the fact that a single HTTP request

may cause multiple JSF portlets to be invoked if IPC is in use. The HTTP request carries only the state of the targeted JSF portlet, and thus the listening JSF portlet(s) will use the wrong client state during restoration. For more information about IPC, see [Interportlet Communication with JSF Portlets](#).

Also, there are several drawbacks with using client state saving in general:

- Increased network bandwidth costs
- Increased processing costs per request to re-establish the view state
- Security concerns, as the user may tamper with the client side state (MyFaces has an encryption feature for this)

Oracle therefore only supports the use of server state saving with WebLogic Portal. This mode works properly with IPC. Also, the WLP portal framework itself retains portal state for each active user session on the server, and thus the server state option for JSF portlets is more consistent with the behavior of the WLP framework.

Caution: Unfortunately, Workshop configures the `STATE_SAVING_METHOD` value as **client** by default. Developers need to manually change this value in `web.xml` to **server** after creating the Portal Web Project ([Listing 12-2](#)).

Listing 12-2 Supported State Storage Configuration in web.xml

```
<context-param>
<param-name>javax.faces.STATE_SAVING_METHOD</param-name>
<param-value>server</param-value>
</context-param>
```

Prefix or Suffix Servlet Mapping

There are two servlet mapping schemes for the Faces servlet.

- Suffix - maps the Faces servlet to a file suffix, such as `*.jsf` or `*.faces`
- Prefix - maps the Faces servlet to a URL path, such as `/faces/*`

When deploying JSF portlets, the JSF portlet container invokes the Faces infrastructure directly. Therefore, WLP does not rely on any servlet mapping being configured. If you wish to have the

portlets also available to direct access (i.e. not as a portlet), you can choose to use either JSF suffix or prefix mapping as with a normal JSF web application.

[Listing 12-3](#) shows the prefix servlet mapping configuration in `web.xml`.

Listing 12-3 Prefix Servlet Mapping Configuration in `web.xml`

```
<servlet-mapping>
    <servlet-name>faces</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>faces</servlet-name>
    <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

Other Settings

The `STATE_SAVING_METHOD` and servlet mapping settings are the most important, but they are far from the only settings available. The Sun RI currently has dozens of extra settings that can be configured. In general, these settings are completely independent of WLP, and so can be set at your discretion. However, due to the number of implementations and number of settings in each, WLP does not certify the integration with the non-default options.

[Listing 12-4](#) shows a partial list of settings in the Sun RI v1.2.8 (see `WebConfiguration.java` in the source code).

Listing 12-4 Partial Listing of Additional Configuration Options for the Sun RI

```
com.sun.faces.managedBeanFactoryDecoratorClass
javax.faces.CONFIG_FILES
javax.faces.LIFECYCLE_ID
com.sun.faces.numberOfViewsInSession
```

```
com.sun.faces.numberOfLogicalViews
com.sun.faces.injectionProvider,
com.sun.faces.serializationProvider
com.sun.faces.responseBufferSize
com.sun.faces.clientStateWriteBufferSize,
com.sun.faces.expressionFactory
com.sun.faces.clientStateTimeout
com.sun.faces.compressViewState
com.sun.faces.compressJavaScript
com.sun.faces.externalizeJavaScript
```

Native Bridge Architecture

This section provides an architectural overview of how WebLogic Portal's JSF native portlet bridge operates. It shows the relationship between the different components in the implementation. This explanation is intended as an overview to help you understand how to best architect and debug your portlet when using this bridge.

This section contains the following topics:

- [Container Architecture Overview](#)
- [Container Interactions](#)

Container Architecture Overview

[Figure 12-3](#) depicts the logical relationship of the containers involved: Servlet, WLP Rendering, and JSF portlet containers. The containers have these properties:

Servlet Container

- Is defined by the Servlet specification.
- Represents a classloader scope, and so all of the containers within it exist within a singleclassloader (and thus JVM).
- The HttpSession is scoped to this container.

WebLogic Portal Rendering Container

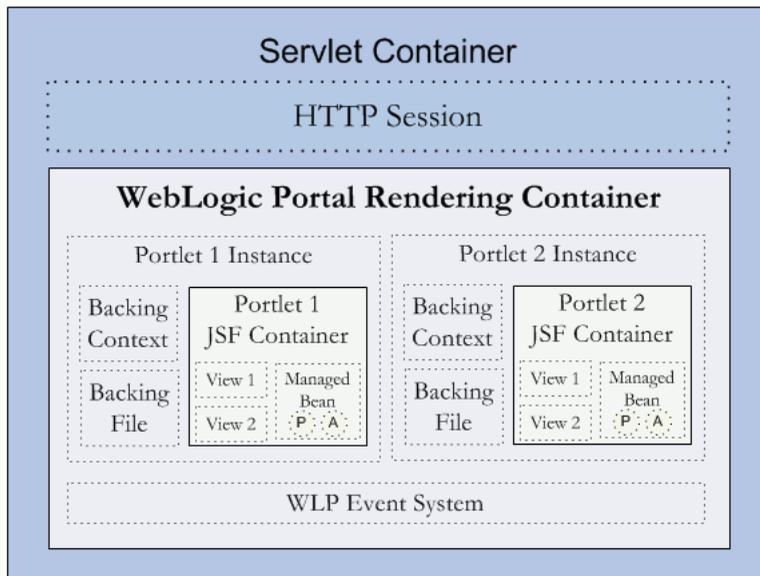
- Is provided by WebLogic Portal.
- Is responsible for rendering the WLP control tree, which describes the structure of the portal.

Portlet JSF Container

- Is provided in WLP by the JSF native portlet bridge.
- One instance of this container exists for each portlet instance on the page.
- The JSF implementation (Sun RI) runs within this container.

Figure 12-3 shows a logical view of how the containers are related. Solid lines depict boundaries between containers, and dotted lines demark logical elements and objects within a container.

Figure 12-3 Logical Container Architecture of WebLogic Portal and JSF Portlet Native Bridge



Note: This discussion applies specifically to local JSF portlets. A JSF portlet that is remote (i.e. via WSRP) has a different relationship to the parent containers.

Container Interactions

The following are some general principles to understand regarding interactions between containers:

- Custom code deployed in a container can access objects of parent containers. However, as will be covered in the [Understanding WLP and JSF Rendering Life Cycles](#) section, not all objects of the parent container are available at all times.
- Custom code deployed in a container cannot invoke objects of child or sibling containers. For example, a portlet backing file deployed in the WLP container cannot access objects in the portlet's JSF container.
- The HttpSession is available to code in all containers, and is thus an available lowest common denominator approach for passing objects between containers.

Understanding WLP and JSF Rendering Life Cycles

This section explains WLP and JSF rendering life cycles. It contains the following topics:

- [WLP and JSF Life Cycles](#)
- [Invocation Order of WLP and JSF Life Cycle Methods](#)
- [Accessing WLP Context Objects from JSF Managed Beans](#)

WLP and JSF Life Cycles

Both WebLogic Portal and JSF frameworks support the concept of component trees that define the rendering of the HTML page. Both also rely on the concept of rendering life cycles. Each component tree is walked multiple times during the execution of the request. Each traversal is called a life cycle or phase.

When developing JSF portlets, it is helpful to understand how those life cycles interact. For more information on the phases of the portal life cycle, see "Understanding Portal Development" in the see the [Portal Development Guide](#).

Invocation Order of WLP and JSF Life Cycle Methods

The following represents the merged life cycle execution order across the WLP and JSF containers:

- PortalContainer:init

- PortletBackingFile.init
- PortalContainer:handlePostBackData (if the request is a portal postback)
 - PortletBackingFile.handlePostBackData for all portlets on active pages
 - Last chance to invoke PortletBackingContext.sendRedirect()
 - Then, if the JSF portlet is the target of the request:
 - JSFContainer:RestoreView
 - JSFContainer:ApplyRequestValues
 - JSFContainer:ProcessValidations
 - JSFContainer:UpdateModelValues
 - JSFContainer:InvokeApplication
 - JSFContainer:invoke Action Listeners
 - JSFContainer:invoke Action method
- PortalContainer:raiseEvents
- PortalContainer:preRender
 - PortletBackingFile.preRender
- PortalContainer:render
 - JSFContainer:RestoreView (only on first render of portlet instance)
 - JSFContainer:RenderResponse

Note:

- It is important to see that the JSF action/action listeners are invoked before the Portal Framework's raiseEvents life cycle. This enables a JSF portlet to raise an event that changes the Portal Framework control tree, such as activating a different page. For details on IPC, see [Chapter 9, “Local Interportlet Communication”](#).
- FacesContext.getCurrentInstance() returns the context only while the JSFContainer is in scope.

Accessing WLP Context Objects from JSF Managed Beans

To enable portlets to programmatically interact with the portal framework, a set of context objects is available.

[Table 12-2](#) shows what portlet context objects are in scope for different managed bean methods for different JSF life cycles. This chart is useful when implementing a managed bean that needs to obtain WLP context information. The key point is that property getters and setters need to be coded so that they work properly with either context object.

Table 12-2 Scope of Portlet Context Objects

JSF Life Cycle	Managed Bean Method	Portlet Backing Context	Portlet Presentation Context	Portal Use Case
PROCESS_VALIDATIONS	get property	Yes	No	Portlet receives a postback, input is being validated.
PROCESS_VALIDATIONS	set property	Yes	No	Portlet receives a postback, input is being validated.
UPDATE_MODEL	set property	Yes	No	Portlet receives a postback, input has been validated.
INVOKE_APPLICATION	action method	Yes	No	Portlet is the target of a postback.
RENDER_RESPONSE	get property	No	Yes	Portlet is being rendered.
RENDER_RESPONSE	set property	No	Yes	Portlet is being rendered.

Understanding Scopes and JSF Portlets

This section covers several scoping topics that apply to JSF portlets.

- [Conceptual Scopes for Standard JSF Applications](#)
- [Conceptual Scopes for Portal Applications](#)
- [Implementation Patterns for Portal Scopes](#)

Conceptual Scopes for Standard JSF Applications

The standard JSF scopes are interpreted differently in a portal environment. This section discusses the differences.

JSF Standard Scopes

JSF managed beans have well-defined scopes as defined in the JSF specification. The JSF 1.1 and 1.2 specifications provides three scopes for managed beans:

- Application - Bean state is accessible by all users in the web application.
- Session - Bean state is accessible to any view for the given user, across the life span of all requests within the session.
- Request - Bean state is accessible for the duration of a single request.

In addition to these, several additional scopes exist. Specifically, JSF 2.0 adds a View scope, and many web frameworks provide a Pageflow scope, as described below.

View Scope

View scope has been added to the JSF 2.0 specification. It enables managed beans to be attached to a specific view across multiple HTTP requests. Once a user navigates to a different view, the bean state is destroyed. This is a helpful pattern for scoping state to a single instance of a view for a user.

Pageflow/Conversation Scope

A Pageflow (also called a conversation) is a subset of the views and controller logic within a web application that pertains to a logical task or business process. Multiple Pageflows can exist within a web application, and each one usually carries state that should only be scoped to that Pageflow. With Pageflow scope, managed bean state is accessible across the life span of all requests within the session, limited to the time in which a user is interacting with the set of views within that Pageflow.

Conceptual Scopes for Portal Applications

Because of the composite nature of a portal user interface, there are more conceptual scopes for portals than for standard JSF applications.

The list of portal scopes includes:

- Application - Bean state is accessible by all users in the web application.
- Global Session - Bean state is accessible to any portlet for the given user, across the life span of all requests within the web application.
- Portlet Group Session - Bean state is accessible to any view within a group of portlet instances or definitions for the given user, across the life span of all requests within the web application. This use case is important for interportlet communication.
- Portlet Instance Session - Bean state is accessible to any view within a single portlet instance for the given user, across the life span of all requests within the web application.
- Pageflow - Bean state is accessible to any view within a Pageflow within a single portlet instance for the given user, across the life span of all requests within the Pageflow.
- View - Bean state is accessible for as long as the user is interacting with the current view across multiple requests within a single portlet instance. And, if the user is interacting with another portlet, the bean state is retained.
- Portal Aware Request - Bean state is accessible with the portlet instance for the duration of a single request. And, if the user is interacting with another portlet instance, the bean state is retained until the next request in which the user interacts with the portlet.

Implementation Patterns for Portal Scopes

[Table 12-3](#) describes how the standard JSF scopes map to the WLP scopes, and how the unrepresented JSF scopes are supported. Details about these implementation strategies are explained in the following sections.

Table 12-3 Managed Bean Scope Implementation Strategies

Portal Managed Bean Scope	Implementation Strategy for JSF Portlets
Application	faces-config.xml scope = application
Global Session	faces-config.xml scope = session, plus custom code
Portlet Group Session	faces-config.xml scope = session, plus custom code
Portlet Instance Session	faces-config.xml scope = session
Pageflow	Use an alternate navigation controller

Table 12-3 (Continued) Managed Bean Scope Implementation Strategies

Portal Managed Bean Scope	Implementation Strategy for JSF Portlets
View	Supported with JSF 2.0
Portal Aware Request	faces-config.xml scope = request

Reinterpretation of the JSF Session and Request Scopes

[Table 12-4](#) compares JSF managed bean scoping levels between a JSF application and a WLP JSF portlet.

Table 12-4 Comparison of Scoping Levels

Faces-Config.xml Scope Label	Conceptual Scope for JSF Application	Conceptual Scope for JSF Portlet
Application	Application	Application
Session	Global Session	Portlet Instance Session
Request	HttpRequest	Portal Aware Request

Because a managed bean declared with session scope in `faces-config.xml` is interpreted as Portlet Instance Session scoped with the portlet bridge, it is possible to put multiple instances of that portlet on a page and not have conflicts. Each portlet instance that uses the managed bean will be provisioned with a distinct instance of the bean.

Also, the different interpretation of request scope prevents a JSF portlet from breaking if the user interacts with a second portlet while interacting with the first JSF portlet.

Pageflow Scope

The standard JSF navigation controller does not support the notion of a Pageflow scope. However, other controllers can be plugged into the JSF runtime to provide such a capability. For example, you can integrate an Apache Beehive Page Flow controller. For details, see [Integrating Apache Beehive Pageflow Controller](#).

Global Session and Portlet Group Session Scopes

The remaining scopes for managed beans cannot be expressed in `faces-config.xml` alone. However, using code patterns involving the `HttpSession`, the remaining scopes can be achieved. For details, see section “[State Sharing Patterns.](#)”

State Sharing Patterns

This section contains the following topics:

- [State Sharing Concepts](#)
- [HttpSession Versus HttpServletRequest](#)
- [Base Code for HttpSession Patterns](#)
- [Single Portlet Pattern](#)
- [Multiple Portlet Patterns](#)

State Sharing Concepts

JSF managed beans are intended to be the storage containers for application state within a JSF application. In general, this works well even within a portal environment. However, this standard JSF pattern is not sufficient. There are cases where state needs to be shared with something outside of the JSF portlet. For example:

- The portlet instance's backing file.
- A different portlet instance's JSF managed bean.
- A portlet instance in a remote servlet container via WSRP.
- A non-portal object, such as a servlet or servlet filter.

But there are limitations that must be heeded when working within the JSF container:

- A JSF managed bean may not invoke any method on any portlet instance's backing file, including its own.
- A JSF managed bean may not invoke any method on a JSF managed bean in another portlet instance.
- A portlet backing file may not invoke any method on any portlet instance's JSF managed bean.

This section discusses patterns that work despite these limitations.

HttpSession Versus HttpServletRequest

Some of the patterns in this section make use of the HttpSession. They set state into the HttpSession as attributes. The first issue to cover is why the HttpSession? In some cases is the HttpServletRequest a better place to store this state? While in certain cases the HttpServletRequest is suitable, in general it is a best practice to use the HttpSession rather than HttpServletRequest. This section explains why.

Store State in the HttpSession

In short, the use of the HttpSession is preferred because of simplicity. In a portal environment, the lifecycle of the request is not always straightforward.

- WSRP uses two requests – When a user interacts with a portlet that is being consumed using WSRP, the handling of that interaction and the render of the portlet can occur over two requests. Therefore, attributes set into the request may not be around as long as they need to be.
- Scoped requests – When executing code within a portlet, that portlet often does not have access to the actual HttpServletRequest. Usually, it is a scoped object. Attributes set into a scoped request are not visible to other portlets. This makes sharing state between portlets not feasible using the request.

Drawbacks of Using the HttpSession

Use of the HttpSession does not come for free. There are several drawbacks to be aware of:

- Must be Serializable – Attributes set into the HttpSession must be Serializable. Not all objects are easily Serializable, so this may be an issue.
- Will be replicated – The reason HttpSession attributes need to be Serializable is for session replication within a cluster. WebLogic Server distributes or stores the attributes by serializing the attributes. Adding more attributes to the HttpSession creates higher overhead for the replication facility.
- Removing the attribute – If an attribute is appropriate only for the current request, the HttpSession attribute must be removed by the managed bean after it is finished with it.
- Multiple portlets – When used for multiple portlet patterns, the approach is fragile. This is discussed in more detail in [Multiple Portlet Patterns](#).

A possible solution to the first two of these drawbacks is to mark some or all of the stored state as transient. Transient objects in Java are not serialized, and so it can be used to mitigate both of the issues.

Base Code for HttpSession Patterns

Some of the patterns below rely on scoping data in the HttpSession. The best approach is to provide a namespace for the attribute name used to set the attribute into the session. The patterns differ in how that namespace is computed. Since these patterns share common code aside from the namespace generation, the following base code is assumed for each of the patterns.

JSFPortletHelper

The code samples below rely on a common class called JSFPortletHelper. It provides a number of helper methods that are useful with a managed bean used in a JSF portlet. For source code, see [The JSFPortletHelper Class](#).

SearchBeanBase (For HttpSession Patterns)

[Listing 12-5](#) shows the code that serves as the base managed bean for illustrating the HttpSession based patterns.

Listing 12-5 The Base Managed Bean for the Examples

```
public abstract class SearchBeanBase {
    // Gets the search string from the HttpSession
    public String getSearchText() {
        HttpSession session = JSFPortletHelper.getSession();
        String namespace = getAttributeName();
        return
        (String)session.getAttribute(namespace+"_searchText");
    }
    // Sets the search string into the HttpSession
    public void setSearchText(String search) {
        HttpSession session = JSFPortletHelper.getSession();
```

```

        String namespace = getAttributeNameSpace();
        session.setAttribute(namespace+"_searchText", search);
    }
    // Each pattern below implements this method differently
    abstract public String getAttributeNameSpace();
}

```

Single Portlet Pattern

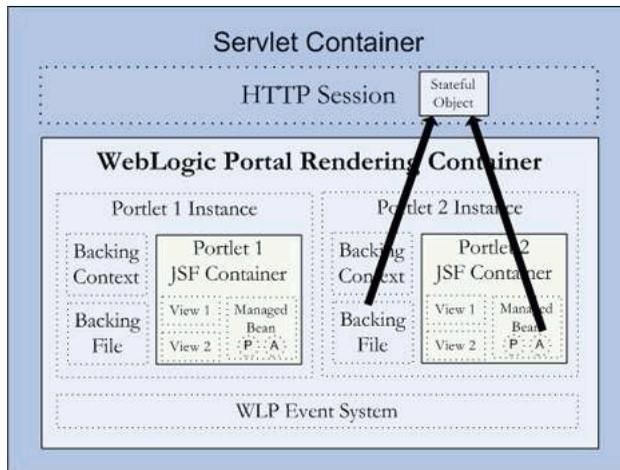
When sharing state amongst components of a single portlet instance, the pattern is straightforward. There is only one pattern necessary.

Portlet Instance Session Scoped Data (uses HttpSession)

In this pattern, the state is to be shared only with other components affiliated with the portlet instance. The state is set into the HttpSession. The best namespace for this use case is the portlet's instance label. This pattern is most often used when sharing data between a portlet's JSF managed bean and the portlet's backing file. Both of those components have easy access to the portlet instance label via the PortletBackingContext or PortletPresentationContext objects, which provide access to the instance label.

[Figure 12-4](#) shows an illustration of how a JSF managed bean and WLP backing file from the same portlet instance can share a stateful object.

Figure 12-4 JSF Managed Bean and WLP Backing File from the Same Portlet Instance Sharing a Stateful Object



The namespace is based simply off of the portlet's instance label. [Listing 12-6](#) shows the code that gets the namespace for the `HttpSession` attribute using the instance label.

Listing 12-6 Getting Namespace for the `HttpSession` Attribute using the Instance Label

```
public class SearchBean extends SearchBeanBase {
    public String getAttributeNamespace() {
        return JSFPortletHelper.getInstanceLabel();
    }
}
```

Multiple Portlet Patterns

These patterns support cases in which multiple portlets are sharing state. Several patterns are offered. Two use the `HttpSession` for state sharing, and one uses the WLP event mechanism. There are benefits and drawbacks to each:

Using the `HttpSession` (not recommended)

- Benefit – simple to implement
- Drawback – the order in which the portlets get and set state into the `HttpSession` will depend in some cases on the layout of the portal. Since users can move portlets on the page, the pattern may not work reliably.
- Drawback – it will break if one of the portlets is moved to a remote server via WSRP

Using Events (best practice)

- Benefit – is layout independent
- Benefit – will work with WSRP
- Drawback – incurs greater overhead
- Drawback – may get complicated to implement if a single request requires multiple portlets to update the same object

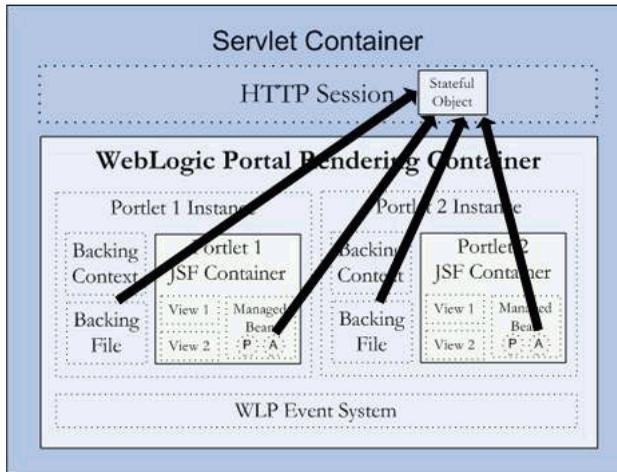
The section contains the following topics:

- [Pattern: Global Session Scoped Data \(uses HttpSession\)](#)
- [Pattern: Portlet Group Session Scoped Data \(uses HttpSession\)](#)
- [Pattern: Portlet Group Session Scoped Data \(uses Events\)](#)

Pattern: Global Session Scoped Data (uses HttpSession)

In this pattern, the state is to be shared with any other component within the web application. All portlets, backing files, JSPs, and other objects need to have access to the same stateful object. In this pattern, the object is set into the `HttpSession` with an attribute name that is not namespaced. The object is truly global in scope.

[Figure 12-5](#) shows an illustration of how Global Session scoped data is accessible to all components.

Figure 12-5 Global Session Scoped Data Being Accessible to all Components

Note: Because it uses the HttpSession, this pattern may be sensitive to the order in which portlets are placed on the page.

[Listing 12-7](#) shows the code that avoids any namespacing of the HttpSession attribute.

Listing 12-7 Code that Avoids any Namespacing of the HttpSession Attribute

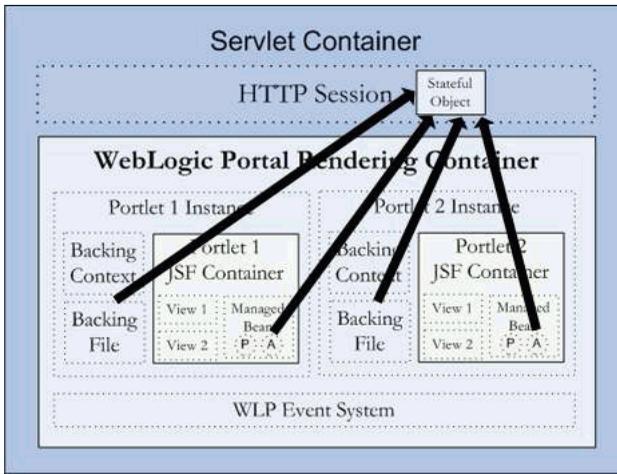
```
public class SearchBean extends SearchBeanBase {
    public String getAttributeNameSpace() {
        return "";
    }
}
```

Pattern: Portlet Group Session Scoped Data (uses HttpSession)

This pattern applies when a group of portlets need to share a stateful object. This pattern employs the HttpSession to enable multiple portlet instances to share a reference to a stateful object. This pattern can also be made to work with a group of portlet definitions – simply replace the instance label pattern with a definition label. With this pattern there are two approaches to generating the namespace for the session attribute: instance label pattern or preference. Both are shown below.

Figure 12-6 shows how multiple portlets can share state via the HttpSession.

Figure 12-6 Multiple Portlets Sharing State via HttpSession



Note: Because it uses the HttpSession, this pattern may be sensitive to the order in which portlets are placed on the page.

Instance Label Pattern

In this pattern, all of the portlets in the group have an instance label that follow a particular pattern. The label pattern contains the attribute namespace in it. For example, the namespace may follow the last underscore character in the instance label. So portlets in the group have labels such as: `master_scope1`, `detail_scope1`, `links_scope1`. Portlets in another group of the same types of portlets would have instance labels like: `master_scope2`, `detail_scope2`, `links_scope2`.

Listing 12-8 shows the code for computing the HttpSession namespace from the portlet's instance label.

Listing 12-8 Computing the HttpSession Namespace from Portlet's Instance Label

```
public class SearchBean extends SearchBeanBase {
    // looks for a namespace in the instance label
    // assumes whatever is found after a trailing _ is the namespace
    public String getAttributeNamespace() {
```

```
        String label = JSFPortletHelper.getPortletInstanceLabel();
        String namespace = JSFPortletHelper.splitNamespaceFromLabel(
            label, "_");
        return namespace;
    }
}
```

Portlet Preference

Another approach is to use a portlet preference to establish the namespace. Every portlet in the group must then use the same value for the preference.

[Listing 12-9](#) shows the code for retrieving a namespace from a portlet preference.

Listing 12-9 Retrieving a Namespace from a Portlet Preference

```
public class SearchBean extends SearchBeanBase {
    // Assumes a portlet preference called "state_namespace"
    // has been defined on each portlet in the group,
    // and contains the namespace key
    public String getAttributeNamespace() {
        PortletPreferences prefs = JSFPortletHelper.getPreferencesObject();
        return JSFPortletHelper.getPreference(prefs, "state_namespace",
            "");
    }
}
```

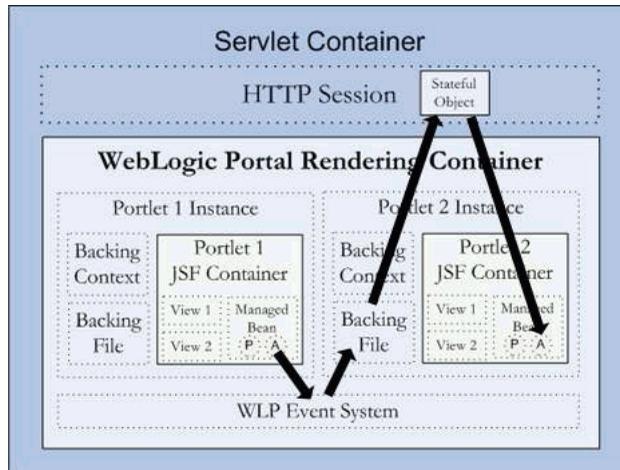
Pattern: Portlet Group Session Scoped Data (uses Events)

WebLogic Portal supports an event mechanism that can carry custom payloads of state from the triggering portlet to any number of listening portlets. This works in a predictable way, and is supported across WSRP. It is a best practice to use this pattern.

There are two sides to this pattern – the trigger and the listeners. The trigger fires a custom event programmatically with the stateful object as the payload. Listener portlets receive the event, and

each stores the state in the `HttpSession` in a portlet instance scoped attribute. A JSF managed bean in that portlet instance can then read the stateful object out of the `HttpSession` (Figure 12-7).

Figure 12-7 Multiple Portlets Sharing State through Events



Trigger Portlet

When a portlet instance needs to write the state object, it must trigger an event with the custom payload. This is easily done in either a portlet backing file or a JSF managed bean. The example below is a managed bean action method.

The code in Listing 12-10 shows how to trigger a WLP event from a JSF managed bean.

Listing 12-10 Triggering a WLP Event from a JSF Managed Bean

```
/**
 * Action method invoked when the user hits the "Search" button
 */
public String doSearch() {
    // searchText is a String that was entered on the search form
    JSFPortletHelper.fireCustomEvent("doSearch", searchText);
    return null;
}
```

```
}
```

Listener Portlets

For any portlet that wishes to listen for the event and receive the stateful object, a portlet backing file must be implemented. It must have an event listener method. The job of the listener method is to set the stateful object into the `HttpSession` scoped to the single portlet pattern shown above. This implies that there is an `HttpSession` entry for every portlet instance listening for the event.

[Listing 12-11](#) shows a backing file method for handling a WLP event and writing the payload into the `HttpSession`.

Listing 12-11 A Backing File Method for Handling a WLP Event and Writing the Payload into the HttpSession

```
public void handleSearchEvent(HttpServletRequest request,
                               HttpServletResponse response, Event event) {
    try {
        String searchText = (String)event.getPayload();
        String namespace =
            JSFPortletHelper.getPortletInstanceLabel(request);
        String attributeName = namespace+"_search_query";
        request.getSession().setAttribute(attributeName,
searchText);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Now, any component of the portlet instance, like a JSF managed bean, can access that state, as shown in [Listing 12-12](#).

Listing 12-12 A JSF Managed Bean Reading the Event Payload out of the HttpSession

```
public String getSearchText() {
```

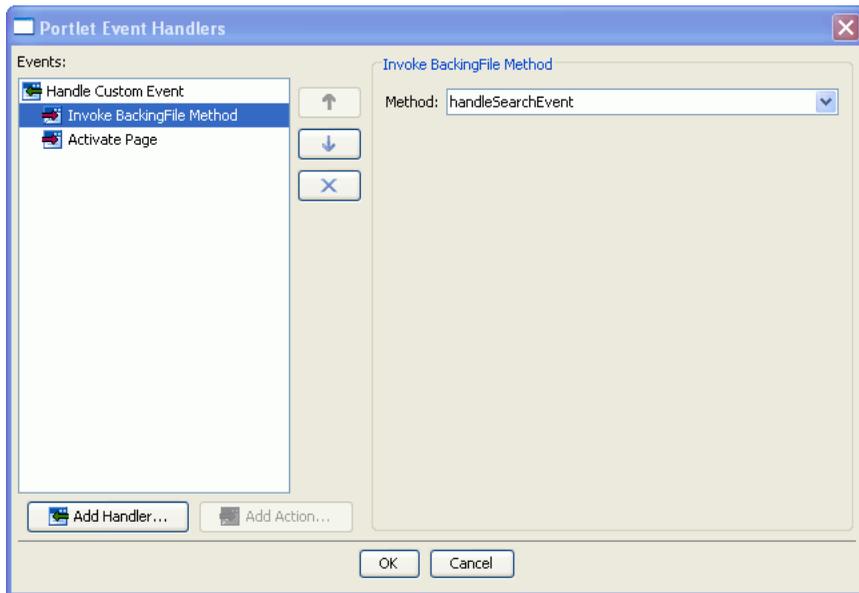
```

String namespace = JSFPortletHelper.getPortletInstanceLabel();
searchText = (String)JSFPortletHelper.getSession().
    getAttribute(namespace+"_search_query");
return searchText;
}

```

To bring it all together, the listening portlet must be configured to listen for the event. This is done in the portlet editor. The portlet must listen for the custom event, and have an "Invoke BackingFile Method" handler defined (Figure 12-8).

Figure 12-8 Configuring a Portlet to Listen for an Event and Invoke a Backing File Method



Using Common WLP Features With JSF Portlets

This section describes how commonly used WebLogic Portal features are used in an environment with JSF portlets.

- [Portlet Container Features](#)
- [Portal Container Features and JSF Portlets](#)

Portlet Container Features

This section discusses these portlet container features:

- [Support for Modes in JSF Portlets](#)
- [Portlet Error Page](#)
- [Portlet Preferences](#)

Support for Modes in JSF Portlets

When creating a new JSF Portlet, only the view mode supports JSF JSPs. The other modes require basic JSP or HTML pages. For more information on portlet modes, see [Portlet Modes](#).

Portlet Error Page

A best practice is to configure an error page for every portlet. You set an error page using the portlet Error Page Path property. For all portlet types, including JSF portlets, this error page must be a standard JSP not a JSF JSP. For details on configuring an error page, see [Portlet Properties](#).

Portlet Preferences

Portlet preferences provide the primary means of associating application data with portlets. Portlet preferences are accessible through the WLP portlet context objects. For detailed information on portlet preferences, see [Portlet Properties](#).

In the context of a JSF portlet, note that after setting preferences values for the portlet, `store()` must be called. This call is best accomplished by setting preferences in JSF managed bean property setters, and then calling `store()` in an action method.

To illustrate this technique, consider a JSF portlet that lets a user get a stock quote. The last quote the user obtains is persisted using portlet preferences. The JSF view is shown in [Listing 12-13](#). The view retrieves portlet preference values from a JSF managed bean. The managed bean, shown in [Listing 12-14](#), sets and gets the preference values from WLP.

Listing 12-13 A JSF View

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
```

```

<%@ taglib uri='http://bea.com/faces/adapter/tags-naming'
prefix='jsf-naming' %>

<f:view>
<jsf-naming:namingContainer id="prefs">
<h:panelGrid columns="4" width="100%">
<h:form id="stockForm">
    <h:panelGroup>
        <h:outputText value="Stock Quote:"/>
    </h:panelGroup>
    <h:panelGroup>
        <h:inputText id="ticker" value="#{WLPPrefsRequestBean.ticker}"
required="true"/>
    </h:panelGroup>
    <h:panelGroup>
        <h:inputText id="shares" value="#{WLPPrefsRequestBean.shares}"
required="true"/>
    </h:panelGroup>
    <h:panelGroup>
        <h:outputText value="#{WLPPrefsRequestBean.currentValue}"/>
    </h:panelGroup>

<h:panelGroup></h:panelGroup>
<h:panelGroup></h:panelGroup>
<h:panelGroup>
    <h:commandButton action="#{WLPPrefsRequestBean.getQuote}"
id="quote" value="Get Quote"/>
</h:panelGroup>
<h:panelGroup>

```

```
<h:commandButton action="#{WLPPrefsRequestBean.resetQuote}"
  id="reset" value="Reset" />
</h:panelGroup>
</h:form>
</h:panelGrid>
</jsf-naming:namingContainer>
</f:view>
```

The managed bean is listed in [Listing 12-14](#).

Note: [Listing 12-14](#) uses a utility class `JSFPortletHelper`. You can find a complete listing and description of this class in the section, [The JSFPortletHelper Class](#).

Listing 12-14 The JSF Managed Bean

```
package oracle.samples.wlp.jsf;

import java.io.Serializable;
import javax.faces.context.FacesContext;
import javax.portlet.PortletPreferences;
import javax.servlet.http.HttpServletRequest;
import
com.bea.netuix.servlets.controls.portlet.PortletPresentationContext;
import
com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext;

/**
 * An example that shows how to use WLP preferences with a JSF managed bean.
 * This example makes the following assumptions for the preference writes to
 * work properly:
```

```
* 1. The bean is request scoped
* 2. The user is authenticated
* 3. The portal is a streaming desktop, not a .portal
*/
public class WLPPrefsRequestBean implements Serializable {
    private static final long serialVersionUID = 1L;
    private static final String TICKER = "ticker";
    private static final String TICKER_DEF = "ORCL";
    private static final String SHARES = "shares";
    private static final String SHARES_DEF = "1000";

    /**
     * The request scoped preferences object.
     */
    private PortletPreferences prefs;

    /**
     * Constructor. Initializes the preference object.
     */
    public WLPPrefsRequestBean () {
        prefs = JSFPortletHelper.getPreferencesObject();
    }

    // ACTION METHODS

    /**
     * Updates the preference values set by the user.
```

```
* @return always null, to retain the same JSF view
*/
public String getQuote() {
    // all the setting work is done in the setters
    // what is left is to store the new prefs into the database
    JSFPortletHelper.storePreferences(prefs);
    return null;
}

/**
 * Resets the preferences back to their defaults.
 * @return always null, to retain the same JSF view
 */
public String resetQuote() {
    JSFPortletHelper.setPreference(prefs, TICKER, TICKER_DEF);
    JSFPortletHelper.setPreference(prefs, SHARES, SHARES_DEF);
    JSFPortletHelper.storePreferences(prefs);
    return null;
}

// GETTERS AND SETTERS

public String getShares() {
    return JSFPortletHelper.getPreference(prefs, SHARES, SHARES_DEF);
}

public void setShares(String shares) {
```

```
        JSFPortletHelper.setPreference(prefs, SHARES, shares);
    }

    public String getTicker() {
        return JSFPortletHelper.getPreference(prefs, TICKER, TICKER_DEF);
    }

    public void setTicker(String ticker) {
        JSFPortletHelper.setPreference(prefs, TICKER, ticker);
    }

    public int getCurrentValue() {
        // convert the String preference into an integer
        String sharesStr = getShares();
        int shares = 0;
        try { shares = Integer.parseInt(sharesStr); }
        catch (Exception e) {}

        // compute some bogus value
        return shares * 52;
    }
}
```

Portal Container Features and JSF Portlets

This section discusses the following portal container features:

- [LocaleProvider](#)

- [Skeleton Files](#)

LocaleProvider

WebLogic Portal by default defers to the `HttpServletRequest` `getLocales()` method to determine the preferred list of locales for the user. JSF implementations do the same.

However, in some cases the portal may have better information about the user's preferred locale. It may use a user profile property for example to store the user's locale preference. In such cases, a developer would implement a WLP `LocaleProvider` to programmatically find the preferred locale in the user profile. In a similar way, with JSF the preferred locale calculation is pluggable. The `ViewHandler` interface has a method `calculateLocale()` that performs the same logic.

When working with JSF portlets, keep these mechanisms in sync. If you implement a custom `LocaleProvider`, use the same code in a custom `ViewHandler` implementation. This ensures that the user sees a portal rendered in a consistent locale.

Skeleton Files

Skeleton files control the rendering of each component of the WLP page. They are implemented as JSPs. The use of JSF components in these JSPs is not supported. The WLP framework renders the skeletons using standard JSP technology. For more information on skeletons, see the [Portal Development Guide](#).

Understanding Navigation

This section discusses navigation within a JSF portlet and within the WLP portal environment when JSF portlets are used.

- [Navigating Within a Portlet with the JSF Controller](#)
- [Support for Redirects](#)

Navigating Within a Portlet with the JSF Controller

This section discusses the use case in which a user interacts with a portlet, triggering an update to that portlet.

It is standard to use Command Buttons and Command Links tied to actions to control navigation through the JSF application (see [Listing 12-15](#) and [Listing 12-16](#)). With the standard JSF navigation controller, there are three approaches to defining the action attribute on a Command Button or Command Link:

- The action attribute is not specified on the component. This indicates that the navigation is a postback to the same view.
- The action attribute is a hard-coded name of a navigation path to follow; this name is mapped in the controller configuration in the `faces-config.xml` file to an actual view (see [Listing 12-15](#)).
- The action attribute contains Expression Language (EL) that invokes a backing bean method that evaluates to a navigation path or null to indicate postback.

Each of these approaches is valid when a JSF application is operating as a JSF portlet. The JSF portlet bridge makes sure the URLs are written properly to maintain correct behavior within a portal. JSF components consult the JSF `ExternalContext` object when constructing action URLs. WLP has implemented an `ExternalContext` that is portal aware, and it properly rewrites the URLs.

Listing 12-15 Navigation Configuration in `faces-config.xml`

```
<navigation-rule>
  <from-view-id>/portlets/sample/page2.jsp</from-view-id>
  <navigation-case>
    <from-outcome>gotoPage1</from-outcome>
    <to-view-id>/portlets/sample/page1.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 12-16 Command Button That Uses the Navigation Rule

```
<h:commandButton action="gotoPage1" id="gotoPage1Button"
  value="Goto Page 1">
</h:commandButton>
```

Support for Redirects

The JSF navigation controller supports HTTP redirect as an annotation on a navigation case. The JSF controller issues an HTTP redirect to the client for the target page of the navigation case if

configured to do so. [Listing 12-17](#) illustrates a redirect in a navigation case. Note the use of the explicit `<redirect/>` element to indicate a redirect.

Listing 12-17 Redirect in a Navigation Case

```
<navigation-rule>
  <from-view-id>/portlets/redirect/page2.jsp</from-view-id>
  <navigation-case>
    <from-outcome>gotoPage1</from-outcome>
    <to-view-id>/portlets/redirect/page1.jsp</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

The navigation case *does not* work properly when the redirect configuration is used within a JSF portlet in WLP. Depending on the context, the navigation case will not be followed or it will send the user to the target page via a redirect taking the user out of the portal context because the redirect URL is not rewritten to remain within the portal environment.

If you must use a redirect and cause the user to remain in the portal, it is possible to achieve using the `sendRedirect()` method on the `PortletBackingContext`.

The last chance to invoke this method is in the `handlePostBackData()` method of a portlet backing file. As discussed in "[Understanding WLP and JSF Rendering Life Cycles](#)", remember that all of the JSF phases for a portlet are processed after `handlePostBackData()`. Therefore, the backing file must decide when to do the redirect before the user interaction is processed by the JSF container. Unfortunately, this means that a redirect cannot be triggered based on the outcome of a JSF form validation.

[Listing 12-18](#) shows a sample code for a backing file that always redirects when the user interacts with the portlet.

Listing 12-18 A Backing File that Redirects when a User Interacts with a Portlet

```
package oracle.samples.wlp.jsf;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```

import
com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking;

import
com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext;

import com.bea.portlet.PageURL;

public class RedirectBackingFile extends AbstractJspBacking {
    private static final long serialVersionUID = 1L;

    @Override
    public boolean handlePostBackData(HttpServletRequest request,
        HttpServletResponse response) {
        // As per the design, this portlet will ALWAYS redirect
        // back to the same portal page
        // if the user interacts with the portlet.
        if (isRequestTargeted(request)) {
            PageURL url = PageURL.createPageURL(request, response);
            // make sure the URL uses the proper ampersands...
            url.setForcedAmpForm(false);
            // ...and is encoded with a session token if necessary
            String redirectUrl =
Response.encodeRedirectURL(url.toString());
            PortletBackingContext pbc =
            PortletBackingContext.getPortletBackingContext(request);
            pbc.sendRedirect(redirectUrl);
        }
        return super.handlePostBackData(request, response);
    }
}

```

Navigation Within a Portal Environment

The previous section discussed navigation techniques within a portlet. This sections covers navigations within the portal environment. It contains the following topics:

- [Programmatically Constructing JSF Portlet URLs](#)
- [Changing the Active Portal Page](#)
- [Using an Output Link](#)
- [Using a Command Link or Button With Events](#)
- [Changing the Active Portal Page Using the Navigation Controller and a Portal Event](#)
- [Changing the Active Portal Page Programmatically](#)

Programmatically Constructing JSF Portlet URLs

For most cases, WLP will automatically handle the construction of URLs that postback to a JSF portlet via the Command Link and Button capability discussed above. But there are use cases in which the portlet developer needs to construct URLs programmatically. The code snippet in [Listing 12-19](#) shows how to do this.

Listing 12-19 Programmatically Constructing a Portlet Postback URL in a JSF Managed Bean

```
public String getJSFPortletPostbackURL() {
    HttpServletRequest request = JSFPortletHelper.getRequest();
    HttpServletResponse response = JSFPortletHelper.getResponse();
    String webapp = request.getContextPath();

    // Create base URL, which includes parameters to target current
portlet
    WindowURL url = WindowURL.createWindowURL(request, response);

    // provide URL to the JSF page. Include webapp context root on the
front
```

```

url.addParameter("_nffvid",webapp+"/portlets/javascript/myJSFPage.faces");
    // make sure the toString() method writes "&" and not "&amp;"
url.setForcedAmpForm(false);
return url.toString();
}

```

Changing the Active Portal Page

The term "page" has many meanings within a portal environment. There is the (X)HTML document that is returned to the browser (the "browser page"), and there is a set of tabs rendered on the browser page, each representing a logical page (a "portal page"). The current visible portal page is called the "active" page.

Within WebLogic Portal, portal pages reside in a container called a book. There is always a main book, which contains the top level portal pages. But note that books can be nested inside of pages, thus creating the ability to create deeply nested user interfaces. This is important to understand in the context of this section because there can be multiple "active" portal pages (in different visible books) rendered in a single browser page.

Using an Output Link

The Output Link component is a perfect for creating link that unconditionally changes the active page. It is simpler to use an Output Link for this task as opposed to the solutions provided with the Command Button and Link. It does require some code to programmatically generate the correct URL.

The first step is to create a managed bean with a property that generates the URL to the appropriate page using a WLP API - the PageURL ([Listing 12-20](#)).

Listing 12-20 A Managed Bean Method for Creating a URL to Navigate to a Portal Page

```

// Creates a property "searchPageURL" that dynamically retrieves
// the right URL to change the page to my search page
public String getSearchPageURL() {

```

```
HttpServletRequest request = JSFPortletHelper.getRequest();
HttpServletResponse response = JSFPortletHelper.getResponse();

// create the URL to the page with definition label "my_search_page"
return PageURL.createPageURL(request, response,
"my_search_page").toString();
}
```

Second, simply add an Output Link component to the JSF portlet. Using JSF EL, bind the managed bean URL to the component ([Listing 12-21](#)).

Listing 12-21 Binding the Dynamic URL into a JSF OutputLink

```
<h:outputLink value="#{Portlet1RequestBean.searchPageURL}">
    <h:outputText value="Change to Search Page"/>
</h:outputLink>
```

Using a Command Link or Button With Events

The Command Link and Command Button components also offer rich capabilities for changing the active page. These features are of particular use when the page change is conditional on server logic.

Changing the Active Portal Page Using the Navigation Controller and a Portal Event

This use case is covered in [Interportlet Communication with JSF Portlets](#). It allows users to implicitly cause a page change based on how they have navigated within a JSF portlet. This produces a loosely coupled approach to the page activation pattern. It is accomplished with the following configuration:

1. The JSF portlet with the Command Button/Link uses standard navigation to change the current JSF view.
2. The JSF portlet is configured to emit an event when that new view is rendered.

3. Locate a listener portlet (it need not be implemented with JSF) that is on the page that is to be activated.
4. That listener portlet is configured to listen for the event triggered by the JSF portlet.
5. An Activate Page action is defined on the handler for the event.

Changing the Active Portal Page Programmatically

Changing the page programmatically with the Command Button and Link is useful when the page change is a conditional decision based on values entered in the accompanying form. There are two approaches.

Solution 1: Fire a Custom Event Programmatically

1. Configure the JSF portlet with the Command Button/Link to use an action listener or method when the button or link is clicked.
2. Implement the method such that it fires a custom event using the PortletBackingContext.
3. Configure a listener portlet for the page to be activated, like in the previous use case.

[Listing 12-22](#) shows how to fire a WLP event from a managed bean that will cause a listener portlet to activate a hidden page.

Listing 12-22 A WLP Event from a Managed Bean that Cause a Listener Portlet to Activate a Hidden Page

```
// Action Listener implemented in a managed bean
public void changeThePage(ActionEvent event)
{
    // fire the event
    // the second parameter can be used to pass arbitrary data
    JSFPortletHelper.fireCustomEvent("changeThePage",
mySerializableObject);
}
```

Solution 2: Invoke a Page Change Programmatically

1. Configure JSF portlet with the Command Button/Link to use an action listener or method when the button or link is clicked.
2. In the method, obtain a reference to the `PortletBackingContext`.
3. Invoke the `setupPageChangeEvent()` method on the `PortletBackingContext`, passing the definition label of the page to be activated.

The code snippet in [Listing 12-23](#) shows how to directly trigger a page change event from a managed bean.

Listing 12-23 Triggering a Page Change Event from a Managed Bean

```
public void changeThePage(ActionEvent event) {
    HttpServletRequest request = JSFPortletHelper.getRequest();
    PortletBackingContext pbc =
        PortletBackingContext.getPortletBackingContext(request);
    // fire the page change event
    // my_search_page is the definition label for the page
    pbc.setupPageChangeEvent("my_search_page");
}
```

Interportlet Communication with JSF Portlets

Most portal implementations employ the pattern known as Interportlet Communication (IPC). IPC refers to the use case in which a portlet needs to notify another portlet, which may or may not be on the visible page, when a user has interacted with it. JSF portlets may participate in both sides of this mechanism, and this section explains how.

WLP provides several facilities for accomplishing IPC – events and notifications, in addition to the basic capability possible with `HttpSession` and `HttpServletRequest` attributes. [Table 12-5](#), available at the end of the section, highlights the differences in capabilities in approaches. Developers are also free to use other approaches beyond what is offered in WLP – JMS for example.

Note: Before proceeding, a clarification must be made. Within WLP, there are two facilities referred to as "events". The first, the one discussed here, operates entirely within the context of the portal rendering framework. It is exposed during portlet configuration and when programmatically interacting with the portal framework context objects. The second, which is not discussed in this chapter, supports the personalization features of WLP, such as Behavior Tracking and Campaigns. That event system is not normally used for IPC.

This section contains the following topics:

- [Using Session and Request Attributes for IPC \(Anti-pattern\)](#)
- [Using the WLP Event Facility for IPC with JSF Portlets](#)
- [Notifications](#)
- [Comparison of the IPC Approaches](#)

Using Session and Request Attributes for IPC (Anti-pattern)

In some basic IPC use cases, it may appear sufficient to use a simple approach of passing state via the `HttpSession` and `HttpServletRequest`. Many developers starting with WLP will use this approach as a first step. However, this technique has many limitations and is not recommended. The drawbacks with this approach are explained in [State Sharing Patterns](#).

Using the WLP Event Facility for IPC with JSF Portlets

This section shows how JSF portlets can participate in the eventing facility provided by the WLP portal framework. Events are extremely well integrated into the WLP portal framework. Therefore, in almost every IPC case, events are the favored approach. More advanced use cases for triggering and receiving events are covered in [State Sharing Patterns](#).

Triggering a Portal Framework Event from a JSF Portlet

This section shows the simplest way in which a JSF portlet can trigger a WLP framework event.

Use case: A search box JSF portlet contains a Command Button that the user presses after entering a search term. The Command Button invokes a navigation rule that changes the view from `searchBox.jsp` to `searchBoxIssued.jsp` being rendered. The Command Button must also raise a WLP event when clicked.

Solution: The following procedure shows how to trigger an event when a user clicks on a JSF Command Link or Button:

1. Create your JSF view for the portlet, which includes an `h:form` (e.g. `/search/searchBox.jsp`).
2. Add a Command Link or Button, with its action defined as a hardcoded logical navigation path ("showResults") or an EL expression that will return one ("`#mybean.mymethod`").
3. Create a navigation rule and case in `faces-config.xml` for that navigation path, directing towards a different JSF view (`searchBoxIssued.jsp`).
4. Using the portlet editor, click on the button in the **Faces Events** property.

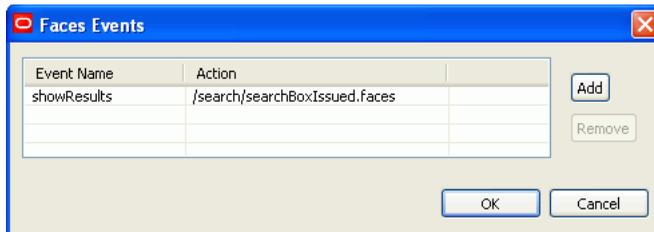
For information about portlet editor, see the “Developing Portlets” chapter available here: http://download.oracle.com/docs/cd/E15919_01/wlp.1032/e14244/configure.htm

5. Declare that the portlet raises the showResults event like the following: (Figure 12-9)

```
Action: /search/searchBoxIssued.faces
eventName: showResults
```

6. Click **OK**.

Figure 12-9 The IDE dialog for Declaring an Event to be Triggered from a JSF Portlet



Listing 12-24 shows the `.portlet` file for a JSF portlet that triggers an event.

Listing 12-24 The `.portlet` File for a JSF Portlet that Triggers an Event

```
<portal:root>
    <netuix:portlet backingFile="wlp.samples.portal.SearchBackingFile"
definitionLabel="searchBox" title="Search Box">
        <netuix:titlebar/>
```

```

        <netuix:content>
            <netuix:facesContent
contentUri="/search/searchBox.faces">
                <netuix:facesEvents>
                    <netuix:facesEvent
action="/search/searchBoxIssued.faces"
                        eventName="showResults"/>
                </netuix:facesEvents>
            </netuix:facesContent>
        </netuix:content>
    </netuix:portlet>
</portal:root>

```

The event will now be triggered in this scenario:

- The user has navigated to a portal page that contains this portlet.
- The portlet's current view contains the Command Button (or Link).
- The user clicks on the button.
- The action method, if specified, returns the correct navigation path to trigger the navigation case to render an alternate view.

Notes:

- If a view has been configured to fire an event when accessed, and it receives a postback, the event is still fired even if form validation fails. This behavior may not be what you want.
- The use of the word “action” as a property key in the event trigger can be confusing. The value of the action property is always the path to a view, and not an action method in a managed bean. The suffix used (.jsp, .faces, etc.) is not significant.
- When defining the "action" property in the event definition, it may not be clear which view is to be specified. It is in fact the destination view that the user will arrive at, not the source view.

Listening for a Portal Framework Event in a JSF Portlet

The previous section detailed how to trigger an event from a JSF portlet. This section explains how to listen for an event.

Use case: In a Master-Detail relationship, the Master is a search box portlet, and the Detail is a JSF search results portlet. When the user enters a query into the search box, the portlet triggers a “showResults” event as shown in the previous section. The search results Detail portlet must be notified such that it can change the view from the default "No Results" page to a view that displays the results.

Solution: A JSF portlet can declaratively listen for portal framework events, which can be used to change the view in the JSF application. This example shows how a portlet can listen for the showResults event from the search box portlet. If the search results portlet receives such an event, it changes the view to a different JSP.

The process of configuring a portlet to listen for an event is as follows:

1. Open the portlet in the Workshop portlet editor.
2. Click on the **Event Handlers** property, and click the button to display the configuration dialog.
3. Click the **Add Handler** button and choose **Custom Event**.
4. Give the name of the event to listen for (e.g. showResults).
5. Click the **Add Action** button, and select **Invoke Faces Action**.
6. Browse to the JSF view that should be displayed upon receiving the event.
7. Click **OK**.

[Listing 12-25](#) shows an example `.portlet` for a JSF search results portlet that listens for an event and changes the JSF view in response.

Listing 12-25 A JSF Search Results Portlet that Listens for an Event and Changes the JSF View in Response

```
<portal:root>

    <netuix:portlet definitionLabel="searchResults" title="Search
Results">

        <netuix:handleFacesEvent
```

```

        eventLabel="doSearchEvent "
        eventName="showResults "
        fromSelfInstanceOnly="false"
        onlyIfDisplayed="false"
        sourceDefinitionLabels="searchBox"
    >

    <netuix:activatePage/>

    <netuix:invokeFacesAction
action="/search/searchResults.faces"/>
    </netuix:handleFacesEvent>
    <netuix:titlebar>
        <netuix:maximize/>
    </netuix:titlebar>
    <netuix:content>
        <netuix:facesContent
contentUri="/search/searchResults_none.faces"/>
    </netuix:content>
    </netuix:portlet>
</portal:root>

```

Notes:

- This example does not show how the search results portlet can get the search text from the search box portlet. Receiving such data in a listening JSF portlet is covered in [State Sharing Patterns](#).
- WLP supports a rich set of actions, beyond Invoke Faces Action, that could also be used in response to an event.
- The listening portlet has additional options for filtering events. See [Chapter 9, “Local Interportlet Communication”](#).

Notifications

WebLogic Portal provides another facility for IPC – notifications. Notifications differ from events in that they are persisted to the database. This allows notifications to persist beyond a user's HttpSession. A notification also differs from an event in that it can be targeted to more than just the user that triggered it. This enables notifications to be used in cases such as sending announcements, advertising a change to a document, broadcasting system-wide messages, and more.

However, notifications do not have the ability to invoke portlet logic (like a backing file method) or activate a portlet's page like an event can. Also, notifications cannot support framework features such as Desktop Async and Portlet Render Cacheable (described below). In cases where those features are important, using a combination of notifications and events can provide the right solution.

Also, the persistence and broadcast features of notifications come at a performance cost. Due to the extra processing required to handle a notification, they are intended to be used sparingly. In short, the event mechanism is normally the correct facility for IPC, but notifications can also be helpful for certain use cases.

Comparison of the IPC Approaches

To make the differences in approaches more clear, [Table 12-5](#) highlights the features supported with each approach:

Table 12-5 A Comparison of Several Implementation Approaches for IPC

FEATURE	SESSION/REQUEST ATTRIBUTES	EVENTS	NOTIFICATION
Recommended?	No	Yes	In moderation
Subscribe or Poll	Poll	Subscribe	Poll
Layout Independent	No	Yes	No
Render Caching	No	Yes	No
Async Desktop	No	Yes	No
WSRP	No	Yes	No

Table 12-5 A Comparison of Several Implementation Approaches for IPC

FEATURE	SESSION/REQUEST ATTRIBUTES	EVENTS	NOTIFICATION
Payload Type	Serializable	Serializable	String
Persistent	No	No	Yes

Legend

This section describes the rows of the table.

- Recommended – Events and notifications are recommended, attributes are not.
- Subscribe or Poll – Events allow a portlet to subscribe to an event, attributes and notifications must poll to detect an event.
- Layout Independent – Depending on certain factors, attributes and notifications may behave differently depending on the order of the master and detail portlets on the page.
- Render Caching – Render caching improves performance by allowing a portlet to cache its rendered view. But the portal framework must invalidate the cached view if a listening portlet has updated itself. The invalidation will only happen if using event based IPC.
- Async Desktop – Enabling the asynchronous desktop feature allows WLP to selectively update only those portlets on a page that need to be updated. This only works for event based IPC.
- WSRP – Portlets can be distributed across different web applications using WSRP. This will not work for attribute or notification based IPC. Events are properly distributed when WSRP is in use.
- Payload – Specifies the Java type that can be used as a payload for the IPC event. Serializable is always recommended, even though Object is technically supported for events and attributes.
- Persistent – Notifications can outlast a session.

Namespacing

This section discusses several namespacing issues that must be addressed when building JSF portlets. It is important to remember in a portal environment, the portlet does not own the entire web page. Therefore, collisions in names can occur if the portlet developer is not careful.

- [Namespacing Managed Bean Names](#)
- [Client ID Namespacing with the View and Subview Components](#)
- [Client ID Namespacing with the WLP NamingContainer](#)

Namespacing Managed Bean Names

With JSF, it is necessary to register each JSF managed bean used in the web application in `facesconfig.xml`. Each bean is given a name, which must be unique to the web application. Since multiple JSF portlets will be brought together into a single web application, it is possible for a naming collision to occur ([Listing 12-26](#)).

Listing 12-26 An Example of a Problematic Managed Bean Name

```
<managed-bean>
    <managed-bean-name>ValidationBean</managed-bean-name>
    <managed-bean-class>samples.ValidationBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

It is a possibility for multiple portlets to define a bean named `ValidationBean`. A better name would be something like `SearchBoxValidationBean`, or some other name very specific to the portlet.

This can especially be an issue when aggregating multiple portlets from disparate development teams. It is therefore a best practice to include the portlet's definition label (or some similar pattern) in the `managed-bean-name` to create an informal namespace for the bean name.

Note that JSF does support dividing `faces-config.xml` into multiple configuration files, but this does not provide any namespacing capability and so the problem remains.

Client ID Namespacing with the View and Subview Components

When a JSF portlet is rendered onto a portal page, it can coexist with other JSF portlets on that same page. In JSF, like other web frameworks, it is critical that the `id` attribute on each element

of the X/HTML browser page be unique. JSF has the concept of a naming container, which provides an id namespace to all components that exist within it. The most common naming container is the View component (the `f:view` tag), but it also provides the Subview component (the `f:subview` tag).

By virtue of the fact that all JSF portlets must contain an `f:view` tag as the root component, the WLP framework for most cases can correctly introduce a namespace for all component ids in each portlet. The framework uses the portlet's instance label as the namespace for the `f:view`. This works automatically, and for most use cases is sufficient.

Client ID Namespacing with the WLP NamingContainer

For the native JSF portlet bridge, WLP provides a `NamingContainer` component to strengthen scoping within the portal. Oracle recommends that you use this component whenever you are using the native JSF portlet bridge.

The purpose of the `NamingContainer` component is to provide a WLP integrated naming container as well as to expose a naming container instance that can be accessed through an EL expression for use in custom id rewriting in backing beans or input to other components.

Add the `NamingContainer` to the JSF view as a direct child of the `f:view` tag. This component takes an `id` attribute for use in EL expressions, which need not be unique across portlets.

[NamingContainer Use Cases](#) illustrates several use cases that demonstrate the `NamingContainer` component.

NamingContainer Use Cases

This section presents several use cases that demonstrate the use of the `NamingContainer` component in JSF views.

Use Case 1: Multiple Portlet Instances on a Page with Desktop Asynchronous Mode Enabled

In general, the `f:view` tag is sufficient for scoping components in a portlet instance. However, if the Desktop Asynchronous Mode feature is enabled on the `.portal` or `desktop`, the `NamingContainer` is required if multiple instances of the same portlet is on the page. Failing to use the `NamingContainer` in this case can cause form submissions to be targeted to the wrong instance. [Listing 12-27](#) illustrates the use of the `NamingContainer` component. For more information on Asynchronous Mode, see "Asynchronous Desktop Rendering" in the [Portal Development Guide](#).

Listing 12-27 Use of the WLP NamingContainer Component

```

<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri='http://bea.com/faces/adapter/tags-naming'
prefix='jsf-naming' %>
<f:view>
<jsf-naming:NamingContainer id="zipcodeScope">
<h:form>
<h:outputLabel value="Enter zip code:" for="zipcode"/>
    <h:inputText id="zipcode" required="false"
        value="#{ZipcodeRequestBean.zipcode}" size="10"/>
    <h:commandButton action="#{ZipcodeRequestBean.submitZipcode}"
        id="submitter" value="Submit"/>
</h:form>
</jsf-naming:NamingContainer>
</f:view>

```

Use Case 2: Using EL to Connect Multiple Components on a Page

A case when the `NamingContainer` `id` attribute is significant is one in which the portlet view has components that must be connected using JSF EL expressions. In this case, the value binding expression needs to have access to the fully scoped name of the other component. The `NamingContainer` `id` attribute is referenced for this use case. The `id` maps to a binding context that is mapped for EL expressions. The binding context provides access to other component `ids` within the `NamingContainer`.

For example, consider a pair of custom components that are configured in master/detail relationship. The detail component needs to have a reference to the master component, but in such a way that will work even if multiple instances of the portlet are on the same page. [Listing 12-28](#) illustrates this use case. Notice the EL expression used by the detail component to refer to the master. The EL expression resolves at runtime to a properly namespaced identifier.

Listing 12-28 JSF EL interacts with the NamingContainer Component

```

<%@ taglib uri=http://bea.com/faces/adapter/tags-naming"
prefix="jsf-portlet" %>
<f:view>
<jsf-portlet:NamingContainer id="masterDetailScope">
    <a:myCustomMasterComponent id="masterComponent"/>
    <a:myCustomDetailComponent id="detailComponent"
        myMasterID="#{masterDetailScope.scopedId['masterComponent']}.string
    }" />
</jsf-portlet:NamingContainer>
</f:view>

```

Use Case 3: Generating DOM Element IDs for Custom JavaScript

The `NamingContainer` is also required in cases where custom JavaScript needs access to the client id of a JSF component to perform dynamic DOM updates. See [Using Custom JavaScript in JSF Portlets](#) for a detailed example.

Best Practice: Always Use the NamingContainer

The specific use cases identified in this section require the `NamingContainer`, but it may not be easy to remember when exactly it is needed. There is no harm in adding the `NamingContainer` component to a portlet. Therefore, it is a best practice to simply add the `NamingContainer` to every portlet. This provides an assurance that users won't experience strange namespacing collisions while navigating the portal, regardless the combination of portlets on the page.

NamingContainer in EL Expressions

When used in an EL expression, the `NamingContainer` is accessed following this pattern:

```
<NamingContainer id>.scopedId[<quoted-string-target-component-id>].string
```

Terminate the JSF value binding expression with `.string` to yield the constructed scoped identifier. For advanced use cases, additional capabilities are available to decorate the generated id. Additional strings may be added to the beginning or end of the generated id:

- `.prefix[<quoted-string>]`
- `.suffix[<quoted-string>]`

For example, this JSF value binding expression

```
myPortletNamespace.scopedId['worldMap'].prefix['mytoken_'].string
```

might produce an identifier like this on the client:

```
mytoken_worldMapPortlet_1:worldMap
```

Using Custom JavaScript in JSF Portlets

This section presents examples and discusses best practices for developing custom JavaScript for JSF portlets.

This section includes these topics:

- [DOM Manipulation within a JSF Portlet](#)
- [Form Validation within a JSF Portlet](#)

Tip: For a summary of general considerations when writing client-side portal code, see "Client-Side Development Best Practices" in the *Client-Side Developer's Guide*. That chapter discusses best practices like using a render dependencies file to specify .js files used by the portlet, using the `wlp_rewrite_` token for namespacing, and other techniques.

DOM Manipulation within a JSF Portlet

This section demonstrates how to rewrite discrete pieces of the HTML page using client-side code.

- [Introduction to the Use Case](#)
- [DOM Manipulation Use Case](#)

Introduction to the Use Case

The need to rewrite pieces of HTML on the client is often called DOM manipulation. It is accomplished by writing custom JavaScript that locates an element on the rendered HTML page (a DOM element), and then programmatically changes that element. The key challenge for this use case for a JSF portlet is determining the target id of the element to be manipulated. [DOM Manipulation Use Case](#) shows how to do this.

Note: It is not generally advisable to manipulate a JSF component's client state via JavaScript, as it fails to properly update the view state stored on the server. But, for custom components or simple components this can be a useful technique.

DOM Manipulation Use Case

This sample shows how to build a simple JSF portlet that contains an Output Link and an Output Text component. The goal is to rewrite the text in the Output Text component when the Output Link is clicked. This happens entirely on the client using JavaScript with no server round trips.

The example starts with the JavaScript. [Listing 12-29](#) shows a sample JavaScript file used by a JSF portlet. It contains very simple logic to find an HTML `<div>` element on the page and update the text. Note the tag `wlp_rewrite`. This tag is rewritten at runtime to the portlet's instance label, which namespaces the JavaScript function. For more information on `wlp_rewrite`, see [Scoping JavaScript Variables and CSS Styles](#).

Listing 12-29 JavaScript File

```
// javascript.js
function wlp_rewrite_updateTarget(divId) {
    var theDiv = document.getElementById(divId);
    theDiv.innerHTML = "Target updated.";
}
```

Next, the JSF portlet's Render Dependencies property is set to a `.dependencies` file. An example `.dependencies` file is shown in [Listing 12-30](#). For more information on render dependencies, see [Portlet Dependencies](#).

Listing 12-30 Sample .dependencies File

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Contents of javascriptPortlet.dependencies file -->
<window xmlns="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">
    <render-dependencies><html>
        <scripts>
```

```
        <script type='text/javascript' content-uri="javascript.js" />
    </scripts>
</html></render-dependencies>
</window>
```

Next, the JSF portlet has a backing bean that provides the portlet instance label, as shown in [Listing 12-31](#). This bean must be registered in `faces-config.xml` (not shown).

Tip: For information about the `JSFPortletHelper` helper class, see [The JSFPortletHelper Class](#).

Listing 12-31 The JSF Managed Bean

```
package oracle.samples.wlp.jsf.javascript;
public class JavaScriptPortletRequestBean {
    private String instanceLabel;

    public String getInstanceLabel() {
        if (instanceLabel == null) {
            instanceLabel = JSFPortletHelper.getInstanceLabel();
        }
        return instanceLabel;
    }
}
```

The JSF JSP file shown in [Listing 12-32](#) completes the example. Notice how it uses the backing bean to retrieve the portlet instance label. It also uses the `NamingContainer` EL expression capability introduced in [Namespacing](#).

The `onclick()` function is the most important component in the code and performs these functions:

- Prepends the JavaScript call with the portlet instance label (to resolve the function namespaced with the `wlp_rewrite` token in the `.js` file)
- Uses the `NamingContainer` EL expression to find the associated `OutputText` component's `` element to update.

Listing 12-32 The JSF JSP That Invokes the JavaScript Function

```

<%@ 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="javascriptPortletNC">
<h:form>
<h:outputLink value="#"
onclick="#{JavaScriptPortletRequestBean.portletInstanceLabel}_updateTarget
('#{javascriptPortletNC.scopedId['outputLinkTarget'].string}'); return
false;">
<h:outputText value="Click to Update OutputLink Target"/>
</h:outputLink>
</h:form>
<h:outputText id="outputLinkTarget" value="OutputLink Target"/>
<%@ 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="javascriptPortletNC">
<h:form>
<h:outputLink value="#"
onclick="#{JavaScriptPortletRequestBean.portletInstanceLabel}_updateTarget
('#{javascriptPortletNC.scopedId['outputLinkTarget'].string}'); return
false;">
<h:outputText value="Click to Update OutputLink Target"/>
</h:outputLink>
</h:form>
<h:outputText id="outputLinkTarget" value="OutputLink Target"/>
</jsf-naming:namingContainer>

```

```
</f:view>
```

Working together, these files create a portlet that works properly by itself, even if multiple instances of the same portlet are on the page.

Form Validation within a JSF Portlet

This use case shows how to accomplish client-side form validation. This case is a common JavaScript use case. This use case follows a similar pattern to the previous use case in [DOM Manipulation within a JSF Portlet](#)

Note: This example is for demonstration purposes only. Oracle recommends that you use the form validation facilities provided by JSF whenever possible.

To start, define the JavaScript function in a `.js` file that performs the validation ([Listing 12-33](#)). Assume the form object is passed in as an argument to the function.

Listing 12-33 The Form Validation JavaScript Function

```
function wlp_rewrite_validateForm(myform) {
    if (myform.elements[0].value.length < 4) {
        alert("The username must be longer than 3 characters.");
        return false;
    }
    return true;
}
```

Using the Render Dependencies technique shown in the previous use case ([DOM Manipulation Use Case](#)):

1. Link the `.js` file to the portlet using a `.dependencies` file.
2. Create the JSF backing bean to accept the form parameters when the form is submitted to the server, as with any standard JSF form. The backing bean must provide the portlet instance label, as shown previously. For brevity, the backing bean is omitted here.
3. Create the JSP JSF page that is the view for the portlet, shown in [Listing 12-34](#). The form element in the JSP triggers client-side form validation. Notice how it uses the same technique used in the previous use case for calling the namespaced JavaScript function.

Listing 12-34 The Form Element in the JSF JSP

```
<h:form id="sampleForm"
    onsubmit="return
#{JavaScriptPortletRequestBean.portletInstanceLabel}_validateForm(this);">
<h:inputText id="username" value="#{JavaScriptPortletRequestBean.name}"
size="40"/>
<h:commandButton id="submitter" value="Submit"/>
</h:form>
```

Ajax Enablement

This section contains the following topics:

- [Ajax in JSF Portlets](#)
- [Partial Page Rendering Pattern](#)
- [Stateless API Request Pattern](#)
- [Portlet Aware API Request Pattern](#)
- [Controlling the WLP Ajax Framework](#)

Ajax in JSF Portlets

Many websites are Ajax enabled to provide a rich user experience. Ajax allows the browser to emulate the interactivity of traditional desktop applications. There are three primary patterns for using Ajax in a Portal. This section describes each, and explains how to implement the pattern with JSF portlets on WebLogic Portal.

Partial Page Rendering Pattern

The Partial Page Rendering (PPR) is the case in which an XMLHttpRequest is issued to the server, and the server responds with visual markup (XHTML or HTML). This markup is written directly into the browser page, updating the content that the user sees. There are several challenges for this capability:

- The Ajax request must enter through a proper Portal entry point, so that state such as portlet preferences and previous view state can be honored.

- It requires the server to be able to return a subset of the markup for the visible page.
- Some portal-like interactions, like interportlet communication, require the server to respond with more markup than the client expected (e.g. markup for multiple portlets).

Because of these complex issues, WLP supports official techniques for implementing PPR with WLP portlets – Asynchronous Desktop and Asynchronous Portlet modes. These facilities are available to all portlet types, and are transparent to the portlet developer. Thus, any JSF portlet can utilize any of these facilities without changes to the portlet implementation. These modes are configured using simple configuration settings.

Asynchronous Desktop Mode

Asynchronous Desktop mode causes all portlets on the portal to become asynchronous. The WLP framework rewrites all URLs on the portal pages such that they invoke Ajax requests (XMLHttpRequests). This facility supports all WLP framework features, including interportlet communication. This setting is either on or off for the entire desktop.

For more information, see the “Asynchronous Desktop Rendering” section in the [Portal Development Guide](#).

Asynchronous Portlet Mode – Ajax or IFrame

Asynchronous Portlet mode is similar to Asynchronous Desktop mode except that it is configured at the portlet level. This allows more fine-grained configuration of what is Ajax-enabled by the framework. This mode can also be configured to use IFrames instead of Ajax. The primary limitation with this mode is that interportlet communication is not supported.

For more information, see the [Asynchronous Portlet Content Rendering](#).

Stateless API Request Pattern

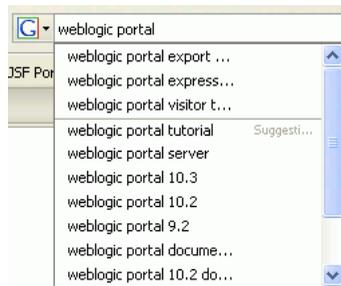
Some use cases work better when the server does not serve up presentation markup, but raw data. In these cases, the client has the necessary code to process that data, and render the presentation markup appropriately.

The first type of this pattern is that of a stateless API. Such an API is not specific to the portal or even portlet; it is a general purpose API. In these cases, the XMLHttpRequest must pass all of the necessary information for the server to process the request. The classic example of this type of API is a search box autocomplete API. In this use case, after a user types a character in a search box, and XMLHttpRequest is sent to the server with the contents of the search box. The server

responds with a list of possible matches, and JavaScript on the client renders them in a drop down list.

Such an API need not be aware of any portal context to fulfill the request. Therefore, any WebLogic Portal portlet (JSF or otherwise) may use such an API without restriction. The Ajax invocation in this case is fulfilled outside the context of the portal environment, and thus there aren't any unique issues related to WLP. For this reason, an example is not provided here. There are many examples available on the web. Searches for "javascript autocomplete example" will yield many results. [Figure 12-10](#) shows an example of autocomplete.

Figure 12-10 An Example of Autocomplete



Portlet Aware API Request Pattern

The third pattern is for Ajax requests that target data service APIs that have access to the portlet context. The service may need access to the portlet instance's preferences, for example. For these cases, WebLogic Portal's framework must be involved in the request processing. In support of this use case, WebLogic Portal must provide the following:

- **Instancing support** – The ability to differentiate which portlet instance on the client issued the request.
- **Context support** – The ability to provide portlet instance context, such as its instance label or preferences.
- **IPC awareness** – If the Ajax invocation changes the state of the portlet, other portlets might also need to be updated.

WebLogic Portal implements a wrapper for the standard XMLHttpRequest to provide these features. [Portlet Aware Data API Example](#) demonstrates its use.

Portlet Aware Data API Example

This example shows how to invoke an API via Ajax that operates within the JSF portlet's context. Meaning, the API implementation has access to portlet instance context such as preferences. The API can return the data in any textual format it chooses, with JSON, XML or simple text being common choices. To keep the example concise, the example as shown will not appear to do anything meaningful. Instead, the example shows how to bring the data back to the browser, and the DOM manipulation is left as an exercise to the reader. See [Using Custom JavaScript in JSF Portlets](#).

[Figure 12-11](#) shows the example. The example consists of a simple portlet with just a link. Clicking on the link on the portlet will cause an Ajax request to target a data API implemented in the JSF portlet. The data can best be seen in a browser debugger like Firebug or a network monitor that shows the response body.

Caution: Using Ajax within a portlet is problematic when that portlet is being offered over WSRP. The 1.0 specification made no allowance for Ajax capabilities, and thus there is no standard approach to solving this problem. Therefore, this example assumes that WSRP is not in use.

Figure 12-11 A View of the Example Portlet



Note: This chapter will not explain render dependencies or the `wlp_rewrite_` token. The example assumes you have already read about JavaScript and render dependencies in section [Portlet Dependencies](#).

Step 1: Create the JavaScript function that issues the Ajax request using the WLP XMLHttpRequest wrapper. Allow the caller to pass in the URL into the function, as generating the proper URL in JavaScript is not easy.

[Listing 12-35](#) shows the JavaScript function that invokes the WLP specific XMLHttpRequest.

Listing 12-35 Invoking the WLP Specific XMLHttpRequest

```
// ajaxPortlet.js
function wlp_rewrite_issueAjax(dataUrl) {
```

```

var xmlhttp = new bea.wlp.disc.io.XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if ((xmlhttp.readyState == 4) && (xmlhttp.status == 200)) {
        var data = xmlhttp.responseText;
        // MODIFY: do something with the data, like DOM
manipulation.

        // This example will appear to do nothing, but the data
is
        // available at this point.
    }
}
xmlhttp.open('GET', dataUrl, true);
xmlhttp.send();
}

```

Step 2: Create the JSF JSP. The URL that is invoked in the Ajax request is retrieved from a JSF backing bean via the EL expression `#{JavaScriptPortletSessionBean.ajaxDataURL}`. The backing bean implementation will be covered next. The portlet instance label is prepended to the function due to the use of the `wlp_rewrite_` token in the JavaScript file.

The code in [Listing 12-36](#) shows the JSF JSP that triggers the Ajax call using an onclick handler.

Listing 12-36 A JSF JSP Triggering the Ajax Call Using an onclick Handler

```

<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri='http://bea.com/faces/adaptor/tags-naming'
prefix='jsf-naming' %>

<f:view>
<jsf-naming:namingContainer id="javascriptPortletNC">

```

```
<h:outputLink value="#"

onclick="#{JavaScriptPortletSessionBean.portletInstanceLabel}_issueAjax('#
{JavaScr

iptPortletSessionBean.ajaxDataURL}'); return false;">

    <h:outputText value="Invoke Portlet Data API via Ajax"/>
</h:outputLink>
</jsf-naming:namingContainer>
</f:view>
```

Step 3: Implement a backing bean that provides the core logic for implementing the Ajax solution. The `getPortletInstanceLabel()` method can remain unchanged, but the `getAjaxDataURL()` and `getJSONData()` methods need to be modified to suit your needs.

[Listing 12-37](#) shows the JSF managed bean that provides the Ajax data API.

Listing 12-37 The JSF Managed Bean that Provides the Ajax Data API

```
package oracle.samples.wlp.jsf;

import javax.faces.context.FacesContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.bea.portlet.WindowURL;
import
com.bea.netuix.servlets.controls.portlet.PortletPresentationContext;
import
com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext;

public class JavaScriptPortletSessionBean {

    // PORTLET INSTANCE LABEL
```

```

private String portletInstanceLabel;

public String getPortletInstanceLabel() {
    return JSFPortletHelper.getInstanceLabel();
}

// AJAX DATA API URL

public String getAjaxDataURL() {
    HttpServletRequest request = JSFPortletHelper.getRequest();
    HttpServletResponse response =
JSFPortletHelper.getResponse();

    String webapp = request.getContextPath();

    // Create the base URL, which includes parameters
    // to target the current portlet
    WindowURL url = WindowURL.createWindowURL(request, response);
    // provide the URL to the data JSF page, include the
    // webapp context root
    url.addParameter("_nffvid",
webapp+"/portlets/javascript/dataJSF.faces");
    // make sure the toString() method writes "&" and not "&amp;"
    url.setForcedAmpForm(false);
    return url.toString();
}

// AJAX DATA API SERVICE IMPLEMENTATION

public String getJSONData() {
    // Could return any text format, like JSON, XML, simple text
    // Data is hardcoded in this example, but could easily be
dynamic.

```

```
        // The service may access the portlet instance context, using
        // a WLP context object like the PortletBackingContext.
        // JSON payload = {"menu": {"id": "file", "value": "File"}}
        return "{\"menu\": {\"id\": \"file\", \"value\": \"File\"}}";
    }
}
```

Step 4: Create the data API, which must be implemented using JSF. The example below uses a single `OutputText` component that obtains the JSON data from the backing bean. Take note that the `contentType` has been set properly because this API returns JSON, which is a good practice.

[Listing 12-38](#) shows the code for the degenerate JSF view that provides the simple data API.

Listing 12-38 The Degenerate JSF View that Provides the Simple Data API

```
<%@ page language="java" contentType="application/json;charset=UTF-8"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<h:outputText value="#{JavaScriptPortletSessionBean.JSONData}"/>
</f:view>
```

Step 5: Create the portlet (`.portlet` file) using the New Portlet wizard. See [JSF Portlets](#).

Step 6: Create the portlet dependencies file (`.dependencies`) to link in the JavaScript file. See [Portlet Dependencies](#).

Step 7: Add the portlet to a portal (`.portal` file) and run. This portal must have DISC enabled. DISC can be enabled in the portal property sheet in the IDE for `.portal` files, or in the Portal Administration Tool for streaming desktops.

Controlling the WLP Ajax Framework

Forcing a Non-Targeted Portlet to Render

This section pertains to cases in which Desktop or Portlet Asynchronous Modes are enabled, or when the WLP specific XMLHttpRequest wrapper is in use. There are cases in which portlets that are not the target of the Ajax request need to be refreshed. If IPC is in use, this will happen automatically. If a user interacts with a portlet and that portlet emits an event that is handled by another portlet, both portlet's will be refreshed as a result of the Ajax interaction. The WLP XMLHttpRequest wrapper ensures this behavior.

However, there are cases where IPC is not being used, yet the same behavior is desired. There is an additional feature of the framework to solve this use case. It requires invoking the `setRenderOnAjaxRequest()` method on the `PortletBackingContext` during the WLP init or `handlePostBack` lifecycles. This can be done either from a portlet backing file or a JSF portlet's managed bean. See the [Understanding WLP and JSF Rendering Life Cycles](#) section for more information on invoking context objects, portlet backing files, and WLP lifecycle methods.

The code snippet in [Listing 12-39](#) shows the backing file method, which would be implemented on the portlet that is NOT the target of the Ajax request.

Listing 12-39 A Backing File that Causes the Portlet to Always be Processed if any Portlet Issues a WLP Ajax Request

```
public void init(HttpServletRequest request, HttpServletResponse response)
{
    PortletBackingContext pbc =
        PortletBackingContext.getPortletBackingContext(request);
    pbc.setRenderOnAjaxRequest(true);
    super.init(request, response);
}
```

Disabling Partial Page Rendering Ajax for a Request

This section applies to cases in which Desktop or Portlet Asynchronous Modes are enabled. While it is useful to allow the framework to automatically rewrite all links and forms to use Ajax,

there are times when a classic page request should be made. For JSF portlets implemented with JSP technology, it is possible to signal to the framework to disable the Ajax rewriting for the components on the portlet view. For more information, see [Asynchronous Content Rendering and IPC](#). The documentation shows how the `<render:context>` tag can be used for this purpose. The way this works within a JSF page is best understood by looking at the following example. The `render:context` tag on the page has disabled the Ajax feature for the JSF form.

The code snippet in [Listing 12-40](#) shows how to disable the Ajax feature for section of a JSF view.

Listing 12-40 Disabling the Ajax Feature for Section of a JSF View

```
<f:view>
<jsf-naming:namingContainer id="ajaxPortletNC">

<render:context asyncContentDisabled="true">
<h:form id="sampleForm">
    <h:inputText id="username" size="40"
        value="#{JavaScriptPortletSessionBean.name}" />
    <h:inputText id="city" size="40"
        value="#{JavaScriptPortletSessionBean.city}" size="40"/>
    <h:commandButton id="submitter" value="Submit"
        action="#{JavaScriptPortletSessionBean.submitForm}"/>
</h:form>
</render:context>

</jsf-naming:namingContainer>
</f:view>
```

Localizing JSF Portlets

This section discusses best practices and techniques for localizing JSF portlets. It contains the following topics:

- [Configuring the JSF Locale](#)
- [Resource Bundles](#)
- [Listing Locales in faces-config.xml](#)
- [Ensuring Parity in Configured WLP and JSF Locales](#)
- [Modularizing Resource Bundles](#)

Configuring the JSF Locale

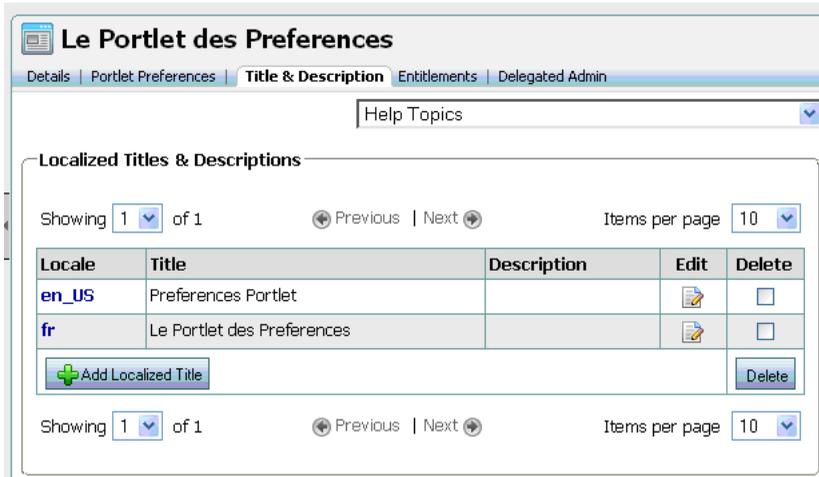
By default, the WLP portal framework and JSF implementation compute the user's locale in the same way. They both call the `HttpServletRequest getLocales()` method to obtain a list of the user's preferred locales, in descending order of desirability. Then, both frameworks find the best match using the localized resource bundles defined for each user interface element.

Resource Bundles

In WebLogic Portal, the resource bundles are called localizations, and are defined using the WLP Administration Console. Localizations cannot be defined when serving a portal from a file (a `.portal` file). The portal must be a streaming desktop that is configured in the database to support localization. For more information on the difference between file based portals and streaming desktops, see "File-Based Portals and Streaming Portals" in the [Portal Development Guide](#).

Each element of the portal framework can have one or more localized titles. [Figure 12-12](#) shows how multiple localized titles for a portlet can be configured in the WLP Administration Console.

Figure 12-12 Defining Multiple Localized Titles for a Portlet



For more information on localizing artifacts in WebLogic Portal, see "Managing Portal Desktops" in the [Portal Development Guide](#).

Listing Locales in faces-config.xml

In JSF, each supported locale must be listed in `faces-config.xml`, as illustrated in [Listing 12-41](#). The views must use an `f:loadBundle` tag to load a resource bundle, as shown in [Listing 12-42](#). That bundle must have a properties file for each supported locale.

Listing 12-41 `faces-config.xml` Showing Multiple Configured Locales

```
<application>
  <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>en</supported-locale>
    <supported-locale>fr</supported-locale>
  </locale-config>
</application>
```

Listing 12-42 Using an f:loadBundle Tag to Load a Resource Bundle

```
<f:loadBundle basename="oracle.samples.wlp.jsf.login_portlet" var="i18n" />
```

Ensuring Parity in Configured WLP and JSF Locales

Because both WLP and JSF compute the desired locale in the same way, the titles written by the portal framework are always in the same locale as the messages written by the JSF implementation. This assumes that WLP localizations and JSF resource bundles support the same locales.

It is a best practice to make sure WLP and JSF are both configured with the same set of locales. This ensures a consistent localized experience for a rendered page.

Modularizing Resource Bundles

JSF portlets support the standard JSF localization facility of resource bundles to provide a localized user experience.

In a portal environment, you need to consider modularizing the resource bundles. In typical JSF applications, one large bundle is created for the entire application. When developing JSF portlets, it is a best practice to create a bundle for each individual portlet definition. Or, if several portlets are always to be used together, a resource bundle can be shared between a group of portlets. This enables portlets to be reused more easily across portal projects.

Preparing JSF Portlets for Production

This section discusses best practices to follow before deploying a JSF portlet into a production environment.

- [Configuration Tasks](#)
- [Performance and Scalability](#)
- [Securing JSF Portlets](#)

Configuration Tasks

This section discusses configuration tasks to perform before deploying a JSF portlet to a production environment.

Configuring URL Templates for Proxy Servers

WLP is responsible for generating URLs that properly stay within the context of the portal. WLP does this for page tabs and also for portlet links. [Understanding Navigation](#) describes how WLP rewrites links for JSF Command Button and Link components automatically.

In production, the URL that a user enters to navigate to a WebLogic Portal instance must not target the machine hosting that instance. Instead, route the user through a proxy server or load balancer. This is a best practice for scalability and for security. WLP provides a configuration facility for configuring URLs properly in such an environment. The JSF link rewriting honors this facility, and so there is nothing JSF-specific about this configuration.

For more information on the WLP URL template facility, see "Working with URLs" in the [Portal Development Guide](#).

The steps for configuring URLs to make use of a proxy server are:

1. Go to the Merged Resources view of your Portal Web Project in Eclipse.
2. Find `WEB-INF/bee-hive-url-template-config.xml`, right click on it, and choose **Copy to Project**.
3. Go to the Project Navigation view and double-click the copied file.
4. Add the URL template entry shown in [Listing 12-43](#), replacing the IP address with the IP address of the proxy server or load balancer.
5. Configure your proxy server to forward to WebLogic Portal.

Listing 12-43 Configure URL Generation That Refers to a Proxy Server

```
<url-template>
  <name>default</name>
  <value>http://192.168.0.5:{url:port}/{url:path}?{url:queryString}{url:currentPage}</value>
</url-template>
```

If using Weblogic Server as your proxy server, see Oracle WebLogic Server proxy plug-in documentation otherwise consult your vendor's proxy server documentation. See the Oracle WebLogic Server proxy plug-in documentation here:

http://download.oracle.com/docs/cd/E12840_01/wls/docs103/plugins/index.html

JSF Portlets with WSRP

When developing portlets with JSF, the WSRP capabilities of WLP work correctly. JSF portlets can be exposed as WSRP portlets, just as with any other portlet type. For detailed information on WSRP portlets, see *Federated Portals Guide*.

Caution: It is not recommended to attempt to implement Ajax features within any portlet, JSF or otherwise, that will be offered over WSRP.

Defining Error Pages

A best practice is to make sure each portlet has a configured error page. This allows the portal framework to provide a user friendly error screen in case of a problem.

A portlet error page is configured as the Error Page Path property in the portlet editor. For more information, see “[Portlet Properties in the Portlet Properties View](#).”

Performance and Scalability

This section discusses best practices for developing JSF portlets that perform well and are scalable.

- [JSF Portlets in a Clustered Environment](#)
- [Portlet Render Caching](#)

JSF Portlets in a Clustered Environment

WebLogic Portal runs on WebLogic Server, which includes industry-leading clustering technology. Clustering provides for both load balancing and failover capabilities. For the most part, WebLogic Server achieves both transparently. You do not need to know about the underlying clustering capabilities.

However, for failover to work properly, WebLogic Server must replicate a user's HttpSession to a secondary node in the cluster. This enables the user to have a seamless experience if one node were to fail. But in order for an HttpSession to be replicated without loss of data, the objects set as attributes within it must be Serializable. For more information, see "Failover and Replication

in a Cluster" in the *Oracle Fusion Middleware Using Clusters for Oracle WebLogic Server* guide available here:

http://download.oracle.com/docs/cd/E12839_01/web.1111/e13709/toc.htm

For JSF portlets to work well in a failover use case, all of the JSF portlet state must be in the HttpSession and be Serializable. As discussed in [Tips for Logging, Iterative Development, and Debugging of JSF Portlets](#), even request-scoped managed beans are written into the HttpSession in a portal environment. Therefore, it is a best practice for all managed beans, even those that are request scoped, to be Serializable.

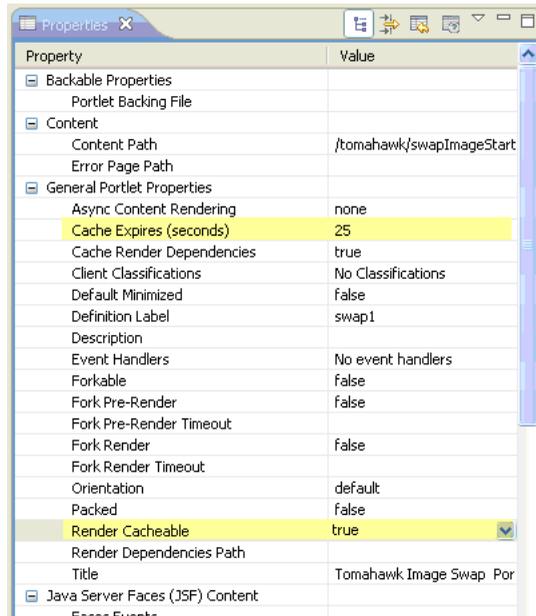
Portlet Render Caching

The WLP portal framework provides an easy solution for improving portlet rendering performance. When a portlet is not the target of a user interaction or does not handle an interportlet communication event during the request, the portlet's markup can be served from cache. By default, a portlet's rendered markup is not cached. To enable portlet caching, set the following values in the Portlet Properties view (see [Figure 12-13](#)).

- Set the Render Cacheable property to `true`.
- Set the Caches Expires property to the number of seconds to cache the portlet markup.

Note: When a JSF portlet's markup is rendered from cache, none of the JSF life cycle methods are invoked during the request.

Figure 12-13 Portlet Cache Configuration



Securing JSF Portlets

This section discusses best practices for securing JSF portlets.

- [Deny Direct Access to the Portlet Views](#)
- [Session Timeouts](#)

Deny Direct Access to the Portlet Views

It is common for developers to develop JSF portlets as standalone JSF applications, in isolation of the portal framework. For this development technique, the Faces servlet must be mapped in `web.xml`, as shown in [Listing 12-44](#).

Listing 12-44 Prefix Servlet Mapping in `web.xml`

```
<servlet-mapping>
  <servlet-name>faces</servlet-name>
```

```
<url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>faces</servlet-name>
  <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

Note that these mappings exist in `web.xml` only to support the direct access use case. Generally, in production it is not recommended to allow users to target the JSF views directly. It is better to force users to navigate to those views through the portal user interface. This can be enforced by removing the prefix and suffix mappings to the Faces servlet in `web.xml`.

Session Timeouts

It is always a good idea to test the behavior of a web application for the request immediately following an `HttpSession` timeout. JSF portlets behave the same as traditional JSF applications – the former state of the portlet is forgotten and the user is taken back to the initial state of the JSF portlet.

Tips for Logging, Iterative Development, and Debugging of JSF Portlets

This section includes these topics:

- [Enabling Logging](#)
- [Using Iterative Development for JSF Portlets](#)
- [Debugging](#)

Enabling Logging

For WebLogic Portal Web Projects the integration of logging is pre-configured. The WebLogic Integrated Commons Logging facet is a required facet for any Portal Web Project. Follow these steps to increase the log sensitivity level to enable Debug messages to be output to the console:

1. Log on to the WebLogic Console, usually at a URL like `http://localhost:7001/console`.
2. Navigate to the **Servers > AdminServer > Logging** tab.

3. Click the **Advanced** link to get the full set of controls.
4. In the Log file section, set the Severity Level to **Debug**.
5. In the Standard out section, set the Severity Level to **Debug**.
6. Click **Save**.
7. On the next request to a portal page that contains JSF portlets, you will see debug output on the server console window.

It may also be beneficial to enable logging for the Sun Reference Implementation of JSF in order to access the JSF debugging messages. The Sun RI of JSF uses Commons Logging for its logging functionality. WebLogic Server has general instructions on integrating Commons Logging into the WebLogic Server. Those instructions can be found at:

http://edocs.beasys.com/wls/docs103/logging/config_logs.html

Using Iterative Development for JSF Portlets

This section discusses techniques for iterative development of JSF portlets.

Testing Outside of the Portlet Container

It is often possible to test a JSF portlet outside of the portlet container. Meaning, it is possible to target the view file directly as with a standard JSF application. In cases where this is possible, this is a good way to develop the portlet. It isolates the JSF container from the portlet container, and so aids in identifying where to look for a solution to a problem. (See also [Securing JSF Portlets](#))

The NamingContainer tag does work correctly outside of the portlet container. But if the portlet relies on features such as backing files or events to work correctly, this approach may not be viable. NamingContainer is discussed in [Namespacing](#).

Using Application Republish

WebLogic Server automatically republishes changes to JSP files; however, changes to source code or deployment descriptors however require an explicit republish in Eclipse (if auto publishing is disabled). This can be done from the Server pane in Eclipse, among other ways. This process is not specific to Portal Web Projects.

HttpSession Caching

JSF caches the user's views in the HttpSession. Therefore, while WebLogic Server detects that a JSP has been updated, any existing session will contain an out of date copy of the view. It is

therefore necessary to begin a new `HttpSession` after modifying a JSF JSP. This issue is not specific to Portal Web Projects.

Handling OutOfMemory Errors

Unfortunately, certain implementations of JSF have memory leaks that occur during a web application redeploy. During long development sessions across many redeployments, the server may fail with an `OutOfMemoryException`. Unfortunately, there is no solution for this issue – a server restart is required. This issue is not specific to Portal Web Projects.

Debugging

This section discusses how to use the Eclipse debugger to troubleshoot JSF portlets. See the Eclipse documentation for information on enabling the Eclipse debugger. This section discusses topics specific to JSF portlets.

Usually the best place to start when debugging a portlet is to set breakpoints in all of the code that you have written. This includes managed bean methods and backing file methods.

But sometimes you need to look into the portal framework. The JSF bridge invokes the JSF implementation, which in turn processes and renders the portlet. For this reason, it's typically best to set break points in the JSF implementation. This is fortunate, as source code is available for the JSF implementations, whereas with the WLP framework source code is not available.

Step 1: Attaching Source

You can investigate resources on the web that explain how to attach source code to the JAR files in your project (for example, you can search for "eclipse attach source"). The main issue for JSF development is locating the proper source files for the JSF implementation used in your web project.

You can download the Sun reference implementation (RI) code from the Mojarra project site at <https://javaserverfaces.dev.java.net>.

Step 2: Suggested JSF Framework Break Points

The following list provides some JSF implementation break points to get started (assumes Sun RI):

- `com.sun.faces.lifecycle.LifecycleImpl.execute()` – The front door to all JSF processing, a good place to start.

- `com.sun.faces.lifecycle.RestoreViewPhase.execute()` – Restores the correct view; useful if the portlet is rendering the wrong view.
- `com.sun.faces.lifecycle.InvokeApplicationPhase.execute()` – Invokes an action; useful when diagnosing issues invoking action methods.

Consolidated List of Best Practices

This section provides a consolidated list of best practices. It contains the following topics:

- [Configuration](#)
- [Namespacing](#)
- [Logging, Iterative Development, Debugging](#)
- [Custom JavaScript](#)
- [Preparing JSF Portlets for Production](#)
- [Interportlet Communication](#)
- [Scopes](#)
- [State Sharing Patterns](#)
- [Rendering Lifecycles](#)
- [Ajax Enablement](#)
- [Login Portlet](#)

Configuration

- Use a Sun RI JSF implementation.
- Always use server `STATE_SAVING_METHOD`.
- Do not use a JSF 1.2 implementation if using the Apache Beehive Page Flow integration.

Namespacing

- Namespace or fully qualify managed bean names in `faces-config.xml`.
- Always add the `NamingContainer` component to a view.

Logging, Iterative Development, Debugging

- Configure WebLogic Server to output JSF debug logging messages during development.
- Test JSF applications standalone before testing as portlets.
- Set breakpoints in the JSF implementation when debugging JSF portlets.

Custom JavaScript

- Make use of the WLP Render Dependencies feature when implementing custom JavaScript.

Preparing JSF Portlets for Production

- Configure the URL templates to properly account for the production network configuration.
- Define an error page for each JSF portlet just in case a problem occurs in production.
- Make all managed beans Serializable to ensure failover works correctly.
- Configure portlet render caching to improve performance.
- Remove mappings to the Faces servlet so that users cannot target a JSF application outside of the portal.
- Test the behavior of the application after the session times out.
- Make sure the supported locales are the same in the JSF and WLP configurations.
- Modularize the localization resource bundles to ensure portlets can be easily reused in other projects.

Interportlet Communication

- Use the WLP event facility to accomplish interportlet communication.
- Use the WLP notification facility if events need to survive across sessions.

Scopes

- The standard JSF scopes are interpreted differently in a portal environment, be sure to understand the differences.

State Sharing Patterns

- Learn the set of patterns for sharing state between a JSF portlet and other components in the portal.

Rendering Lifecycles

- Understand how the JSF and WLP lifecycles work together.
- Be sure to heed the limitation of redirect – they must be triggered in a portlet backing file, not a JSF managed bean.
- Managed beans must be careful accessing the `PortletBackingContext` and `PortletPresentationContext` objects. They fall in and out of scope during the JSF lifecycles.

Ajax Enablement

- Rely on WLP's Asynchronous Desktop feature for partial page rendering use cases. It works automatically.

Login Portlet

- Implementing a login portlet is an expert task. The example provided in [Login Portlet Example](#) is one way in which this could be implemented.

Working With JSF Portlets

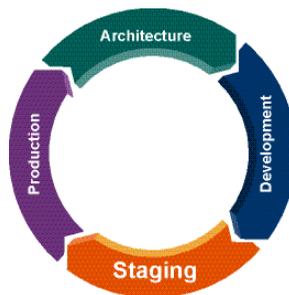
Part III Staging

Part III includes the following chapters:

- [Chapter 13, “Assembling Portlets into Desktops”](#)
- [Chapter 14, “Deploying Portlets”](#)

Oracle 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](#).



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](#)
- [Managing Portlets Using the Administration Console](#)

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 Workshop for WebLogic 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](#)
- [Modifying Library Portlet Properties](#)
- [Modifying Desktop Portlet Properties](#)
- [Deleting a Portlet](#)
- [Managing Portlets on Pages](#)
- [Overview of Portlet Categories](#)
- [Overview of Portlet Preferences](#)
- [Creating a Portlet Preference](#)
- [Editing a Portlet Preference](#)
- [Overview of Delegated Administration](#)
- [Overview of Visitor Entitlements](#)

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 13-1 Modifying Library Portlet Properties

Title and Description	
Change title and description of the portlet in the current locale	<ol style="list-style-type: none"> 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 13-1 Modifying Library Portlet Properties (Continued)

Add a localized title for the portlet	<ol style="list-style-type: none"> 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 13-9 and “Editing a Portlet Preference” on page 13-10 .
Portlet Theme	<ol style="list-style-type: none"> 1. Click Appearance; the Edit Appearance dialog displays. 2. From the drop-down menu, select a Theme. 3. Click Update.
Render caching and timeout	<ol style="list-style-type: none"> 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 13-2 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 13-3 .
Portlet Preferences	Refer to “Creating a Portlet Preference” on page 13-9 and “Editing a Portlet Preference” on page 13-10 .
Portlet Theme	<ol style="list-style-type: none"> 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 13-3 Modifying Portlet Category Properties

Title and Description	
Change title and description of the category in the current locale	<ol style="list-style-type: none"> 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 13-3 Modifying Portlet Category Properties (Continued)

Add a localized title for the category	<ol style="list-style-type: none"> 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 13-8.
Categories in Category	<ol style="list-style-type: none"> 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 13-4 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 13-5 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 13-5 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.

Assembling Portlets into Desktops

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*.

Deploying Portlets

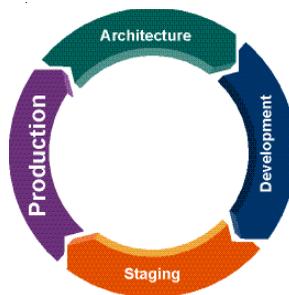
Part IV Production

Part IV includes the following chapter:

- [Chapter 15, “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 [WebLogic Portal Overview](#).



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](#)
- [Transferring Changes from Production Back to Development](#)

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](#).

Part V Appendixes

Part V includes the following appendixes:

- [Appendix A, “Portlet Database Data”](#)
- [Appendix B, “JSF Portlet Development”](#)

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 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.

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_PORTLET_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 Console are directly reflected in the database.

This section contains the following sections:

- [Types of Database Tables](#)
- [Management of Portlet Data](#)

- [How the Database Shows Removed Portlets](#)

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 Console, but you must use the Administration Console 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 Console.
- **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_PORTLET_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 Console, 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*](#).

Portlet Database Data

JSF Portlet Development

This appendix provides information on specific use cases, development tips, and code examples. It contains the following sections:

- [Code Examples](#)
- [Using Facelets](#)
- [Using Tomahawk](#)
- [Integrating Apache Beehive Pageflow Controller](#)
- [Building Unsupported JSF Implementations](#)

Note: For information about procedures and best practices for developing and configuring JSF portlets, see [Chapter 12, “Working With JSF Portlets.”](#)

Code Examples

This section includes the following topics:

- [The JSFPortletHelper Class](#)
- [Login Portlet Example](#)

The JSFPortletHelper Class

This section provides sample code for a helper class that could be used in JSF portlets. It provides helpful methods for developing managed beans that need to have access to WLP context.

Listing B-1 A Comprehensive Helper Class for JSF Portlets

```
package oracle.samples.wlp.jsf;

import java.io.Serializable;
import java.security.Principal;
import java.util.ResourceBundle;

import javax.faces.context.FacesContext;
import javax.portlet.PortletPreferences;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import weblogic.servlet.security.ServletAuthentication;

import
com.bea.netuix.servlets.controls.portlet.PortletPresentationContext;
import
com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext;
import com.bea.netuix.servlets.manager.AppContext;

/**
 * A helper class with many useful methods for JSF portlets. These
 * methods are expected to be called from JSF managed beans primarily,
 * but some may also be useful in the portlet backing files of JSF portlets.
 */
public class JSFPortletHelper {
```

```
// STANDARD CONTEXT

/**
 * Gets the HttpServletRequest from the FacesContext.
 * Must only be called from a JSF managed bean.
 *
 * @return a HttpServletRequest implementation, which is actually a
FacesRequest object
 */
static public HttpServletRequest getRequest() {
    FacesContext fc = FacesContext.getCurrentInstance();
    return
(HttpServletRequest)fc.getExternalContext().getRequest();
}

/**
 * Gets the HttpSession from the FacesContext.
 * Must only be called from a JSF managed bean.
 *
 * @return a HttpSession implementation
 */
static public HttpSession getSession() {
    HttpServletRequest request = getRequest();
    return request.getSession();
}

/**
 * Gets the HttpServletResponse from the FacesContext.
 * Must only be called from a JSF managed bean.
```

JSF Portlet Development

```
*  
* @return a HttpServletResponse implementation, which is actually a  
FacesResponse object  
*/  
static public HttpServletResponse getResponse() {  
    FacesContext fc = FacesContext.getCurrentInstance();  
    return  
(HttpServletResponse)fc.getExternalContext().getResponse();  
}  
/**  
* Gets the localized resource bundle using the passed bundle name  
* Must only be called from a JSF managed bean.  
*  
* @param bundleName the String name of the bundle  
* @return the ResourceBundle containing the localized messages for  
the view  
*/  
static public ResourceBundle getBundle(String bundleName) {  
    FacesContext context = FacesContext.getCurrentInstance();  
    ResourceBundle bundle = ResourceBundle.getBundle(bundleName,  
        context.getViewRoot().getLocale());  
    return bundle;  
}  
/**  
* Gets the localized message using the passed bundle name and message  
key.  
* Must only be called from a JSF managed bean.  
*  
*
```

```

    * @param bundleName the String name of the bundle
    * @param messageKey the String key to be found in the bundle
properties file
    * @return the String containing the localized message
    */
    static public String getBundleMessage(String bundleName, String
messageKey) {
        String message = "";
        ResourceBundle bundle = getBundle(bundleName);
        if (bundle != null) {
            message = bundle.getString(messageKey);
        }
        return message;
    }
// PORTAL ENVIRONMENT
/**
 * Gets the PortletBackingContext object. This method will return null
 * if called during the RENDER_RESPONSE JSF lifecycle.
 * Must only be called from a JSF managed bean.
 *
 * @return the active PortletBackingContext, or null
 */
    static public PortletBackingContext getPortletBackingContext() {
        FacesContext fc = FacesContext.getCurrentInstance();
        HttpServletRequest request =
(HttpServletRequest)fc.getExternalContext().getRequest();
        return
PortletBackingContext.getPortletBackingContext(request);
    }

```

```
    }  
    /**  
    * Gets the PortletPresentationContext object. This method will return  
null  
    * if NOT called during the RENDER_RESPONSE JSF lifecycle.  
    * Must only be called from a JSF managed bean.  
    *  
    * @return the active PortletPresentationContext, or null  
    */  
    static public PortletPresentationContext  
getPortletPresentationContext() {  
        FacesContext fc = FacesContext.getCurrentInstance();  
        HttpServletRequest request =  
(HttpServletRequest)fc.getExternalContext().getRequest();  
        return  
PortletPresentationContext.getPortletPresentationContext(request);  
    }  
    /**  
    * Returns true if the user can make customizations  
    * (preferences, add/move/remove portlets, add pages) to the portal.  
    * This is based on factors such as: is the user authenticated,  
    * is it a streaming portal (not a .portal file),  
    * and customization is enabled in netuix-config.xml.  
    * Can be called from any web application class.  
    *  
    * @return a boolean, true if it is possible for the user to make  
customizations  
    */  
    static public boolean isCustomizable() {
```

```

        return ApplicationContext.isCustomizationAllowed(getRequest());
    }

    // AUTHENTICATION
    /**
     * Is the current user authenticated?
     * Must only be called from a JSF managed bean.
     *
     * @return true if the user is authenticated, false if not
     */
    static public boolean isAuthenticated() {
        Principal principal =
FacesContext.getCurrentInstance().getExternalContext().getUserPrincipal();
        return principal != null;
    }

    /**
     * Get the current user's username from the container.
     * Must only be called from a JSF managed bean.
     *
     * @return the user name, null if the user is not authenticated
     */
    static public String getUsername() {
        String username = null;

        Principal principal =
FacesContext.getCurrentInstance().getExternalContext().getUserPrincipal();
        if (principal != null) {
            username = principal.getName();
        }

        return username;
    }

```

JSF Portlet Development

```
    }  
    /**  
     * Get the current user's username for display. If the user is not  
    authenticated, it  
     * will return the name passed as the anonymousUsername parameter. DO  
    NOT use this method  
     * for anything other than display (e.g. access control, auditing,  
    business logic),  
     * as the passed anonymous name may conflict with an actual username  
    in the system.  
     * Must only be called from a JSF managed bean.  
     *  
     * @param anonymousUsername a String localized name to use for an  
    anonymous user, like "Guest"  
     * @return the user name  
     */  
    static public String getUsernameForDisplay(String anonymousUsername)  
    {  
        String username = anonymousUsername;  
        Principal principal =  
    FacesContext.getCurrentInstance().getExternalContext().getUserPrincipal();  
        if (principal != null) {  
            username = principal.getName();  
        }  
        return username;  
    }  
    // USER AUTHENTICATION ROUTINES  
    /**  
     * Authenticate the user with WebLogic Server
```

```

* Can be called from any web application class.
*
* @param username the String username
* @param password the String password as provided by the user
* @return true if the login was successful, false if not
*/
static public boolean authenticate(String username, String password)
{
    HttpServletRequest request = getRequest();
    HttpServletResponse response = getResponse();
    int result = ServletAuthentication.weak(username, password,
request, response);
    return result !=
ServletAuthentication.FAILED_AUTHENTICATION;
}
// NAMESPACES AND LABELS
/**
* Gets the current portlet's instance label.
* Must only be called from a JSF managed bean.
*
* @return the String instance label
*/
static public String getInstanceLabel() {
    return getInstanceLabel(getRequest());
}
/**
* Gets the current portlet's instance label.
* Can be called from any web application class.

```

```
*
* @param the HttpServletRequest object
* @return the String instance label
*/
static public String getInstanceLabel(HttpServletRequest request) {
    String label = "_global";
    PortletBackingContext pbc =
PortletBackingContext.getPortletBackingContext(request);
    if (pbc != null) {
        label = pbc.getInstanceLabel();
    }
    else {
        PortletPresentationContext ppc =
PortletPresentationContext.getPortletPresentationContext(request);
        if (ppc != null) {
            label = ppc.getInstanceLabel();
        }
    }
    return label;
}
/**
* Gets the current portlet's definition label.
* Must only be called from a JSF managed bean.
*
* @return the String definition label
*/
static public String getDefinitionLabel() {
    return getDefinitionLabel(getRequest());
}
```

```

    }
    /**
     * Gets the current portlet's definition label.
     * Can be called from any web application class.
     *
     * @param the HttpServletRequest object
     * @return the String definition label
     */
    static public String getDefinitionLabel(HttpServletRequest request)
{
    String label = "_global";
    PortletBackingContext pbc =
PortletBackingContext.getPortletBackingContext(request);
    if (pbc != null) {
        label = pbc.getDefinitionLabel();
    }
    else {
        PortletPresentationContext ppc =
PortletPresentationContext.getPortletPresentationContext(request);
        if (ppc != null) {
            label = ppc.getDefinitionLabel();
        }
    }
    return label;
}
    /**
     * Finds a namespace embedded in a portlet instance
     * or definition label. This namespace is intended to

```

```
* be used as a prefix for attributes set into the
* HttpSession as a way to share state between portlet
* instances. See the State Sharing Patterns section.
* <p>
* This method expects to find the namespace at the end
* of the label, following the passed delimiter.
* <p>
* For example: <ul>
* <li>label = "myportlet_group1"
* <li>delimiter = "_"
* <li>return = "group1"
* </ul>
* @param label the String instance or definition label
* @param delimiter the
* @return the String namespace
*/
static public String splitNamespaceFromLabel(String label,
                                             String delimiter) {
    String namespace = label;
    int lastIndex = label.lastIndexOf(delimiter);
    if (lastIndex > -1) {
        // namespaced
        namespace = label.substring(lastIndex);
    }
    else {
        // not namespaced, noop
    }
}
```

```

        return namespace;
    }

    // PORTAL EVENTS
    /**
     * Fires a custom portal event with a payload.
     * Must only be called from a JSF managed bean.
     *
     * @param eventName the String name of the event
     * @param payload the Serializable payload
     * @return true if the event was fired, false if it could not be
     */
    static public boolean fireCustomEvent(String eventName, Serializable
payload) {
        boolean fired = false;
        PortletBackingContext pbc = getPortletBackingContext();
        if (pbc != null) {
            pbc.fireCustomEvent(eventName, payload);
            fired = true;
        }
        return fired;
    }

    // WLP PORTLET PREFERENCE OPERATIONS
    /**
     * Gets an instantiated preferences object for the portlet;
     * it must be obtained once per request.
     * Must only be called from a JSF managed bean.
     *
     * @return the PortletPreferences object for the request

```

```
*/
static public PortletPreferences getPreferencesObject() {
    PortletPreferences prefs = null;
    PortletBackingContext pbc = getPortletBackingContext();
    HttpServletRequest request = getRequest();
    if (pbc != null) {
        prefs = pbc.getPortletPreferences(request);
    } else {
        PortletPresentationContext ppc =
PortletPresentationContext.getPortletPresentationContext(request);
        if (ppc != null) {
            prefs = ppc.getPortletPreferences(request);
        }
    }
    return prefs;
}
/**
 * Gets the single value preference.
 *
 * @param name the String name of the preference
 * @param value the String default value to use if the preference isn't
set
 * @return the String value
 */
static public String getPreference(PortletPreferences prefs, String
name, String value) {
    if (prefs != null) {
        value = prefs.getValue(name, value);
    }
}
```

```

        }
        return value;
    }
    /**
     * Sets a single value preference into the preferences object.
     * storePreferences() must be called subsequently to persist the
change.
     *
     * @param name the String name of the preference
     * @param value the String value of the preference
     */
    static public void setPreference(PortletPreferences prefs, String
name, String value) {
        if (prefs != null) {
            try {
                prefs.setValue(name, value);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    /**
     * After setting updated values into the preferences object, call this
atomic
     * method so they can be stored in the persistent store in a single
     * operation.
     *
     * @param prefs the PortletPreferences to be persisted

```

```

    * @return a boolean, true if the store succeeded
    */
    static public boolean storePreferences(PortletPreferences prefs) {
        if (!isCustomizable()) {
            return false;
        }
        try {
            prefs.store();
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
        return true;
    }
}

```

Login Portlet Example

This section provides one example of how a Login portlet may be implemented with JSF in order to demonstrate a few key concepts already discussed. Each situation may be different, but this example provides an illustration to help explain details of what can be a complex topic.

This section includes the following topics:

- [Login Portlet Motivation](#)
- [Login Portlet Design](#)
- [Login Portlet Implementation](#)

Login Portlet Motivation

WebLogic Portal does not provide a Login portlet out of the box. It has assumed that each developer has custom login requirements, and therefore needs to implement a custom login portlet. While this is true, often the first portlet that a developer attempts to build is a Login

portlet. Unfortunately, a Login portlet is not the easiest task to start from, and many improperly designed Login portlets have been created for WLP.

This section, in recognition of this history, provides a working Login portlet implemented with JSF that is both a useful resource to deploy, but also demonstrates a few key concepts already discussed. It also presents solutions to several unique challenges that are encountered when implementing a login/logout portlet.

Login Portlet Design

The login portlet is implemented in JSF. It uses a single view that toggles the visibility of the login/logout controls based on the authenticated state of the user. In summary, the login portlet offers the following features:

- Implemented as a JSF portlet.
- A single view with two forms – one for login and one for logout.
- Uses a managed bean to perform the authentication logic.
- Employs the JSF localization facility.

Figure B-1 The Login Portlet When the User is Not Authenticated



Figure B-2 The Login Portlet When the User is Authenticated



It appears to be a straightforward form driven JSF portlet. However, as stated in the introduction, a login portlet has more complexities than a typical portlet. Upon logging into or out of a portal, the portal framework must recompute the view that the user has of the portal to take into account that user's authorization and customizations. When login is implemented as a standalone page

within a WLP web application, this is not an issue. But when the login/logout facility is itself a portlet, there are several problems.

This section includes the following topics:

- [Redirects](#)
- [Invalidating the Session](#)

Redirects

By the time a user becomes authenticated, the portal framework is already in the middle of processing the rendered page. It is too late for the framework to recompute the page. Therefore, the solution is to force a redirect after the user is authenticated or logged out with the login portlet. As covered in [Native Bridge Architecture](#), there are some constraints that must be observed to accomplish a redirect:

- Processing of the JSF login form with username and password is done in the JSF Invoke Application lifecycle, naturally.
- The last chance for a portlet to redirect is in a WLP portlet backing file's `handlePostBackData()` method.
- All of the JSF lifecycles, including Invoke Application, occur after the backing file's `handlePostBackData()` method.

This leads to the design requirement that the portlet has both a JSF managed backing bean and a portlet backing file. The backing file is responsible for the redirect, and the managed bean handles the form processing and authentication. However, these aren't executed in the right order. Ideally, the redirect would only occur on a successful authentication or log out. The solution offered in this login portlet is to always redirect whenever a user interacts with the portlet. This causes the browser to redirect even for failed login attempts, but that is a minor concession.

Invalidating the Session

A second issue must also be addressed. The WLP JSF portlet native bridge has a limitation in that a user's `HttpSession` cannot be invalidated in the middle of processing the JSF lifecycles. This best location for the logout logic is in the backing file.

Login Portlet Implementation

This section includes the following topics:

- [JSF Login View](#)

- [JSF Managed Backing Bean](#)
- [faces-config.xml](#)
- [Backing File](#)
- [Resource Bundle](#)
- [Portlet Definition File](#)

JSF Login View

The `login.jsp` page contains two forms – one for login and one with a button to logout. Only one form is visible at a time, and the visibility is controlled by a flag that indicates whether the user is authenticated. Otherwise, this JSF view is straightforward.

Listing B-2 The JSF JSP for the Login Portlet

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@ taglib uri='http://bea.com/faces/adapter/tags-naming'
    prefix='jsf-naming' %>

<f:loadBundle
    basename="oracle.samples.wlp.jsf.portlets.login.loginportlet"
    var="i18n" />
<f:view>
<jsf-naming:namingContainer id="login_portlet">

<h:form id="loginBeanForm"
    rendered="#{!JSFLoginPortletRequestBean.authenticated}">
<h:panelGrid id="outerLayout" columns="1" width="100%"
```

```
                style="background-color: azure; ">
<h:panelGroup id="titleLine">
    <h:outputText value="#{i18n.login_intro}:"
                style="color: cornflowerblue; font-size: medium"/>
</h:panelGroup>

<h:panelGroup id="errorMessage">
    <h:messages layout="table" style="color: red; font-weight: bold"/>
</h:panelGroup>

<h:panelGroup id="formFields">
    <h:panelGrid columns="2" width="60%"
                style="background-color: azure">
        <h:panelGroup style="text-align: right">
            <h:outputText value="#{i18n.login_username}:" />
        </h:panelGroup>
        <h:panelGroup style="text-align: left">
            <h:inputText id="username" required="true"
                value="#{JSFLoginPortletRequestBean.username}" />
        </h:panelGroup>

        <h:panelGroup style="text-align: right">
            <h:outputText value="#{i18n.login_password}:" />
        </h:panelGroup>
        <h:panelGroup style="text-align: left">
            <h:inputSecret id="password" required="true"
                value="#{JSFLoginPortletRequestBean.password}" />
        </h:panelGroup>
    </h:panelGrid>
</h:panelGroup>
```

```

    </h:panelGroup>

    <h:panelGroup/>
    <h:panelGroup style="text-align: left">
        <h:commandButton id="loginButton" immediate="false"
            action="#{JSFLoginPortletRequestBean.authenticate}"
            value="#{i18n.login_button}"/>
    </h:panelGroup>
    </h:panelGrid>
</h:panelGroup>
</h:panelGrid>
</h:form>

<h:form id="logoutForm"
    rendered="#{JSFLoginPortletRequestBean.authenticated}">
    <h:commandButton action="#{JSFLoginPortletRequestBean.userLogout}"
        id="logoutButton" value="#{i18n.logout_button}"/>
</h:form>
</jsf-naming:namingContainer>
</f:view>

```

JSF Managed Backing Bean

The JSF managed backing bean contains the core logic for logging the user in. There is nothing tricky about this code. It does contain the WebLogic specific code for authenticating a user, but is otherwise standard code. For information about the referenced JSFPortletHelper class, see [The JSFPortletHelper Class](#).

Listing B-3 The JSF Managed Bean for the Login Portlet

```
package oracle.samples.wlp.jsf.portlets.login;
```

JSF Portlet Development

```
import java.io.Serializable;

import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;

import oracle.samples.wlp.jsf.JSFPortletHelper;

public class JSFLoginPortletRequestBean implements Serializable {
    private static final long serialVersionUID = 1L;
    private static final String BUNDLE_NAME =
"oracle.samples.wlp.jsf.portlets.login.loginportlet";
    private String username;
    private String password;
    // ACTION METHODS
    /**
     * Action method called during logout
     */
    public String userLogout() {
        // Due to a limitation in the bridge, logout must NOT
        // be done in the middle of the JSF lifecycles.
        // Therefore logout is actually done prior to the JSF
        // lifecycles in the request, in the handlePostbackData()
        // method of the JSFLoginPortletBacking backing file.
        return null;
    }
    /**
     * Action method for the login CommandButton
     */
}
```

```

* @return
*/
public String authenticate() {
    // perform the actual authentication call
    boolean success = JSFPortletHelper.authenticate(
        username, password);
    // Handle the result
    if (!success) {
        // Login failed
        // Add an error message to indicate login failure
        String errorText = JSFPortletHelper.getBundleMessage(
            BUNDLE_NAME, "login_error");
        // Add the message to the context
        FacesContext fc = FacesContext.getCurrentInstance();
        FacesMessage msg = new FacesMessage(
            FacesMessage.SEVERITY_ERROR, errorText,
errorText);
        fc.addMessage(null, msg);
    }
    else {
        // Login succeeded

        // Wipe out the password, just in case someone
        // is tempted to make this bean Session scoped (should
be
        // Request) By keeping this password around in the
bean,
        // it is open to temptation for abuse

```

```
        password = "invalidated";
    }
    return null;
}

// GETTERS AND SETTERS
/**
 * @return true if the user is authenticated
 */
public boolean isAuthenticated() {
    return JSFPortletHelper.isAuthenticated();
}

/**
 * @return the user name
 */
public String getUsername() {
    return JSFPortletHelper.getUsernameForDisplay("");
}

/**
 * @param username the user name to be authenticated
 */
public void setUsername(String username) {
    this.username = username;
}

/**
 * Retrieves a placeholder for the password. We never
 * want to display back the actual password to the user,
```

```

    * so just return an empty string
    * @return an empty String
    */
    public String getPassword() {
        return "";
    }

    /**
     * @param password the password to be used for authentication
     */
    public void setPassword(String password) {
        this.password = password;
    }
}

```

faces-config.xml

The managed bean needs to be wired into the application. The following XML element must be added to `faces-config.xml`.

Listing B-4 Registering the Login Managed Bean in faces-config.xml

```

<managed-bean>
    <description>Handles authentication for the Login
portlet.</description>
    <managed-bean-name>JSFLoginRequestBean</managed-bean-name>
    <managed-bean-class>
        oracle.samples.wlp.jsf.JSFLoginPortletRequestBean
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>

```

```
</managed-bean>
```

Backing File

The backing file is where the redirect happens. In this example, the redirect is directed at the page on which the login portlet resides. This is important because the login credentials passed by the user may be incorrect, so redirecting to the same page allows the user to see if the login failed.

In addition, the WLP JSF portlet native bridge does not work properly if the user's HttpSession is invalidated during the middle of the JSF lifecycles. Therefore, logout cannot happen in the JSF managed bean. It must happen before the JSF lifecycles are invoked, which should be done in a backing file.

Listing B-5 Implementation of the Backing File that Performs the Redirect for the Login Portlet

```
package oracle.samples.wlp.jsf.portlets.login;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import oracle.samples.wlp.jsf.JSFPortletHelper;
import
com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking;
import
com.bea.netuix.servlets.controls.portlet.backing.PortletBackingContext;
import com.bea.portlet.PageURL;

public class JSFLoginPortletBacking extends AbstractJspBacking {
    private static final long serialVersionUID = 1L;
    @Override
    public boolean handlePostBackData(HttpServletRequest request,
        HttpServletResponse response) {
```

```

// As per the design, the login portlet will ALWAYS redirect
if the

// user interacts with the portlet.
if (isRequestTargeted(request)) {
    // If the user is authenticated, a form POST
    // to this portlet signals a logout.
    // Logout must be done before the JSF lifecycles start
    // (bridge limitation).
    if (("POST".equals(request.getMethod())) &&
        isAuthenticated(request)) {
        HttpSession session =
request.getSession(false);
        session.invalidate();
    }
    // redirect back to the same portal page
    PageURL url = PageURL.createPageURL(request,
response);

    // make sure the URL uses the proper ampersands...
url.setForcedAmpForm(false);

    // ...and is encoded with a session token if necessary
    String redirectUrl =
response.encodeRedirectURL(url.toString());
    PortletBackingContext pbc =
PortletBackingContext.getPortletBackingContext(request);
    pbc.sendRedirect(redirectUrl);
}
return super.handlePostBackData(request, response);
}

```

```
/**
 * Is the current user authenticated?
 *
 * @return true if the user is authenticated, false if not
 */
protected boolean isAuthenticated(HttpServletRequest request) {
    return JSFPortletHelper.isAuthenticated(request);
}
}
```

Resource Bundle

As per best practices, create a resource bundle for the login portlet. This one should be located in a file `oracle/samples/wlp/jsf/portlets/login/loginportlet.properties` in the Java Resources/src folder.

Listing B-6 The Localizable Resource Bundle for the Login Portlet

```
# login.jsp
login_title=Login Page
login_imageAlt=Login
login_intro=Please enter your username and password
login_username=Username
login_password=Password
login_button=Login
logout_button=Logout
login_error=The username or password are invalid.
```

Portlet Definition File

Finally, the `.portlet` file ties everything together. Most important is the backing file reference. You will not normally create this file as XML – use the portlet editor.

Listing B-7 The Portlet Definition File for the Login Portlet

```

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

<portal:root
xmlns:netuix="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0"

xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/portal/support/1
.0.0 portal-support-1_0_0.xsd">

    <netuix:portlet
backingFile="oracle.samples.wlp.jsf.portlets.login.JSFLoginPortletBacking"
    definitionLabel="login" title="Login">

        <netuix:titlebar>

            <netuix:minimize/>

        </netuix:titlebar>

        <netuix:content>

            <netuix:facesContent
contentUri="/portlets/login/login.faces"/>

        </netuix:content>

    </netuix:portlet>

</portal:root>

```

Using Facelets

This section contains the following topics:

- [Introduction to Facelets](#)
- [Configuring Facelets Support](#)

Introduction to Facelets

Intermixing JSF components with JSP tags can cause problems. While JSP is the default technology in which to implement JSF views, there are issues in how the two technologies work together. There are many references to these problems available in articles and blog posts on the internet.

An alternate view technology called Facelets has been created by the JSF community which avoids the problems seen with JSPs. It offers an XML grammar for declaratively wiring up a JSF view. It more cleanly separates the view definition from the programming logic. As an assertion of the strength of this approach, JSF 2.0 has adopted Facelets as the preferred view technology. This section explains how to configure a Portal Web Project with Facelets.

Note: Before committing to Facelets, be sure to understand the IDE impact. The Workshop IDE only supports Facelets configuration with JSF 1.1. For more details, see [Chapter 12, “Working With JSF Portlets.”](#)

Configuring Facelets Support

Follow these steps to configure Facelets within a Portal Web Project:

1. Start with a Web Project that is properly configured to support JSF JSPs, including a prefix or suffix mapping if you wish to support access to the views outside of portlets.
2. Download a Facelet implementation library (see <https://facelets.dev.java.net/>).
3. Copy the library into the Web Project's `WEB-INF/lib` directory.
4. Add the Facelets view handler to `faces-config.xml`. (see [Listing B-8](#)).
5. Change the default suffix for Faces to be `.xhtml`. (see [Listing B-9](#)).
6. Create a new Facelet file with suffix `.xhtml`. (Source code is provided in [Listing B-10](#)).
7. Test the configuration by targeting the Facelet directly. The URL will depend on how you have configured prefix or suffix mapping for Faces. `http://localhost/jsfweb/mytest.jsf`
8. Create a new portlet, selecting JSF as the type, and the `.xhtml` file as the content. **Note:** You will have to enter the path by hand as the wizard does not allow you to select an `.xhtml` file.
9. Add the portlet to a portal, and republish the web application.

Note: The current JSF portlet support (native bridge) cannot support both JSP and Facelets JSF portlets in the same web application. You must choose one or the other. The standard workaround for combining both JSPs and Facelets in the same web application, the

VIEW_MAPPINGS technique, is not possible with the WLP native bridge. WSRP could be used as a workaround for this issue.

Listing B-8 The view-handler Element in faces-config.xml

```
<application>
    <!-- tell JSF to use Facelets ->
    <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
</application>
```

Listing B-9 Change the Default Suffix for Faces Requests in web.xml

```
<context-param>
    <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value>.xhtml</param-value>
</context-param>
```

Listing B-10 Create a Basic facelet .xhtml File in the WebContent Folder

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type"
    content="text/html; charset=iso-8859-1" />
<title>My First Facelet</title>
</head>
<body>
    <!-- body content goes here -->
    Hello #{param.name}!
```

```
</body>
```

```
</html>
```

Using Tomahawk

This section includes the following topics:

- [What is Apache MyFaces Tomahawk?](#)
- [Support for Tomahawk in WLP](#)
- [Installing and Configuring Tomahawk](#)
- [Installing and Configuring Tomahawk](#)
- [Resolving the Duplicate ID Issue](#)
- [Referring to Resources](#)
- [forceId Attribute](#)
- [File Upload](#)

What is Apache MyFaces Tomahawk?

Apache MyFaces Tomahawk is an open source component library that provides an enhanced set of JSF components that go beyond the basic set provided by the core JSF specification. Tomahawk enhances the standard components, and provides more than 40 additional components not included in the standard set. This section covers how to use Tomahawk components within WLP JSF portlets.

Official documentation and project information can be found at these locations:

- Tomahawk Official Site –

<http://myfaces.apache.org/tomahawk/index.html>

- Tomahawk Official Wiki –

<http://wiki.apache.org/myfaces/Tomahawk>

The enhancements to the standard set include:

- User-role Awareness – renders the component visible and/or enabled based on the user-roles of the current user.
- DisplayValueOnly – toggles between output/input mode.
- forceId – do not let JSF generate an assembled id of the ids of the component and its parents, instead use the provided id.
- ExtendedMessagesSupport – automatically replaces the id of the message with the corresponding label or column header.

The additional components offered include a calendar, a data grid, a tree, a tabbed pane, and many more.

Support for Tomahawk in WLP

The JSR-286 and JSR-329 standards bodies are ensuring that JSF applications work properly in a portal environment. In addition, the component libraries such as Tomahawk continue to change to support the new portal requirements. In general, the Tomahawk components work properly. But WLP does not certify Tomahawk or officially support it, so issues may be found in certain use cases.

Use Tomahawk 1.1.7 and Later

Note that some of the work that the Tomahawk team implemented for JSR-301 and JSR-329 will help even when using WebLogic Portal versions that do not support JSR-329. For example, Tomahawk version 1.1.7 obviated the need for the MyFaces Extensions Filter, which is problematic in a portal environment, to help support JSR-329. This change fortunately also benefits WLP 10.3.

Tomahawk 1.1.8 was the version used in these examples. Because of ongoing work to support JSR-329, it is beneficial to look at the latest available version of Tomahawk. It is not recommended to adopt a version prior to 1.1.7.

Portlet Scoping

The [Namespacing](#) section covered the WLP specific NamingContainer component. This aids in preventing conflicts when the same portlet is placed on a portal page multiple times. One case in which this is necessary is when a component uses JavaScript. Since Tomahawk components are typically JavaScript-enabled, remember to add a NamingContainer to any portlet that uses a Tomahawk component.

However, not all Tomahawk components work properly when they appear in multiple portlets on the same page, even when the NamingContainer is used. Oracle recommends explicitly testing all Tomahawk components to determine if they suffer from this issue. Check with the Tomahawk documentation for more information on this topic. Later versions may be less problematic.

Ajax Enablement

Tomahawk components are not Ajax enabled. However, the WLP asynchronous desktop and portlet features described in the [Ajax Enablement](#) section do work properly with the Tomahawk components. This provides the user with the Ajax responsiveness usually only found in Ajax enabled component libraries such as Apache MyFaces Trinidad.

Installing and Configuring Tomahawk

Tomahawk is not included in a WebLogic Portal installation. It is necessary to download Tomahawk and install and configure it into the WLP web application.

Follow these steps:

1. Download the Tomahawk JAR file, like `tomahawk-1.1.8.jar` or later.
2. Copy the JAR file into `WEB-INF/lib`.
3. Add the following entries in `web.xml` in the appropriate places.

Listing B-11 `web.xml` Settings for Tomahawk

```
<!-- Make two entries that explicitly enable the portal-friendly changes
introduced with v1.1.7 -->
    <context-param>
        <param-name>org.apache.myfaces.CHECK_EXTENSIONS_FILTER</param-name>
        <param-value>>false</param-value>
    </context-param>
    <context-param>
        <param-name>
            org.apache.myfaces.DISABLE_TOMAHAWK_FACES_CONTEXT_WRAPPER
        </param-name>
```

```

    <param-value>>false</param-value>
  </context-param>

  <!-- Configure the mechanism for bringing in resources (.js, .css).
->

  <context-param>
    <param-name>org.apache.myfaces.ADD_RESOURCE_CLASS</param-name>
    <param-value>

org.apache.myfaces.renderkit.html.util.NonBufferingAddResource

    </param-value>
  </context-param>

  <!-- Map the resource loading capability of Tomahawk ->
  <servlet-mapping>
    <servlet-name>faces</servlet-name>
    <url-pattern>/faces/myFacesExtensionResource/*</url-pattern>
  </servlet-mapping>

```

Resolving the Duplicate ID Issue

Some Tomahawk components, when used in a portlet environment will emit a Duplicate Id Exception. To handle this case, the CleanupPhaseListener Class is a recommended workaround.

CleanupPhaseListener Class

This class is a solution to a Duplicate ID issue found with the use of some Tomahawk components. The JSF specification allows custom components to be marked transient, which means they must be discarded at the end of the HttpRequest. Some component libraries, such as Tomahawk, rely on this behavior.

This PhaseListener is a solution to the problem. [Listing B-12](#) shows the code for the PhaseListener.

Listing B-12 The CleanupPhaseListener Source Code

```
package oracle.samples.wlp.jsf;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import javax.faces.component.UIComponent;
import javax.faces.component.UIViewRoot;
import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

/**
 * This PhaseListener must be employed when using the WLP JSF portlet native
 * bridge with certain component libraries like Tomahawk. The native bridge
 * by default does not correctly handle UIComponent's that are marked as
 * transient via the UIComponent interface.
 * <p>
 * This PhaseListener implementation corrects that by clearing out all
 * transient UIComponents at the end the request lifecycle.
 * <p>
 * @version Affects all WLP releases 10.x (10.0, 10.2, 10gR3)
 */
public class CleanupPhaseListener implements PhaseListener {
```

```

/**
 * This phases listener is only executed after all JSF phases
 * are run. Therefore this method returns PhaseId.RENDER_RESPONSE
 */
@Override
public PhaseId getPhaseId() {
    return PhaseId.RENDER_RESPONSE;
}
/**
 * After RENDER_RESPONSE, purges any transient UIComponent from the
view.
 */
@Override
public void afterPhase(PhaseEvent event) {
    FacesContext context = event.getFacesContext();
    UIViewRoot root = context.getViewRoot();
    purgeTransientFromTree(context, root, "");
}
/**
 * Walks the component tree, clearing out any transient component.
 */
protected void purgeTransientFromTree(FacesContext context,
UIComponent component, String parentIndent) {
    UIComponent child;
    List<UIComponent> children = component.getChildren();
    Iterator<UIComponent> childrenIterator =
children.iterator();
    List<UIComponent> transients = new ArrayList<UIComponent>();

```

```

        while (childrenIterator.hasNext()) {
            child = childrenIterator.next();
            if (child.isTransient()) {
                transients.add(child);
            } else {
                purgeTransientFromTree(context, child,
                    parentIndent+" ");
            }
        }
        // now remove the children we found to be transient
        childrenIterator = transients.iterator();
        while (childrenIterator.hasNext()) {
            child = (UIComponent) childrenIterator.next();
            children.remove(child);
        }
    }
    @Override
    public void beforePhase(PhaseEvent event) {}
    private static final long serialVersionUID = 1L;
}

```

It must be registered in `faces-config.xml`, as shown in [Listing B-13](#).

Listing B-13 Registering the CleanupPhaseListener in faces-config.xml

```

<lifecycle>
    <!-- The CleanupPhaseListener corrects an issue in the native bridge
         implementation only seen with custom components. This phase

```

```

        listener can always be used, but is needed specifically when
        Tomahawk components are used.

->
<phase-listener>
    oracle.samples.wlp.jsf.CleanupPhaseListener
</phase-listener>
</lifecycle>

```

Referring to Resources

Most web applications require JavaScript and CSS files. JSF portlets will often have such a requirement, and the [Using Custom JavaScript in JSF Portlets](#) section explained how to accomplish this with WLP for the general JSF case. Most Tomahawk components require resources to be loaded to function properly. This section explains the options when Tomahawk components are present.

Tomahawk has a pluggable mechanism for inserting references to JavaScript and CSS files into a page containing Tomahawk components. There are three implementations that are included in the library, each with benefits and drawbacks. WLP offers an additional option, Render Dependencies, that should also be considered. This section will explain several options.

The pluggable resource handling interface within Tomahawk is `AddResource`, and the three implementations are:

- `DefaultAddResource`
- `NonBufferingAddResource`
- `StreamingAddResource`

The chosen implementation is configured in `web.xml`, shown in .

This section includes the following topics:

- [Using `DefaultAddResource` \(Not recommended\)](#)
- [Using `NonBufferingAddResource` \(Simplest\)](#)
- [Using a Static WLP Render Dependencies File \(Most correct, but tedious\)](#)

- [Using Dynamic WLP Render Dependencies \(Not possible, for reference only\)](#)

Using DefaultAddResource (Not recommended)

The `DefaultAddResource` buffers the entire response, and inserts any needed resources into `<HEAD>` before returning the response to the client. The primary benefit is that the resources are included in the proper location in the HTML document (`HEAD`). The downside is that the entire response must be buffered, which can consume significant server memory.

When using this option with JSF portlets within WLP, there is an additional drawback. The `DefaultAddResource` implementation works by locating the `HEAD` element in the markup rendered by the JSF view. In a portal, each JSF portlet contains a JSF view and thus its own `DefaultAddResource` instance. Because of this fact, each portlet must contain a `HEAD` element into which its `DefaultAddResource` instance will write the resource references. However, it is a best practice to not render HTML document elements (`HTML`, `HEAD`, `BODY`) in portlets as it creates invalid HTML.

Another drawback in a portal environment is that this approach cannot eliminate duplicate resource references. This is due to each portlet having its own `DefaultAddResource` instance. Each portlet will have references to the resources it needs, even if another portlet has already included it.

When the `DefaultAddResource` is operating outside of a portal environment, its chief advantage is that it renders valid HTML. Within a portal, it does not have this advantage. There are several additional drawbacks with this approach:

- Each portlet's response is buffered, which will consume server memory.
- Resources may be referenced multiple times in the same page.
- It requires a `HEAD` element per portlet within the `BODY` element of the HTML document. This is invalid HTML.
- The portlet developer must remember to include a `HEAD` element in each JSF view. Omitting it will cause failures.

[Listing B-14](#) shows an example portlet JSP that will work properly with `DefaultAddResource`. Notice the required `HEAD` element, which will be populated at runtime with the necessary resource references.

Listing B-14 A JSP Using Tomahawk with DefaultAddResource

```

<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t" %>
<%@ taglib uri='http://bea.com/faces/adapter/tags-naming'
prefix='jsf-naming' %>
<head><!-- required for Tomahawk, do not remove --></head>
<f:view>
<jsf-naming:namingContainer id="swapImage">
    <t:panelGroup id="swapImagePanel">
        <t:swapImage id="image" value="/tomahawk/images/MyFaces_logo.jpg"
            swapImageUrl="/tomahawk/images/MyFaces_logo_inverse2.jpg"/>
    </t:panelGroup>
</jsf-naming:namingContainer>
</f:view>

```

Using NonBufferingAddResource (Simplest)

The `NonBufferingAddResource` implementation does not buffer the response. Instead, it writes resource references directly into the Tomahawk component's markup within the `BODY` element of the HTML document. This is technically invalid HTML, as the resource references should be in the `HEAD` element. However modern browsers can handle this markup properly.

The advantage of this approach is it is simpler and does not buffer the response. Also, the portlet developer does not need to remember to add an empty `HEAD` element. There are several drawbacks with this approach:

- Resources may be referenced multiple times in the same page, as with the `DefaultAddResource`.
- It writes resource references into the `BODY` element of the HTML document. This is invalid HTML.

Listing B-15 A JSP of a Portlet that Works Properly with NonBufferingAddResource

```

<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t" %>
<%@ taglib uri='http://bea.com/faces/adapter/tags-naming'
prefix='jsf-naming' %>

<f:view>
<jsf-naming:namingContainer id="swapImage">
    <t:panelGroup id="swapImagePanel">
        <t:swapImage id="image" value="/tomahawk/images/MyFaces_logo.jpg"
            swapImageUrl="/tomahawk/images/MyFaces_logo_inverse2.jpg"/>
    </t:panelGroup>
</jsf-naming:namingContainer>
</f:view>

```

Using a Static WLP Render Dependencies File (Most correct, but tedious)

WLP provides a general facility for solving the resource inclusion problem that works across all portlet types and the various WLP asynchronous modes. It generates valid HTML, and eliminates duplicate resource references. It is a best practice to use this facility. The Render Dependencies facility is covered in [Using Custom JavaScript in JSF Portlets](#).

As an example, this is a `.dependencies` file for a JSF portlet that contains a Tomahawk Tree2 component. Notice how it explicitly references the Tomahawk scripts that it needs.

Listing B-16 Using the WLP Render Dependencies Mechanism to Include Tomahawk Resources

```

<?xml version="1.0" encoding="UTF-8"?>
<window
    xmlns="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
lafwindow-1_0_0.xsd">
    <render-dependencies>
        <html>
            <scripts>
                <script type="text/javascript"
src="/wlpJSF/faces/myFacesExtensionResource/org.apache.myfaces.renderkit.h
tml.util.MyFacesResourceLoader/12306184/tree2.HtmlTreeRenderer/javascript/
tree.js" />
                <script type="text/javascript"
src="/wlpJSF/faces/myFacesExtensionResource/org.apache.myfaces.renderkit.h
tml.util.MyFacesResourceLoader/12306184/tree2.HtmlTreeRenderer/javascript/
cookielib.js" />
            </scripts>
        </html>
    </render-dependencies>
</window>

```

However, there are several issues with this approach.

First, the developer must assemble the list of scripts that are needed by all of the Tomahawk components within all views of the portlet. This is a manual process, and must be kept up to date as the portlet implementation changes. One way to assemble the list is to use one of the other approaches covered above during development, and look into the HTML document to determine what scripts were referenced.

Second, when using WLP Render Dependencies mechanism to write the resource references, it is important to make sure that a Tomahawk AddResource implementation doesn't also write resource references into the response. But Tomahawk does not provide an AddResource implementation that never writes resources to the response. One way to achieve this with the standard Tomahawk distribution is to use the DefaultAddResource implementation and ensure

that all portlets do not include a `HEAD` element. Another option would be to implement a custom `AddResource` implementation that does not write resource references.

Using Dynamic WLP Render Dependencies (Not possible, for reference only)

This section documents an approach that will not work. This is provided to illuminate boundaries as to what is possible.

The previous section described how a static `Render Dependencies` file could be used to include the resource references. There are drawbacks with that approach in that the list of references must be known at development time, and must be maintained as the portlet implementation changes. Another approach would be to attempt to write a new `AddResource` implementation that would invoke the `WLP Render Dependency` mechanism to dynamically add the references. This would be ideal, as it would eliminate the drawbacks discussed with the previous approach.

The documentation for the dynamic render dependencies facility is available here:

- [DynamicHtmlRenderDependencies JavaDoc](#)

Unfortunately, the Tomahawk `AddResource` mechanism operates during the JSF `Render` lifecycle. As seen in the [Understanding WLP and JSF Rendering Life Cycles](#) section, the JSF `Render` lifecycle is invoked within the `WLP Render` lifecycle. This is a problem because the dynamic `Render Dependencies` must be established before the `WLP Render` lifecycle. Therefore, this approach will not work.

Here is an example of how an `AddResource` method might be implemented, but unfortunately does not work because of the timing of the lifecycles.

[Listing B-17](#) shows a non-working example of how `WLP` can be dynamically notified of render dependencies using the Tomahawk `AddResource` mechanism.

Listing B-17 Using the Tomahawk `AddResource` Mechanism to Dynamically Notify `WLP` of Render Dependencies

```
@Override
public void addJavaScriptHerePlain(FacesContext context, String uri)
throws IOException {
    HttpServletRequest request =
        (HttpServletRequest)context.getExternalContext().getRequest();
```

```

PortalLookAndFeel laf = PortalLookAndFeel.getInstance(request);
DynamicHtmlRenderDependencies injector =
laf.getDynamicHtmlRenderDependencies();
injector.addScript("text/javascript", null, null, uri, null);
}

```

forceId Attribute

Tomahawk provides an attribute named `forceId` for many Tomahawk components. This attribute indicates that the component's client id should be exactly what is specified in the component's `id` attribute. This feature short circuits any namespacing provided by the enclosing JSF naming containers (`f:view`, `f:subview`, `WLP NamingContainer`). This feature helps in cases in which JavaScript cannot cope with the scoped client ids.

However, as noted in the [Namespacing](#) section and in JSF texts, namespacing client ids is sometimes critical. In the case of WebLogic Portal, when the same portlet is placed on a portal page multiple times, the client id scoping is needed to prevent certain collisions. For better or for worse, the Tomahawk `forceId` attribute does exactly what it promises, even in a WLP environment. It eliminates any scoping added for components in portlet instances. Be careful using this attribute, especially when the same portlet may be added to a page more than once.

[Listing B-18](#) shows the use of `forceId`. When this JSP is used as a portlet, the HTML id for the `<div>` that is written for the `panelGroup` component is exactly "swapImagePanel". It is not namespaced with the portlet instance label.

Listing B-18 An Example Showing the Use of the Tomahawk `forceId` Attribute

```

<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t" %>
<%@ taglib uri='http://bea.com/faces/adapters/tags-naming'
prefix='jsf-naming' %>

<f:view>
    <jsf-naming:namingContainer id="swapImage">

```

```
<t:panelGroup id="swapImagePanel" forceId="true">
    <t:swapImage id="image"
value="/tomahawk/images/MyFaces_logo.jpg"
swapImageUrl="/tomahawk/images/MyFaces_logo_inverse.jpg"/>
</t:panelGroup>
/jsf-naming:namingContainer>
</f:view>
```

File Upload

Tomahawk offers a File Upload component that enables a developer to easily create a web page that allows a user to upload a document. Unfortunately, this component does not function properly within a WLP portal. The only workaround currently is to implement File Upload in a non-JSF portlet.

Integrating Apache Beehive Pageflow Controller

This section includes the following topics:

- [Apache Beehive Page Flow](#)
- [JSF and Page Flows](#)
- [Configuring the JSF Integration with Page Flows](#)

Apache Beehive Page Flow

The Apache Beehive project supports a full featured controller technology called Page Flows. Page Flows are annotated Java class files that provide navigation control, state management, form validation and more. Page Flows are built on top of the Struts controller.

The use of the Page Flow controller is only recommended for customers that have existing assets written in Apache Beehive.

JSF and Page Flows

Beehive Page Flows use regular JSPs as the view technology. However, the Beehive project built an integration with JSF such that JSF JSPs can be used. This integration combines the full JSF

rendering framework (with lifecycles and managed beans) with the power of the Beehive Page Flow controller. This integration was built independently of WebLogic Portal and it works equally well both inside and outside of WLP JSF portlets.

The integration provides the following features:

Actions

- Allows a JSF action to be wired to a Beehive Page Flow action.
- Ability to map a Page Flow action result to `Jpf.NavigateTo.previousPage` and `Jpf.NavigateTo.currentPage`, and the correct JSF view will be restored.

Backing Beans

- Defines a base class for Page Flow aware JSF backing beans – `FacesBackingBean`.
- Manages the lifecycle of the JSF backing bean for a page. It provides what the JSF 2.0 specification calls "View" scope.
- Does not require an entry for the backing bean in `faces-config.xml`; it is wired using an annotation.

JSF Expression Language Binding Contexts

- Maps the view scoped backing bean into the 'backing' context.
- Maps the Netui `pageInput` into the 'pageInput' context.
- Maps the current Page Flow into the 'pageFlow' context.
- Maps shared Page Flows into the 'sharedFlow' context.

The integration works by plugging a custom `ViewHandler` and `NavigationHandler` into the JSF implementation. There are several places to go for more information about the JSF-Beehive integration:

- Integrating JavaServer Faces with Beehive Page Flow Article –
<http://www.oracle.com/technology/pub/articles/dev2arch/2005/12/integrating-jsfbeehive.html>
- Official Beehive Documentation –
http://beehive.apache.org/docs/1.0/netui/sample_jpf_jsf_integration.html
- Tutorial: JavaServer Faces / NetUI Integration –

<http://edocs.bea.com/wlw/docs103/guide/webapplications/jsf/jsf-integrationtutorial/tutJSFIntro.html>

Configuring the JSF Integration with Page Flows

The JSF integration is built directly into the Apache Beehive library. Since all Portal Web Projects currently require the Apache Beehive facet, this integration is provisioned with every WLP web application. There aren't any configuration changes necessary for the integration to work within a WLP portlet.

PageFlowApplicationFactory

After creating a Portal Web Project with the JSF facet enabled, the default `faces-config.xml` will contain this configuration artifact:

Listing B-19 The Beehive PageFlowApplicationFactory Configuration in faces-config.xml

```
<application-factory>
    org.apache.beehive.netui.pageflow.faces.PageFlowApplicationFactory
</application-factory>
```

This configuration is not coming from a WebLogic Portal facet – it is added by the Apache Beehive Netui facet. It enables the Apache Beehive Page Flow integration with JSF. If the integration is not needed for the web project, it can be removed.

Note that the Page Flow integration is currently only certified with JSF 1.1.

Building Unsupported JSF Implementations

It is not officially supported by Oracle to use a JSF implementation outside of the set provided in the box. However, there are times when it is helpful to experiment with different versions of an implementation when troubleshooting an issue. The exact steps may vary depending on what implementation and version is in use.

1. Extract the closest existing JSF library module from `BEA_HOME/WL_HOME/common/deployable-libraries` into an empty folder.
2. Update `META-INF/manifest.mf` to the new implementation version.

3. Copy in the new implementation JARs into the `lib` folder.
4. Create a new WAR file with the updated contents, updating the filename with the new version number.
5. Copy the WAR file into `BEA_HOME/WL_HOME/common/deployable-libraries`.
6. In Workshop, navigate to **Project > Properties > Java Build Path > Libraries > Add > Manage WebLogic Shared Libraries > Add...** and locate the new library module.
7. Navigate to `weblogic.xml` in your web application, and set the JSF `libraryImplementationVersion` to exactly the version you just added.
8. Republish the application.
9. Some JSPs may throw an exception when rendered after the above changes. In these cases, go to Workshop and drag and drop any JSF tag to the page and then remove it. This will clear out the old compiled view.

