Oracle® Fusion Middleware

Federated Portals Guide for Oracle WebLogic Portal

10*g* Release 3 (10.3.6)

E14235-07

June 2013

ORACLE®

Oracle Fusion Middleware Federated Portals Guide for Oracle WebLogic Portal, 10*g* Release 3 (10.3.6)

E14235-07

Primary Author: William Witman

Contents

3 Federated Portal Architecture

Part II Development

4 Creating Remote Portlets, Pages, and Books

5 Configuring Remote Portlets

6 Offering Books, Pages, and Portlets to Consumers

7 Interportlet Communication with Remote Portlets

10 Federating User Profiles

11 Consumer Entitlement

12 Transferring Custom Data

13 WSRP Interoperability with Oracle WebCenter Portal and Oracle Portal

14 Other Topics and Best Practices

16 Configuring User Name Token Security

17 Configuring WSRP Security Between WLP and a WebCenter Portal: Framework Application

18 Adding Remote Resources to the Library

19 Configuring Two-Way SSL

List of Examples

List of Figures

List of Tables

Preface

A federated portal is a portal that includes remotely distributed resources, including remote portlets, books, and pages. These remote resources are collected and brought together at runtime to a portal application called a consumer, which presents the federated portal to end users.

This guide describes how to plan, develop, assemble, and maintain federated WebLogic Portals.

Audience

This document is intended for portal and portlet developers, and portal administrators.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the following documents in the WebLogic Portal documentation set:

- *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*
- *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Architecture

In the WebLogic Portal development life cycle, architecture represents the starting point for the subsequent phases of development, staging, and production.

This part of the Federated Portal Guide presents an architectural overview of federated portals. The chapters in this part focus on the logical components of federated portals, how these components interact, and the technologies that make federation possible. By understanding the architecture of federated portals, and specifically federated portals developed on Oracle WebLogic platforms, developers can more effectively plan their specific implementations of remote features such as remote portlets.

For a detailed description of the architecture phase of the portal life cycle, see the *Oracle Fusion Middleware Overview for Oracle WebLogic Portal*.

Part I contains the following chapters:

1 Introduction

Federated portals represent an exciting new paradigm for the development, management, testing, and deployment of portal applications. This new, Service-Oriented Architecture (SOA) based paradigm offers immediate and significant savings in time and resources to organizations that develop and manage portals using Oracle WebLogic Portal.

This guide describes how to plan, develop, assemble, and maintain federated WebLogic Portals. As the following section explains, the tasks described in this guide are organized to reflect the stages of the portal life cycle: architecture, development, staging, and production.

This chapter includes these sections:

1.1 Support for WSRP 2.0

WLP supports the WSRP 2.0 OASIS standard and fully supports all of the required WSRP 2.0 features. OASIS, the Organization for the Advancement of Structured Information Standards, is responsible for creating the WSRP standard. To read more about WSRP 2.0, including the full technical specification, go to: `http://www.oasis-open.org/committees/wsrp`. This standards compliance ensures interoperability between producers and consumers across all platforms that support WSRP 2.0.

> **Note:** All of the feature extensions, such as render dependencies and event handling, that WLP supported in its WSRP 1.0 implementation are still fully supported with the WSRP 2.0 implementation. In cases where the previously implemented feature extensions are supported by WSRP 2.0, those features have been re-implemented to comply to the WSRP 2.0 standard. Events are one such feature.

WLP also supports several optional WSRP 2.0 features, listed in Table 1–1.

Table 1–1 Optional WSRP 2.0 Features Supported by WLP

Optional WSRP 2.0 Feature	For Detailed Information
Events	See Chapter 7, "Interportlet Communication with Remote Portlets."
Full shared parameter distribution on the consumer	See "Using Shared Parameters" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.
Client attributes (cookie and header configuration)	See Section 14.14, "Managing Delivery of Headers and Cookies to the Browser."
Portlet-served resources on the producer (for JSR286 portlets only)	See "Using Container Runtime Options" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.
Importing and Exporting remote portlets	The WLP propagation tool supports the importing and exporting of remote portlets. For details on propagation, see the *Oracle Fusion Middleware Production Operations Guide for Oracle WebLogic Portal*.
Registration properties	See Section 14.19, "Storing Registration Properties."
The `wsrp-extra:doctype` extension	This extension carries the doctype URI as its value. For more information, see the section "Well Known Extensions" in the WSRP 2.0 Specification.

1.2 Federation in the Portal Life Cycle

Like a standard portal, the creation and management of a federated portal flows through a portal life cycle.

The portal life cycle contains four phases:

- Section 1.2.1, "Architecture"
- Section 1.2.2, "Development"
- Section 1.2.3, "Staging"
- Section 1.2.4, "Production"

The tasks in this guide are organized according to the portal life cycle, which implies best practices and sequences for creating and updating federated portals. For more information about the portal life cycle, see the *Oracle Fusion Middleware Overview for Oracle WebLogic Portal*.

1.2.1 Architecture

The architecture part of this guide discusses the basic components of a federated portal. A federated architecture promises to streamline and improve the way in which your portal resources, such as portlets, are developed, deployed, and maintained. By understanding the technology that lies behind federated portals, you can more effectively plan for the development of your own federated portal applications.

1.2.2 Development

The development phase of a federated portal focuses primarily on developing portlets, pages, and books that will be offered as remote portlets, pages, and books to consumers. Developers need to be aware of the techniques and best practices for developing remote portlets, pages, and books in a WebLogic Portal environment.

In the development stage, careful attention to best practices is crucial. Wherever possible, this guide makes those best practices clear.

1.2.3 Staging

As for all portal development, Oracle recommends that you deploy your portal to a staging environment, where it can be assembled and tested before going live. In the staging environment, you use the WebLogic Portal Administration Portal to assemble and configure federated portals. The Administration Portal lets you search for and consume remote portlets, books, and pages. In the staging environment, you also test your federated portal before propagating it to a live production system.

1.2.4 Production

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

1.3 Getting Started

This section describes the basic prerequisites to using this guide, lists guides containing related information and topics, and briefly explains how to use this guide.

This section includes the following topics:

- Section 1.3.1, "Prerequisites"
- Section 1.3.2, "Related Guides"
- Section 1.3.3, "Using this Guide"

1.3.1 Prerequisites

This guide does not assume that you are familiar with federation or its related standards and technologies, such as WSRP. Whenever possible, this guide provides sufficient background information or refers to appropriate documents and specifications.

> **Tip:** See Section 3.7, "For More Technical Details" for a list of specifications and white papers related to WSRP and related technology. This material provides an excellent background for developers who plan to design and create federated portals.

In general, this guide assumes that you are familiar with the basic operation of the tools used to create WebLogic portals and desktops, particularly Oracle Enterprise Pack for Eclipse (OEPE) and the Administration Portal. The following section, Section 1.3.2, "Related Guides", lists other guides that you may want to refer to before attempting to develop federated portals.

1.3.2 Related Guides

This guide covers topics that are specific to developing and assembling federated portals. In general, this guide assumes that you are familiar with the basic concepts and tools required for both portal and portlet development. If you are planning to create federated portals, we recommend that you review the following guides:

- *Oracle Fusion Middleware Overview for Oracle WebLogic Portal*
- *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*
- *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*

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

1.3.3 Using this Guide

If you are new to federation we recommend that you begin with the chapters in *Part I Architecture*. These chapters provide a detailed overview of federated portals, and describe the technological components that make up federation.

Part II Development includes the topics that are of primary interest to developers creating portal components with Oracle Enterprise Pack for Eclipse. This part includes chapters on creating remote portlets, establishing interportlet communication with remote portlets, working with producers, using custom events, and other topics.

Part III Staging and *Part IV Production* are targeted typically toward users who use the Administration Portal to assemble and manage federated portals and establish security.

2

What are Federated Portals?

This chapter presents a brief introduction to federated portals and discusses the advantages of federated portals and the kinds of problems that federation solves. This chapter includes the following sections:

2.1 Overview

A federated portal is a portal that includes remotely distributed resources, including remote portlets, books, and pages. These remote resources are collected and brought together at runtime to a portal application called a consumer, which presents the federated portal to end users. Unlike a non-federated, entirely local portal, in most cases, the individual remote parts of a federated portal can be maintained, updated, and released independently without redeploying the consumer portal in which they are surfaced.

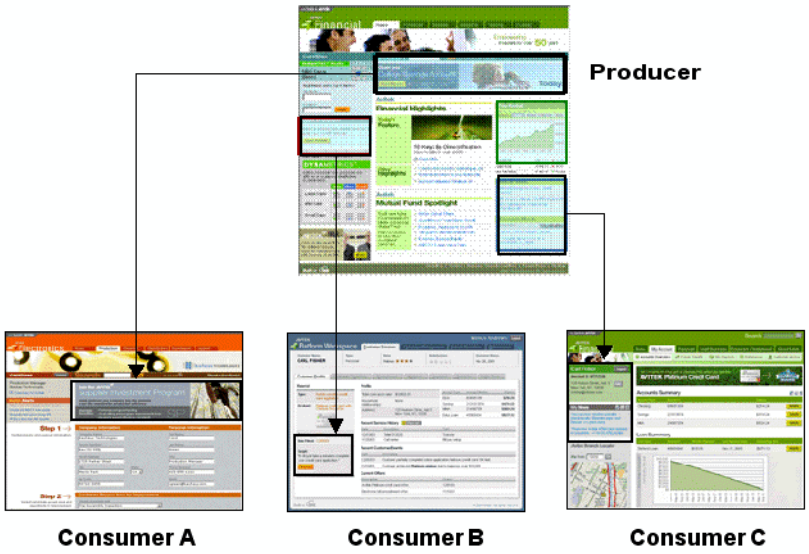
Federated portals are:

- **Distributed** – Portlets are deployed on remote systems across the enterprise.
- **Decoupled** – The portal and its portlets do not depend upon one another. In most cases, remote portlets can be maintained and deployed separately from the federated portal.
- **Collaborative** – Remote portlets can communicate and share data.
- **Plug-and-Play** – You can easily locate and use remote portlets. Programming is usually not required to consume remote portlets.
- **Standards based** – WebLogic Portal federated portals are built upon open standards, such as WSRP, SOAP, WSDL, SAML, and WS-Security. WLP supports all of the required features of the WSRP 2.0 standard.

Figure 2–1 illustrates the basic parts of a federated portal: producers and consumers. A producer is a portal web application that offers remote portlets to other portal web applications, called consumers. Both producers and consumers implement a web services layer that enable them to communicate. This web services layer allows producers to offer portlets to consumers on remote systems. Consumers bring these remote, distributed portlets together at runtime. The remote portlets themselves can be

developed and maintained by different groups of people. If one remote portlet on a producer is changed, other portlets within a consumer that consumes the updated portlet are not typically affected. Furthermore, the look and feel of a remote portlet can be made to be consistent with the federated portal in which is resides. To end users of federated portals, the remote portlets are indistinguishable from local ones.

Figure 2–1 Federated Portals Consume Remote Portlets from a Producer



Tip: A federated portal reflects a true Service Oriented Architecture (SOA). An SOA strategy organizes discrete functions contained in enterprise applications into interoperable, standards-based services that can be combined and reused quickly to meet business needs. As you will see, this definition of SOA describes well the essence of a federated portal.

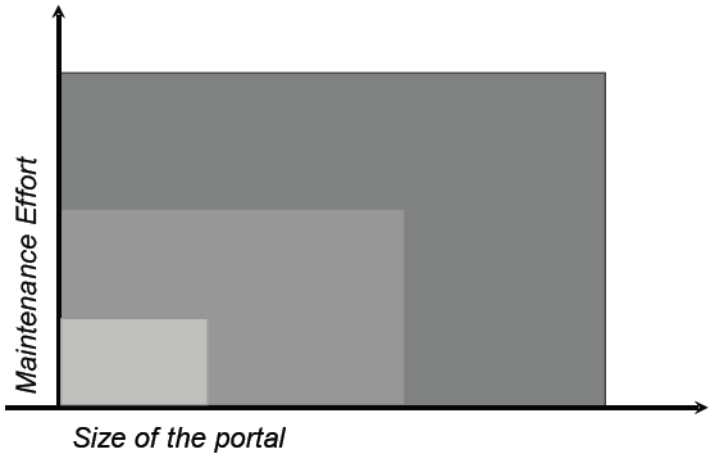
2.2 Basic Terminology

Throughout this guide, the term remote portlet refers to a portlet that is deployed in a consumer application and that references a portlet deployed in a producer application. Another term for a remote portlet is a proxy portlet. The term proxy portlet appears in some WebLogic Portal configuration files. Please note that remote portlet and proxy portlet are synonymous. In a federated environment, a producer hosts functioning portlets, while consumer applications host proxy portlets.

2.3 Traditional Portals: Before Federation

Before federation, all of a portal's portlets were deployed within the same web application. This model works well for a portal's initial deployment, but as the portal grows the maintenance effort grows proportionally, as illustrated in Figure 2–2.

Figure 2–2 Non-Federated Portal Maintenance

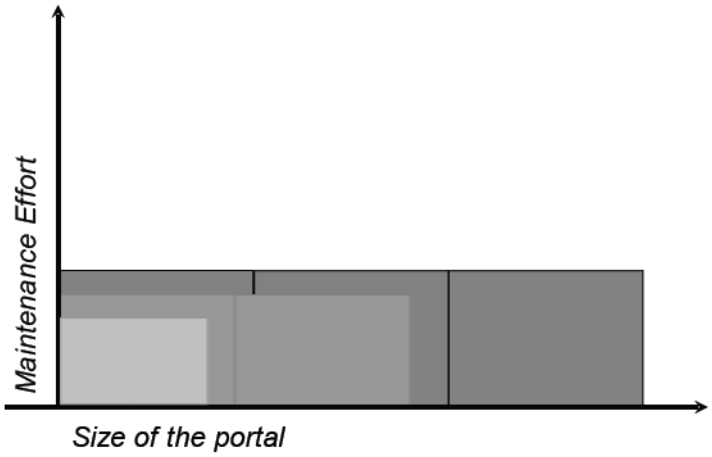


Typical portal maintenance includes bug fixes, enhancements, adding new portlets, testing, and propagating the portal from a development to a staging to a production environment. Larger portals simply contain more parts, more code, which must be bound within the same portal application, and which require the coordination of developers, quality assurance engineers, administrators, and others with each update. For many organizations, the cost of such maintenance is significant and can include portal downtime. Federation simplifies portal maintenance.

2.4 Federated Portals: A New Paradigm

With a federated portal architecture, separate development teams, perhaps in separate business units, operating in different geographical locations, can focus on and develop their respective portlets. These development teams can update, test, and release their portlets independently from one another. You do not need to redeploy a federated portal every time a portlet deployed in a producer changes. When a remote portlet is updated in a producer, all of the consumers of that portlet receive the change immediately and automatically. As illustrated in Figure 2–3, the most significant benefit of a federated portal architecture is that the maintenance effort for a portal is greatly reduced compared to a non-federated portal.

Figure 2–3 Federated Portal Maintenance



The next section further discusses the advantages of using a federated portal architecture.

2.5 Advantages of Federation

As explained in the previous section, federation offers significant benefits in portal deployment and maintenance. This section looks more closely at these and other benefits, and includes these topics:

2.5.1 Overview

Federation represents more than just a new WebLogic Portal feature. Federation represents a new paradigm for developers and administrators of portal web applications, particularly moderate to large-scale portal web applications. Central to this new paradigm are standards, such as Web Services for Remote Portlets (WSRP), that allow portlets to be decoupled from portals. For more information on WSRP, see Section 3.3, "What is WSRP?".

Rather than bundling all of a portal's portlets into a single application, you can deploy portlets in separate web applications running on remote systems while the federated portal consumes them using WSRP. Because the federated portal is decoupled from its portlets, you do not need to redeploy the portal every time a portlet changes. For most WebLogic Portal projects, this decoupling represents an immediate and significant savings in time and money.

2.5.2 Reducing the Cost of Portal Deployment

Perhaps the most significant benefit of portal federation is this: *Federated portals do not have to be redeployed when their remote portlets are updated.*

In a standard portal, all portlets are part of a monolithic enterprise application. If you want to change a portlet, even make a trivial change, the entire enterprise application must be redeployed. Likewise, adding new portlets requires redeployment. Usually, portal redeployment, particularly of large-scale enterprise portals, involves expensive testing and potential downtime.

In a federated portal, remote portlets are not part of a single enterprise application. Remote portlets are deployed in separate web applications, typically, on remote systems called producers. The federated portal consumes these portlets using standard Web Services for Remote Portlet (WSRP) and Web Services Description Language (WSDL). When you change a portlet, such as by adding or removing a feature or fixing a bug, the remote portlets that reference it automatically reflect the change. You do not have to redeploy your enterprise portal application.

2.5.3 Plug and Play SOA

A federated portal is a true example of a plug and play Service Oriented Architecture. In most cases, a portal administrator can locate a remote portlet and incorporate it into a portal without enlisting the help of a developer.

2.5.4 Increasing the Flexibility of Release Schedules

Because the portlets and other services in federated portals are distributed, multiple teams can work on and deploy new features independently of one another. Before federation, different teams had to synchronize their deployment schedules and their software configurations, such as service pack releases and software library versions. With federation, independent teams can focus on producing the best possible software solutions without such tight coupling. Through the mechanism of web services, developers of federated portals simply consume the software resources produced by these independent development teams.

2.5.5 Reducing the Cost of Testing Your Portal

Portal administrators can incorporate new remote portlets into a portal by locating a producer and picking the desired portlets. From the administrator's standpoint, these remote portlets are fully tested and ready for use. No coding, testing, or complex configuration is required by the developers or administrators of the consumer portal.

2.5.6 Decreasing Dependencies Among Software Components

If a portlet relies on specific software libraries, a strong dependency exists that must be managed. Changes to either the portlet or the library version can create incompatibilities with existing code. Because remote portlets are developed, tested, deployed, and run on remote systems, a federated portal that uses remote portlets is isolated from such dependencies.

2.5.7 Promoting Reuse of Portal Components

A portlet that is exposed through a producer can be reused by any number of consumers with minimal work and no additional coding. As mentioned previously, with federation, this reuse can be accomplished without the overhead of integration, deployment, configuration, and testing that would be required otherwise.

2.5.8 Interoperability

Because federated portals are loosely coupled and standards based, it is possible for a WebLogic Portal to consume portlets from third-party vendors. Likewise, it is possible for third-party portals to consume portlets hosted in WebLogic Portal. WLP fully supports the WSRP 2.0 standard.

3 Federated Portal Architecture

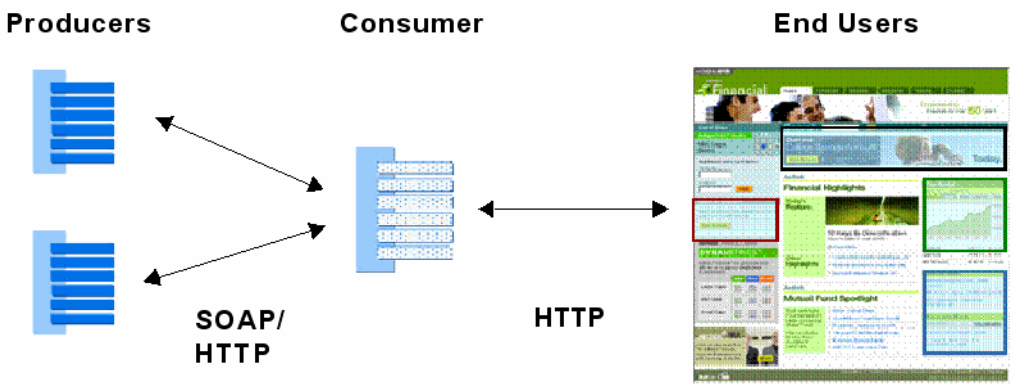
This chapter describes the key actors and logical parts of a federated portal and discusses how they interact. The information in this chapter informs many of the best practices recommended for developers of federated portals. It is helpful to review this chapter before reading Chapter 14, "Other Topics and Best Practices." In addition, this chapter discusses a key standard technology upon which federation relies: Web Services for Remote Portlets (WSRP).

This chapter includes these topics:

3.1 Key Actors in a Federated Portal

The key actors in a federated portal are producers, consumers, and end users, as illustrated in Figure 3–1.

Figure 3–1 Components of a Federated Portal



A consumer is a web application that collects remote portlets and offers them in a unified portal to end users who use a browser to view and interact with the portal. In

addition to federating portlets, WebLogic Portal lets you federate books and pages. See Section 3.2, "Federating Books and Pages" for more information.

Typically, a consumer does not include the business logic, data, or user interface parts of a portlet: it simply collects user interface markup delivered from producers and presents that user interface to users.

> **Tip:** Although most business logic processing occurs in producer applications, you can write consumer-side classes called *interceptors* that let you programmatically examine the content of a WSRP message and take specific action based on that content. Interceptors are discussed in Chapter 9, "The Interceptor Framework."

Consumers are administration-centric. This means that administrators, rather than developers, typically focus their time on consumers. Administrators locate and consume remote portlets, manage users, set up entitlements, and so on.

A producer is also a web application, typically running on a remote system from the consumer. The producer acts as a container for portlets that are offered to consumer portals. The producer is where the user interface, data, and business logic for remote portlets reside. While a consumer is administration centric, a producer is application centric. This means that developers write the actual portlet code and deploy those resources on producers.

> **Tip:** All WebLogic Portal applications are, by default, both consumers and producers. This means that every WebLogic Portal application is capable of hosting remote portlets and consuming them.

For more information on producers and consumers, see Section 3.4, "Understanding Producers and Consumers".

3.2 Federating Books and Pages

WebLogic Portal has extended the WSRP protocol to include the ability to federate books and pages. This feature is useful if you have large numbers of portlets that you want to federate. You can group the portlets in books and pages in the producer application, and consume them as a group, rather than one at a time. For more information, see Chapter 4, "Creating Remote Portlets, Pages, and Books" and Chapter 18, "Adding Remote Resources to the Library."

3.3 What is WSRP?

Web Services for Remote Portlets (WSRP) is a web services protocol for aggregating content and interactive web applications from remote sources. WSRP is a key standard that underlies federated portals. Essentially, WSRP allows remote, distributed portlets to be brought together at runtime into a unified portal page.

Web Services for Remote Portlets provide both application and presentation logic. This is different from standard web services, or data-oriented web services, which contain business logic but lack presentation logic and thus require that every client implement that logic on its own.

While the data-oriented approach works well in many implementations, it is not well suited for dynamically integrating business applications. For example, to integrate an order status web service into a commerce portal, you would need to write code to display the results of the status services into the portal. Using WSRP, with the presentation logic included in the web service, you can achieve the aggregation of

applications and services dynamically. You no longer need to develop the presentation logic in order to do the integration; you can simply request the order status service to show up as a portlet inside the commerce portal at a predetermined location.

> **Tip:** WLP fully supports the WSRP 2.0 OASIS standard. OASIS, the Organization for the Advancement of Structured Information Standards, is responsible for creating the WSRP standard. To read more about WSRP 2.0, including the full technical specification, go to: http://www.oasis-open.org/committees/wsrp.

One way to understand WSRP is to compare it with another web protocol, HTTP. The most familiar use of HTTP is viewing and interacting with remote web applications using a browser. Using HTTP, browsers communicate with remote HTTP servers to post data (for example, by submitting forms) and to retrieve markup (typically, HTML). WSRP is a similar protocol between server and client applications. In WSRP terminology, the server is called a producer. It hosts services, typically portlets, that clients, or consumers, communicate with.

Like a browser in the HTTP analogy, the consumer retrieves markup and submits user interactions to the producer. The producer hosts actual portlets while the consumer contains proxies for those portlets. Consumers use WSRP to collect and present markup from the remote portlets to end users who interact with that markup. To an end user, a remote or proxy portlet is indistinguishable from a local portlet.

WSRP consumers are more sophisticated than browsers, however. Unlike browsers, consumers can:

- Offer features like personalization, customization, and security
- Handle markup fragments rather than entire HTML documents
- Combine markup from different producers into a single page and apply consistent consumer-specific layouts and styles to that page

In summary, the WSRP protocol defines a set of web services that WSRP producers implement. Consumers view and interact with these web services; they retrieve user interface fragments from the producer, display the fragments, and allow users to interact with them. The WSRP protocol allows consumers to act as clients for applications hosted by producers.

3.4 Understanding Producers and Consumers

This section focuses on the producer and consumer implementations for WebLogic Portal. This section includes these topics:

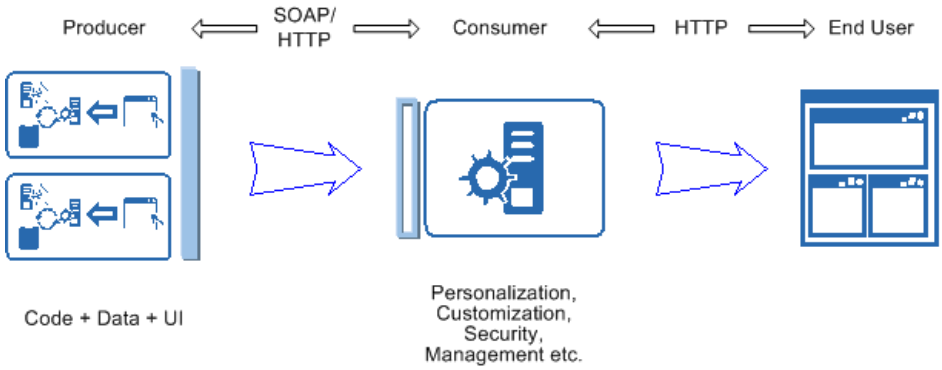
- Section 3.4.1, "Overview"
- Section 3.4.2, "WebLogic Portal Producers"
- Section 3.4.3, "WebLogic Portal Consumers"

3.4.1 Overview

A producer is a container web application that hosts portlets. Through proxy portlets (called "remote portlets" in WLP), consumers collect and present portlets hosted on producers to users. All application code (backing files, Java classes, controls, EJBs, and so on) resides on the producer. Consumers only receive fragments of markup from producers which are collected and presented to users.

Figure 3–2 illustrates the components of a federated portal. Note that the WebLogic Portal WSRP implementation allows the addition of typical WebLogic Portal services, such as personalization, customization, and user management. This means that remote portlets are given the same look and feel and the same levels of portal security as local portlets.

Figure 3–2 Web Services Between Producer and Consumer



Tip: Every WebLogic Portal contains both producer and consumer implementations. That is, all WebLogic Portals can function as producers and consumers. For an in-depth technical explanation of the WebLogic Portal producer and consumer implementations, refer to the technical white paper, *Inside WSRP* at `http://www.oracle.com/technology/pub/articles/dev2arch/2005/03/inside_wsrp.html`.

3.4.2 WebLogic Portal Producers

WebLogic Portal supports two kinds of producers: simple and complex. Before describing these two kinds of producers, it is helpful to understand the parts of the WSRP protocol and which operations must be implemented in a producer (required operations) and which are optional.

Table 3–1 lists the set of required and optional operations defined by the WSRP protocol. Note that the minimum requirement for a WSRP-compliant producer is to implement the required service description and markup operations. As you will see, WebLogic Portal simple and complex producers differ in the kinds of operations they support.

Table 3–1 Required and Optional WSRP Operations

WSRP Protocol	Operations	Implemented Methods
Required for WSRP	Service description operations	getServiceDescription()
	Markup operations	initCookie()
		getMarkup()
		performBlockingInteraction()

Table 3–1 (Cont.) Required and Optional WSRP Operations

WSRP Protocol	Operations	Implemented Methods
Optional for WSRP	Registration	register()
	Portlet Management	modifyRegistration()
		deregister()
		getPortletPropertyDescription()
		setPortletProperties()
		getPortletProperties()
		clonePortlet()
		destroyPortlets()
Extensions	Event Handling	handleEvents()
(Add new operations to the WSRP protocol.)	Render Dependencies	getRenderDependencies()

3.4.2.1 Simple Producers

A simple producer supports only some basic features of the WSRP protocol. These basic features do not require the producer to be deployed in a full WebLogic Portal domain. You can, for example, deploy a simple producer in a basic WebLogic Server domain.

> **Tip:** For detailed information on configuring a simple producer in a WebLogic Server domain, see Chapter 8.

A simple producer:

- **Does not depend upon WebLogic Portal features** – A simple producer cannot take advantage of WebLogic Features features such as user management and personalization.
- **Does not depend on WebLogic Portal APIs** – Again, a simple producer cannot rely on any WebLogic Portal dependencies.
- **Does not require registration** – Registration allows a producer to associate portlets and any portlet customization data with the consumer that is interacting with it. The producer can also use the registration to tailor the scope of the portlets offered to specific consumers.
- **Does not support event handling** – You cannot use the event handling API with a simple producer.

Despite these limitations, you might want to use a simple producer for the following reasons:

- To WSRP-enable non-portal projects, such as WebLogic Server projects
- To offer portlets without actually installing WebLogic Portal

The section Section 8.2, "Using WSRP in a Basic WebLogic Server Domain" describes how to configure a (non-portal) WebLogic Server environment as a WSRP producer so that you can expose portlets based on Struts or Java Page Flows. The exposed portlets can then be consumed as remote portlets running in a regular WebLogic Portal Domain.

Note: Page flows are a feature of Apache Beehive, which is an optional framework that you can integrate with WLP. Apache Struts is also an optional framework that you can integrate with WLP. See "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

3.4.2.2 Complex Producers

A complex producer supports the complete WSRP 2.0 protocol plus some extensions for interportlet communication, portlet look and feel, and other features. A complex producer also lets you take advantage of other WebLogic Portal features, such as personalization, customization, and user management security features. By contrast, simple producers cannot take advantage of these WebLogic Portal features.

By default, all WebLogic Portal applications are complex producers. Portlets that are exposed in a complex producer can use the APIs and features that are available in any WebLogic Portal application.

Tip: In some cases, it is inappropriate to use API calls in portlets deployed on a producer. This is because a producer does not have access to portal artifacts, such as books and pages, in that are deployed in consumer applications. See Chapter 14 for information on best programming practices for portlet development in producers.

Typically, a complex producer:

- **Requires registration** – Registration allows a producer to associate portlets and any portlet customization data with the consumer that is interacting with it. By deploying consumer entitlements, the producer can also use the registration to tailor the scope of the portlets offered to specific consumers. For detailed information on consumer entitlements, see Chapter 11, "Consumer Entitlement." By default, registration is enabled; however, you can disable registration by setting the `<registration required>` element in the `/WEB-INF/wsrp-producer-config.xml` file to `false`.
- **Supports a management interface** – By default, the WebLogic Portal management interface is enabled; however, you can disable the management interface by setting the `<portlet-management required>` element to `false` in the file `/WEB-INF/wsrp-producer-config.xml`.
- **Supports interportlet communication** – Extensions that support event handling allow remote portlets to communicate with one another.
- **Supports portlet render dependencies** – WebLogic Portal allows you to specify certain dependencies associated with individual portlets, such as Cascading Style Sheets (CSS files) and script files, such as JavaScript (JS) files.

3.4.2.3 Summary of Complex and Simple Producers

A complex producer includes the required WSRP interfaces, optional interfaces, and some extended interfaces. A simple producer implements the required interfaces. A complex producer requires WebLogic Portal, but a simple producer can be deployed in a basic WebLogic Server domain. Figure 3–3 illustrates these relationships.

Figure 3–3 Simple and Complex Producers

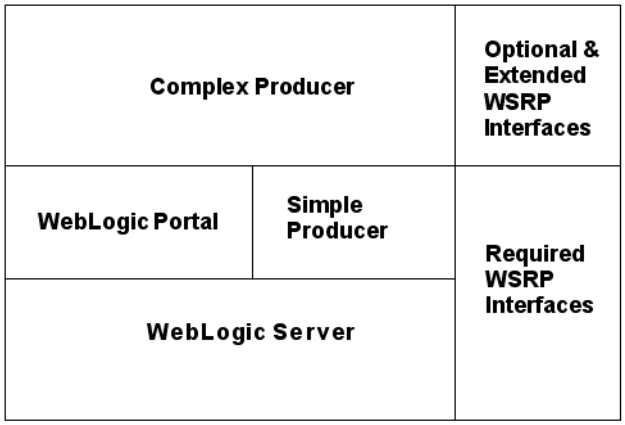


Table 3–2 compares the capabilities of standard and complex producers.

Note: Page flows are a feature of Apache Beehive, which is an optional framework that you can integrate with WLP. See "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

Table 3–2 Comparison of Producer Features

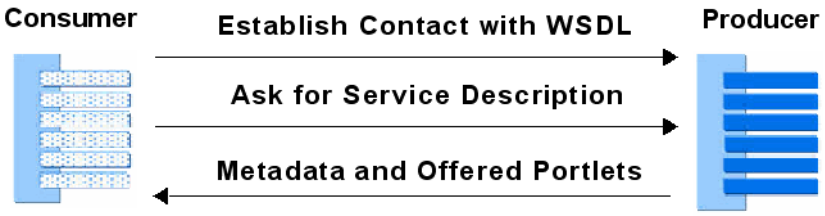
Feature	Complex Producer	Simple Producer
Java portlets	Yes	No
Page flow portlets	Yes	Yes
Registration	Required	Not Supported
Support for URL rewriting (producer and consumer)	Yes	Yes
Support for portal administration	Yes	No
Support for JSP portlets	Yes	No
Support for backing files	Yes	No
Support for JSF portlets	Yes	No
Support for Struts portlets	Yes	Yes

3.4.3 WebLogic Portal Consumers

As previously noted, all WebLogic portals are, by default, able to consume portlets hosted on producers. The WebLogic Portal consumer implementation is closely integrated into the WebLogic Portal framework.

To consume a remote portlet hosted on a producer, a consumer must ask a producer for information about the portlets it offers. The consumer first contacts a producer using the producer's WSDL URL. This initial contact verifies the availability of the producer and its services. Next, the consumer asks the producer for a description of the portlets it offers. The producer responds to the consumer with a SOAP message that describes the portlets and associated metadata that are offered by the producer. This communication is illustrated in Figure 3–4.

Figure 3–4 Getting the Service Description for a Producer



After a consumer receives a producer's metadata, the metadata is added to the consumer enabling you to create remote portlets. A remote portlet is a proxy to a portlet hosted on a producer. When a remote portlet is added to a portal or desktop, the WebLogic Portal framework uses the WSRP protocol to present the portlet to portal users. To users, remote portlets look and feel just like local portlets; users are not aware that a given portlet is hosted remotely. Furthermore, remote portlets inherit the particular styles, layouts, and themes from the portal in which they reside. To the user, this integration is seamless.

> **Tip:** As noted previously, WebLogic Portal lets you create consumer-side classes called interceptors. Interceptors let you programmatically examine the content of a WSRP message and take specific action based on that content. Interceptors are discussed in Chapter 9, "The Interceptor Framework."

3.4.4 Cookie Handling

WebLogic Portal consumers handle cookies by following the prescriptions of RFC2109:

1. A `Set-Cookie` response header whose `NAME` is the same as a previous cookie, and whose `Domain` and `Path` attribute values exactly (string) match those of the previous cookie, will replace the old cookie with the new one.
2. If the `Set-Cookie` has a value for `Max-Age` of zero, the (old and new) cookie is discarded.
3. Otherwise cookies accumulate until they expire (resources permitting), at which time they are discarded.
4. Cookies are sent based on the specified request-host (including request-URI) and should be sent until they expire.
5. In WSRP, cookies are specific to the `portletHandle` and the end user on whose behalf the consumer is invoking the producer and may only be resupplied for this specific pair (the `portletHandle` is relaxed to one from a group for cookies returned from `initCookie()` when `ServiceDescription.requiresInitCookie=perGroup`.)

3.5 Life Cycle of a Remote Portlet

A remote portlet goes through a well defined life cycle. The steps of this life cycle that are executed depend on which of the following scenarios is requested:

- The portlet is simply being rendered (or re-rendered).
- A user is interacting with the portlet (submitting a form, for instance).
- An event is fired.
- The portlet has render dependencies

It is important to realize that these life cycle phases are decoupled from one another. As explained later in this section, this decoupling has implications for developers writing portlets hosted on producers. For example, you cannot expect a portal to receive the same HTTP response or request for the render phase as it receives for an interaction.

This section does not address the ways in which interceptors can influence the remote portlet life cycle. Interceptors are consumer-side classes that intercept WSRP messages and allow you to programatically take specific actions based on the content of those messages. Interceptors are discussed in Chapter 9.

> **Tip:** The information in this section informs many of the best practices for developers discussed in Chapter 14, "Other Topics and Best Practices." It is particularly important for developers creating portlets in a producer to understand the life cycle of a remote portlet. By understanding this life cycle, you will avoid making unwarranted assumptions and avoid common mistakes.

This section includes the following topics:

- Section 3.5.1, "Rendering a Remote Portlet"

  Rendering occurs independently of user interaction in a remote portlet. The render phase does not allow a remote portlet's state to change. It happens, for instance, when a portal page is refreshed.

- Section 3.5.2, "Interacting With a Remote Portlet"

  The interaction phase occurs when a user interacts with a remote portlet, for example, by submitting a form or clicking a link.

- Section 3.5.3, "Rendering Versus Interaction"

  This section summarizes the differences between rendering and interaction.

- Section 3.5.4, "Interportlet Communication with Events"

  A third life cycle for remote portlets occurs when events are fired. Events provide the best mechanism for interportlet communication between remote portlets.

- Section 3.5.5, "Retrieving Render Dependencies"

  A fourth life cycle for remote portlets occurs a portlet deployed on a producer includes render dependencies.

> **Tip:** This section provides an overview of the remote portlet life cycle phases. For an in-depth technical review of this subject, refer to the white paper, Inside WSRP at `http://www.oracle.com/technology/pub/articles/dev2arch/2005/03/inside_wsrp.html`.

3.5.1 Rendering a Remote Portlet

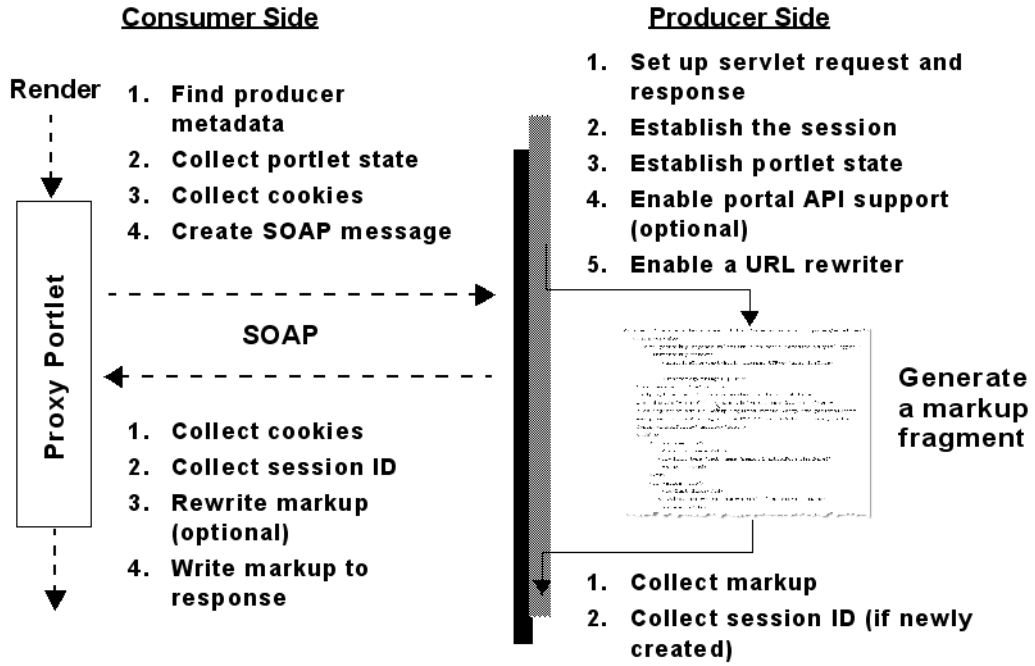
A well defined series of steps occurs whenever a remote portlet is rendered in a consumer portal. Anytime a portlet needs to be re-rendered in exactly the same state (for example, if the user simply refreshes a page), the rendering phase is executed. If a user directly interacts with a remote portlet, then a different phase, called the interaction phase is triggered. The interaction phase is discussed in the next section.

With a typical (non-federated) web application, when you send a request to a JSP, for instance, you receive back the markup for the requested page. In a federated portal, the user is viewing a page that consists of markup fragments received from portlets

hosted on producers (or, possibly a mix of local portlets and portlets deployed on producers). The consumer's job is to contact the producers, retrieve their markup, and render it in a unified portal page.

In Figure 3–5 a portlet exists on a producer, and a proxy for that portlet (a remote portlet) has been created in a consumer portal. The sequence of steps needed to render the portlet in the consumer are listed, and discussed in the following sections.

Figure 3–5 Rendering a Remote Portlet



3.5.1.1 Initial Steps on the Consumer

To render a remote portlet, a consumer must first compose a SOAP message to send to the producer. These initial steps include:

1. Find producer metadata.

   The consumer's first job is to find the metadata for a producer. When a developer or administrator creates a remote portlet, metadata about each producer is received and stored internally by WebLogic Portal (on the consumer).

2. Collect the portlet state. The state consists of a view state and a navigational state:

   - **View state** – This includes the mode (view mode or edit mode) and the state (minimized or maximized).
   - **Navigational state** – For example, if a user has already filled in a form and submitted it, the navigational state reflects the fact that the user has moved from page one to page two of the portlet. Knowledge of this state allows remote portlets to be re-rendered correctly any number of times.

3. Collect all cookies.

   Just as a browser acts as a client to a web server, a consumer acts as a client to a producer. For example, a browser maintains cookies that keep track of sessions on the server. In the same way, a consumer maintains cookies that keep track of the

producer sessions for each user of the portal. For each user interacting with a consumer, the consumer maintains one session with each producer.

4. Create a SOAP message.

   All of the information gathered by the consumer must be formed into a SOAP message and sent to the producer.

3.5.1.2 Initial Steps on the Producer

After the producer receives the SOAP message from the consumer, the producer must take the following steps to render the requested portlet and return the portlet's markup to the consumer.

1. Set up servlet request and response objects.
2. Establish a session.
3. Establish the portlet state.

   In steps 1, 2, and 3, the producer creates an HTTP environment for the portlet. Because the producer receives a SOAP request, and not an HTTP request, the producer must take the information in the SOAP request and recreate the appropriate HTTP environment for the portlet, including such things as the servlet request and response objects, the session, and the portlet state.

4. Enable portal API support (optional).

   The producer must decide whether or not to offer complex features. WebLogic Portal has implemented WSRP extensions and optional interfaces. These extensions and options allow WebLogic Portal producers to offer features such as user management, entitlements, portlet preferences, and event handling. In some cases, you may want to deploy a producer portal without these extra capabilities; therefore, the producer must determine whether or not to enable them. For more information on simple versus complex producers, see Section 3.4, "Understanding Producers and Consumers".

5. Create a URL rewriter.

   In a traditional web application, URLs in, for instance, JSP pages, always point to the web server hosting the JSP page. In a federated portlet, URLs must always point back to the consumer, not to the producer. This is because the producer might, in fact, be inaccessible to the user clients. The producer may be located behind a firewall, for instance. To properly manage URLs, the producer contains URL rewriters that know how to create URLs that are consumer oriented.

   > **Tip:** If you are developing portlets in a producer, always be sure to use WebLogic Portal APIs and tags to create URLs. These APIs and tags know how to generate URLs so that they function properly in a federated environment. For more information, see Section 14.4, "Avoid Coupling by URL".

6. Generate markup for the portlet.

   At this stage, the producer renders the portlet. The producer may have created a new session for the portlet or added new cookies. The producer collects all this information and generates a response to the consumer.

7. Collect markup and the session ID (if one was created), and send this data back to the consumer.

3.5.1.3 Final Steps on the Consumer

The consumer receives from the producer markup fragments from the producer. The consumer must take these fragments and compose them into a portal page that can be displayed to the user. To do this, the consumer takes these final steps:

1. Collect cookies sent from the producer.
2. Collect the session ID for each portlet.
3. Rewrite markup (optional).
4. Write markup to the response.

This cycle repeats each time the portlet is rendered, as long as caching is not enabled on the consumer.

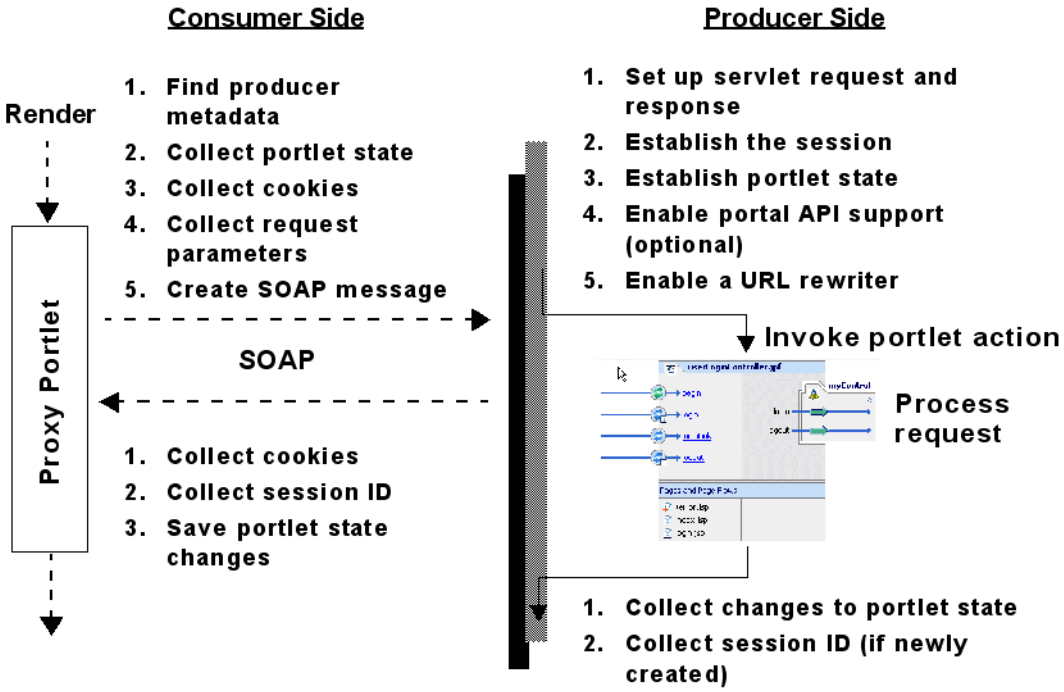
3.5.2 Interacting With a Remote Portlet

Just as with the rendering life cycle described in the previous section, interaction with a remote portlet follows a well-defined series of steps. Interaction occurs, for example, when a user submits a form through a JSF portlet. Typically, a JSP on the producer performs some background action, such as executing business logic, and prepares a response.

As you will see, the steps taken for remote portlet interaction are similar to those taken for simple rendering. The differences are highlighted in Figure 3–6 and described in this section.

> **Tip:** It is crucial to understand that the rendering and interaction phases of the remote portlet life cycle are decoupled. This decoupling has important consequences on how you develop portlets in a producer. You cannot expect a portal to receive the same HTTP response or request for the render phase as it receives after an interaction. For more information on this and other practical advice, see Chapter 14, "Other Topics and Best Practices."

Figure 3–6 Remote Portlet Interaction Life Cycle



3.5.2.1 Initial Steps on the Consumer

When a user interacts with a remote portlet, the consumer must first compose a SOAP message to send to the producer. These initial steps include the following. Steps highlighted in bold differ from the render phase described previously.

1. Find producer metadata.
2. Collect the portlet state.
3. Collect all cookies.
4. **Collect request parameters.**

   Because the user is interacting with the portlet, the request parameters must be collected and returned to the producer.
5. Create a SOAP message.

   All of the information gathered by the consumer must be formed into a SOAP message and sent to the producer.

3.5.2.2 Initial Steps on the Producer

After the producer receives the SOAP message from the consumer, the producer must take the following steps to invoke the portlets action, generate markup for the requested portlet, and return the portlet's markup to the consumer. Steps highlighted in bold differ from the render phase described previously.

1. Receive the SOAP request from the consumer.
2. Create an HTTP environment for the portlet.
3. Decide whether or not to offer extended features.
4. Create a URL rewriter.

5. **Invoke the portlet's action.**

   In this step, the actions submitted by the consumer must be replayed on the producer. The producer is not directly aware of where the request for this action comes from. After the business logic is executed, a page must be prepared that displays the results of the logic. For instance, if the user submitted a login form, after a successful login, the portlet must return another page that tells the user that the login was successful.

6. **Collect changes to the portlet state.**

   After the request is processed, the producer must collect changes to the portlet's state. This state is returned to the consumer in the form of a markup fragment that can be collected and displayed for the end user.

7. Collect the session ID, if a new session was created.

8. Sends the markup back to the consumer.

   These steps are the same as for the render phase described previously.

3.5.2.3 Final Consumer Steps

Steps highlighted in bold differ from the render phase described previously.

1. Collect cookies.

2. Collect the session ID for each portlet.

3. **Save portlet state changes.**

   Portlet state information is maintained on the consumer.

4. Rewrite markup (optional)

5. Write markup to the response.

This cycle repeats each time the portlet is rendered, as long as caching is not enabled on the consumer.

3.5.3 Rendering Versus Interaction

Both the rendering and interaction phases are available to any remote portlet. The render phase is required in cases where the user does not directly interact with a portlet, but the portlet needs to be refreshed anyway. For instance, if a user interacts with one portlet on a page, she doesn't want the other portlets on the page to change or disappear.

Because the render phase is not always driven by a user's interaction, it is idempotent (the producer returns the same portlet state that the consumer submits). This makes sense, because if you simply refresh a page, you do not want to regenerate data from a database. Likewise, in the render phase, mode and state changes are not allowed.

> **Tip:** Separate rendering and interaction phases are not unique to remote portlets; local portlets incorporate a rendering and interaction phase as well.

Table 3–3 summarizes the characteristics of these two phases. These two phases are decoupled so that a portlet's state can only change if you interact with it. If, for instance, you submit a form from Portlet A, and then interact with Portlet B in the same portal page, you do not want the state or view of Portlet A to change when it is refreshed. A portlet's state must only change if you interact with it directly. The

interaction phase is always driven by user interaction, while the render phase can happen any number of times and is driven arbitrarily.

Table 3–3 Render Versus Interaction for Remote Portlets

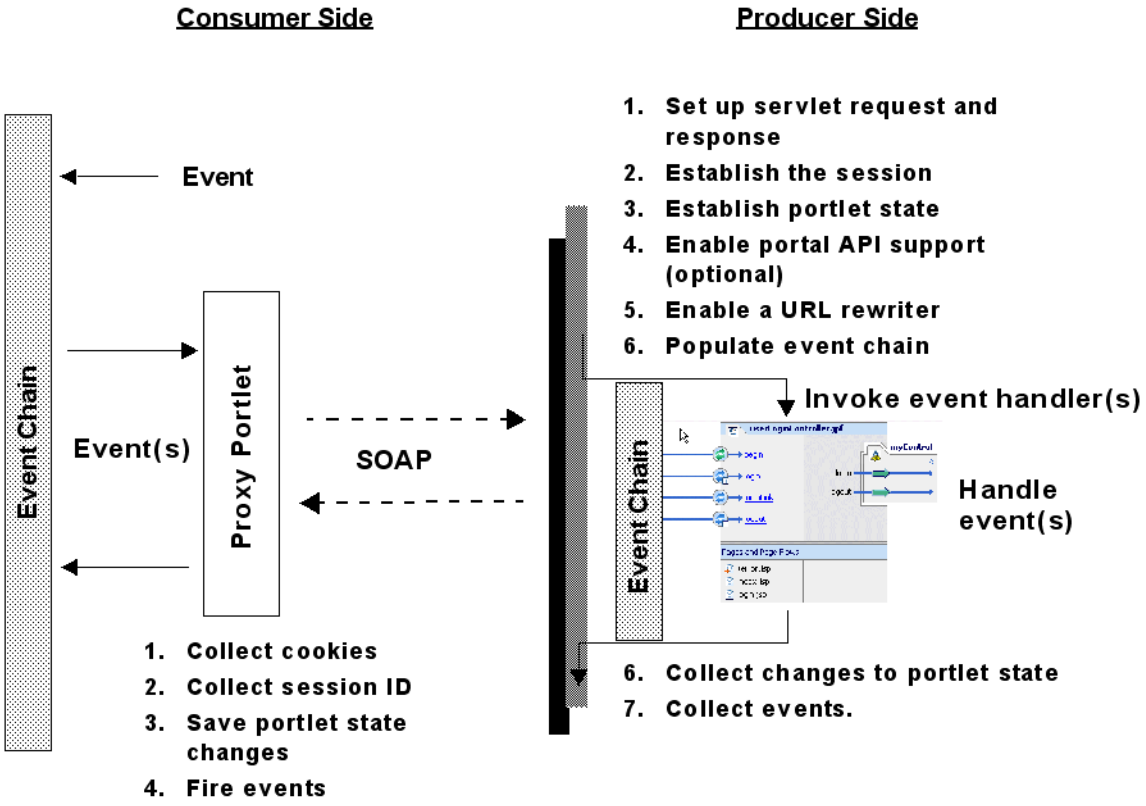
Render Phase	Interaction Phase
Focus on presentation (view)	Focus on business logic (controller)
May be replayed several times	Driven by user interaction
Idempotent ■ No state/mode changes ■ No changes to portlet preferences ■ Cannot redirect the user	Non-idempotent ■ Can ask for mode/state changes ■ Can change portlet preferences Can redirect users
Generates markup	Optionally generates markup.

3.5.4 Interportlet Communication with Events

To facilitate interportlet communication with events, WebLogic Portal extended the WSRP protocol to add a third phase in the remote portlet life cycle. This extension allows a portlet to fire an event during its interaction phase. You can register events with remote portlets; however, when a proxy portlet receives an event, it cannot process it locally because it is a proxy. The remote portlet must send the event to the producer for processing. The portlet on the producer then fires the event and the producer handles it as appropriate.

> **Note:** The WebLogic Portal IPC framework is location independent. This means that the framework is not concerned with where an event originated. Portlets in consumers and producers can both listen for and fire events.

Figure 3–7 Event Handling Phase



This phase is similar to the interaction phase. The primary difference is that in the interaction phase, the portlet gets the user interaction data and in response it changes the navigational state of the portlet. The portlet can also fire events.

In the event processing phase, the portlet does not receive a user interaction; rather, it always gets an event fired by another component. In response to the event, the producer can change the state of the portlet and can generate more events, which are stored in an event chain. If it generates events, then the cycle repeats.

3.5.5 Retrieving Render Dependencies

WebLogic Portal allows you to specify certain dependencies associated with individual portlets. Dependencies typically include Cascading Style Sheets (CSS files) and script files, such as JavaScript (JS) files. Portlet dependencies are configured in an XML file that is referenced in a `.portlet` file. The dependencies file explicitly lists the CSS and script files upon which the portlet depends.

WebLogic Portal has extended the WSRP protocol to allow remote portlets to retrieve render dependencies from producers.

3.6 Summary of Federated Portal Architecture

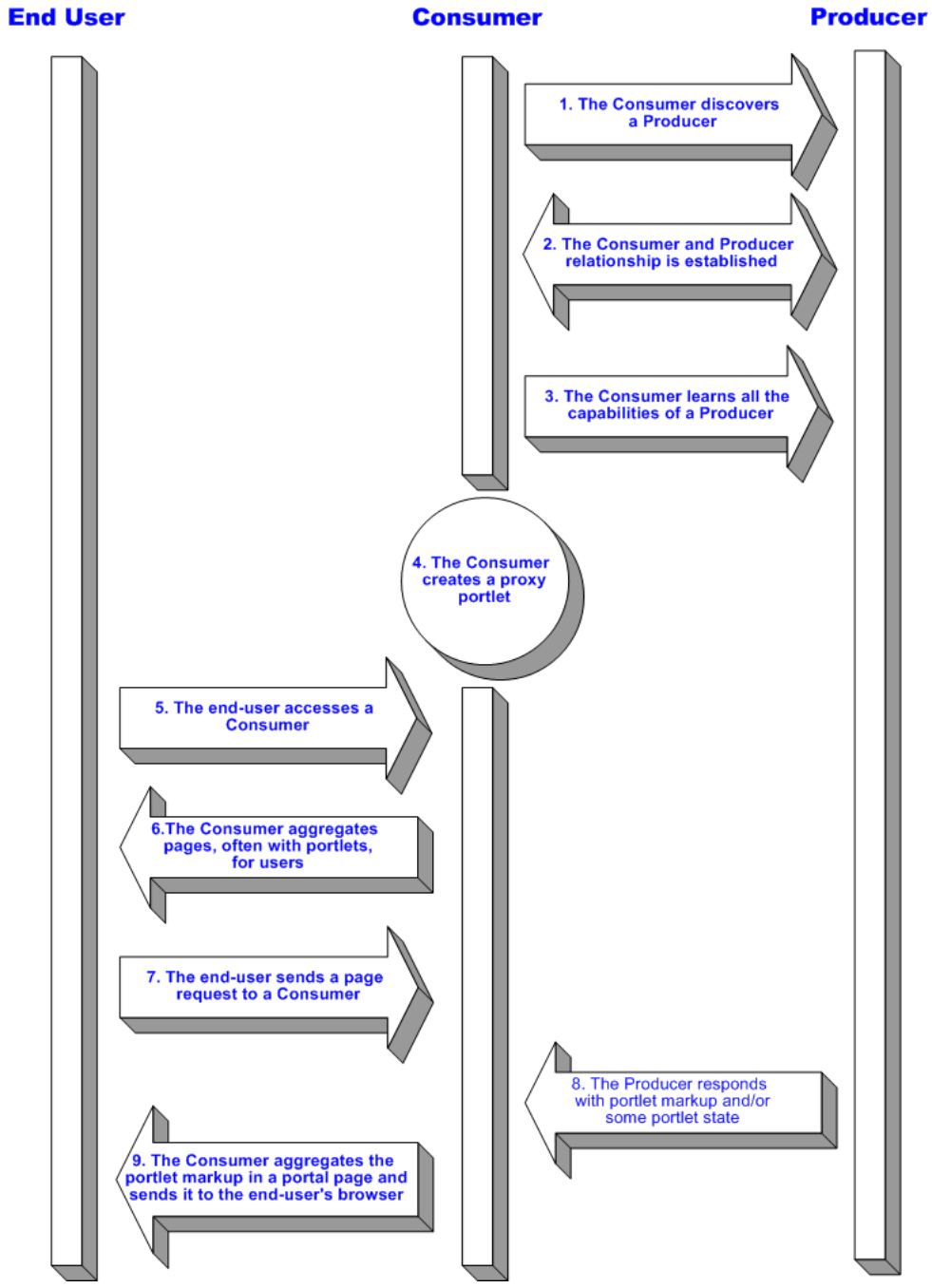
In summary, WebLogic Portal applications can act as consumers and/or producers. WebLogic Portals are configured to handle:

- Communication with multiple producers
- Federation from producers running WebLogic Server or other non-WebLogic Portal applications.

- Aggregation of remote portlets
- Addition of personalization, customization, security management, look and feel, and other typical WebLogic Portal features to remote portlets
- Displaying remote portlets to end users as rendered HTML content

Figure 3–8 shows a sequence diagram depicting the flow of actions between end users, consumers, and producers. This diagram highlights the notion that a consumer acts as an intermediary between a producer and an end user.

Figure 3–8 WSRP Sequence Diagram



3.7 For More Technical Details

If you are interested in learning more about the technical details of the WebLogic Portal implementation of WSRP, you can refer to the following:

- *Inside WSRP* at http://www.oracle.com/technology/pub/articles/dev2arch/2005/03/inside_wsrp.html

  This Oracle white paper discusses in detail the messaging that occurs between consumers and producers and highlights best practices for developers writing portlets that will be hosted on producers.

- *WSRP v.1.0 Primer* at http://www.oasis-open.org/committees/wsrp.

  The purpose of this document is to provide a tutorial-oriented explanation of the main concepts of the WSRP 1.0 specification. This document is maintained by OASIS, the Organization for the Advancement of Structured Information Standards. OASIS is responsible for creating the WSRP standard.

- *Web Services for Remote Portlets Specification Version 1.0* at http://www.oasis-open.org/committees/wsrp.

  This is the official specification for WSRP version 1.0. This document was also created by and is maintained by OASIS.

Part II

Development

Some of the tasks described in this chapter require you to use Oracle Enterprise Pack for Eclipse features to create remote portlets, implement interportlet communication, and create interceptors. Some features, such as interceptors, require Java coding expertise. Other features, such as user profile mapping, require you to edit configuration files.

For a detailed description of the development phase of the portal life cycle, see the *Oracle Fusion Middleware Overview for Oracle WebLogic Portal*.

Part II contains the following chapters:

4

Creating Remote Portlets, Pages, and Books

This chapter focuses on how to use Oracle Enterprise Pack for Eclipse to create and configure remote portlets, pages, and books. This chapter includes the following sections:

4.1 Introduction

Before getting started, we recommend that you review the following chapters in the architecture part of this guide:

- Chapter 2, "What are Federated Portals?"
- Chapter 3, "Federated Portal Architecture"

.These chapters explain basic concepts such as producers, consumers, and remote portlets, pages, and books. This chapter assumes you familiar with these concepts.

4.2 What Types of Portlets Can Be Remote?

WebLogic Portal applications can consume the following types of portlets:

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

> **Note:** Page flows are a feature of Apache Beehive, which is an optional framework that you can integrate with WLP. Apache Struts is also an optional framework that you can integrate with WLP. See "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

To be consumable, a portlet must be deployed to a web application that is running in a WSRP-compliant producer. The consumer application must also be capable of contacting the producer using the producer's WSDL URL.

4.3 Creating a Remote Portlet

This section presents a step-by-step example showing you how to create a remote (proxy) portlet in a consumer application using Oracle Enterprise Pack for Eclipse. This section includes the following topics:

4.3.1 Overview

For this example, you will consume a portlet deployed in a producer application. The producer application in this example is a Portal Web application deployed to a WebLogic Portal Domain.

> **Tip:** For information on working with a producer that is running in a WebLogic Server domain (as opposed to a WebLogic Portal Domain), see Chapter 8, "Configuring a WebLogic Server Producer."

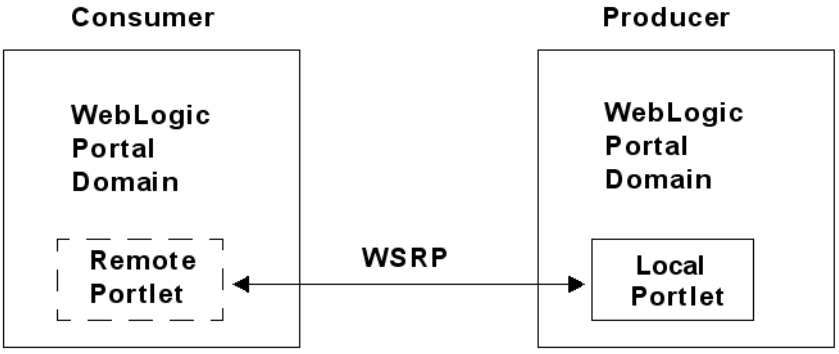
The basic procedure for creating remote portlets is fairly simple: Oracle Enterprise Pack for Eclipse provides a convenient wizard for this purpose. No programming is required. The basic steps always include:

1. Locating a producer.
2. Selecting a remote portlet on the producer.
3. Consuming the portlet.

Figure 4–1 illustrates the basic parts of a federated portal, where the consumer includes a remote portlet. A remote portlet is a proxy for a portlet that is deployed in a producer application.

> **Tip:** To an end user, the features of the remote portlet are indistinguishable from the actual portlet deployed on the producer. It is possible, however, to customize many of the properties of a proxy portlet, as explained in Chapter 5, "Configuring Remote Portlets."

Figure 4–1 Remote Portlet in a Consumer



4.3.2 Setting Up the Example

If you want to try the example discussed in this section, you need to run Oracle Enterprise Pack for Eclipse and perform the prerequisite tasks outlined in this section.

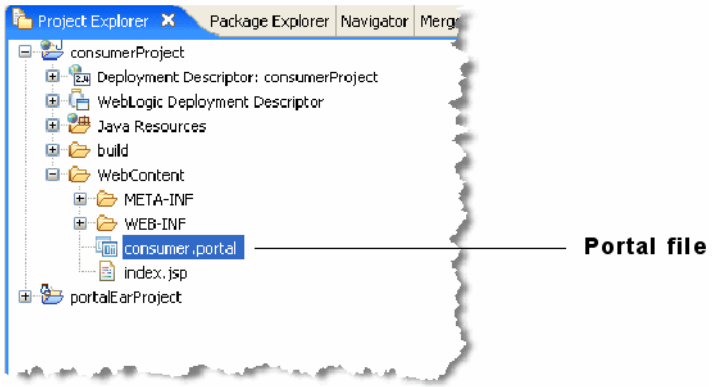
To set up the example environment, perform the prerequisite tasks outlined in Table 4–1. If you are not familiar with the specific procedures for these tasks, they are described in detail in *Oracle Fusion Middleware Tutorials for Oracle WebLogic Portal*.

Table 4–1 Prerequisite Tasks

Task	Recommended Name
Create a WebLogic Portal domain.	`consumerPortalDomain`
Create a Portal EAR Project.	`portalEarProject`
Create an Oracle WebLogic Server v10.x.	N/A
Associate the EAR project with the server.	N/A
Create a Portal Web Project and add it to the EAR.	`consumerProject`
Create a portal.	`consumer.portal`

Figure 4–2 shows the Project Explorer after the prerequisite tasks have been completed.

Figure 4–2 Project Explorer After Prerequisite Tasks are Completed



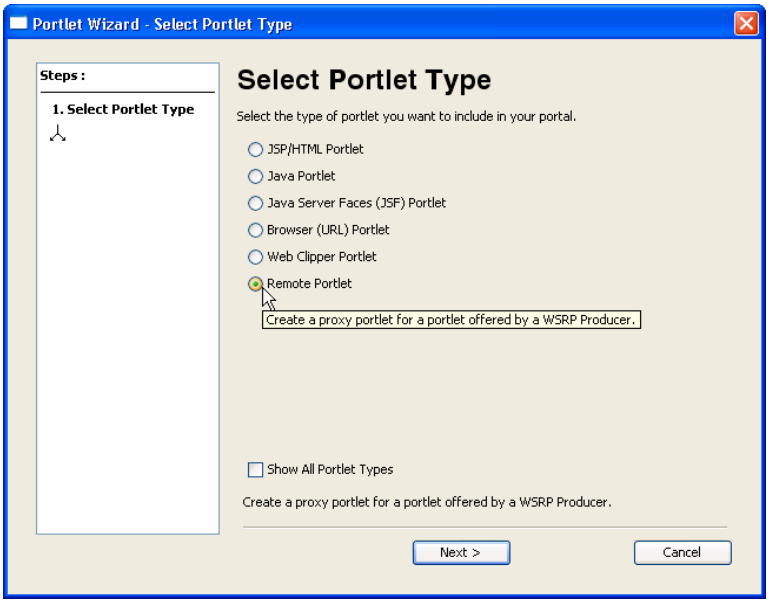
4.3.3 Locating and Consuming a Portlet

1. Be sure you have set up the example environment as explained previously in Section 4.3.2, "Setting Up the Example".
2. Open the consumerProject folder in the Project Explorer, right-click on the WebContent folder, and select **New** > **Portlet**.

   Tip: If you do not see the Portlet feature on the New menu, be sure to open the Portal perspective using **Window** > **Open Perspective** > **Portal**.

3. In the New Portlet dialog, enter `remoteExample.portlet` in the File name field, and click **Finish**. The Select Portlet Type dialog appears.
4. In the Select Portlet Type dialog, select **Remote Portlet**, as shown in Figure 4–3, and click **Next**. The Portlet Wizard – Producer dialog box appears.

Figure 4–3 Select Portlet Type Dialog



5. In the Portlet Wizard – Producer dialog, select **Find Producer** and, in the field provided, enter a WSDL URL. For example:

   `http://wsrp.bea.com/portal/producer?wsdl`

   Just remember that the pattern for the URL is as follows:

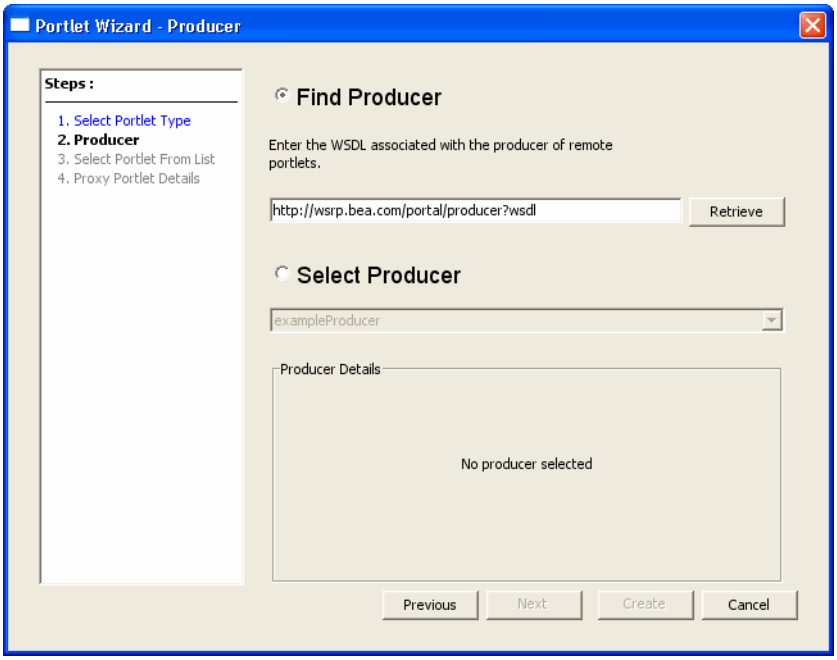
   `http://`*`host`*`:`*`port`*`/`*`webAppName`*`/producer?wsdl`

   where host is the *`host`* and *`port`* are the hostname and port number of the server on which the producer is deployed, and *`webAppName`* is the name of the web application in which the producer's portlets are deployed.

   Tip: If the producer was previously added to the consumer, it will appear in the Select Producer list, and you can simply choose **Select Producer** and select the producer from the list.

Tip: WSDL stands for Web Services Description Language and is used to describe the services offered by a producer. For more information, see Chapter 3, "Federated Portal Architecture."

Figure 4–4 Entering the WSDL



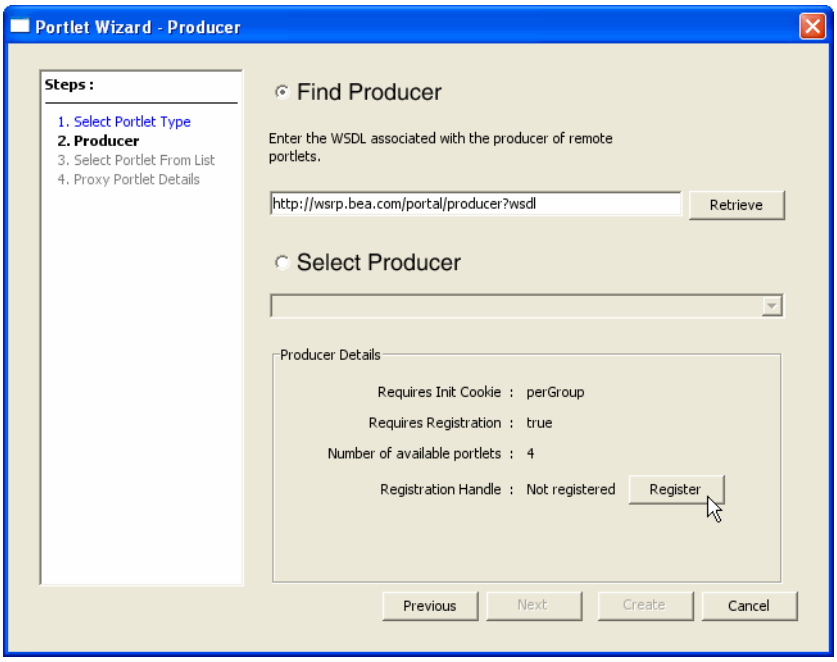
6. After entering the WSDL URL, click **Retrieve**.

   Checkpoint: At this point, the consumer uses the WSDL to locate the producer and learn about its available portlets. The Producer Details section of the wizard panel now displays information about the producer, including the number of portlets that are available to the consumer, as shown in Figure 4–5.

7. Click **Register** in the Producer Details section of the wizard panel, as shown in Figure 4–5. The Register dialog appears.

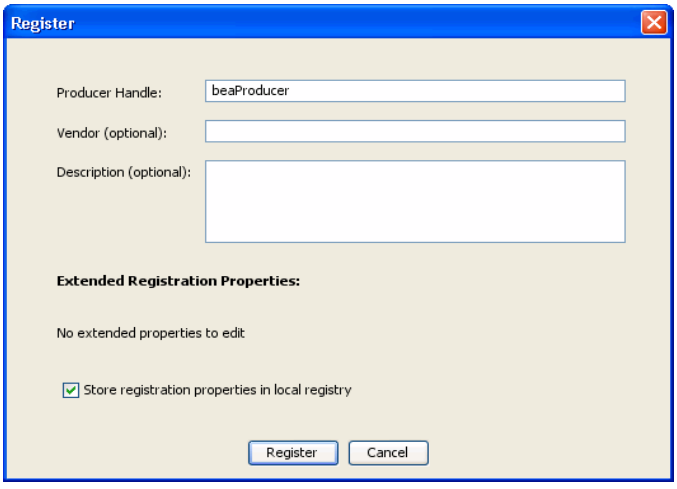
   Tip: During registration, the producer stores information about the consumer and returns a handle to the consumer. Registration is an optional feature described in the WSRP specification. A WebLogic Portal complex producer implements this option and, therefore, requires consumers to register before discovering and interacting with portlets offered by the producer. See Section 3.4.2.2, "Complex Producers" for more information.

Figure 4–5 Producer Details



8. In the Register dialog, enter `beaProducer` in the Producer Handle field, as shown in Figure 4–6. This handle identifies the producer on the consumer. This dialog also lets you choose to store registration properties on the consumer and (if available) edit the registration properties Registration properties are values that are passed from the consumer to the producer when the producer is registered. These values can be used to allow producers to control which portlets are offered to specific consumers. For detailed information storing registration properties, see Chapter 14, "Other Topics and Best Practices."

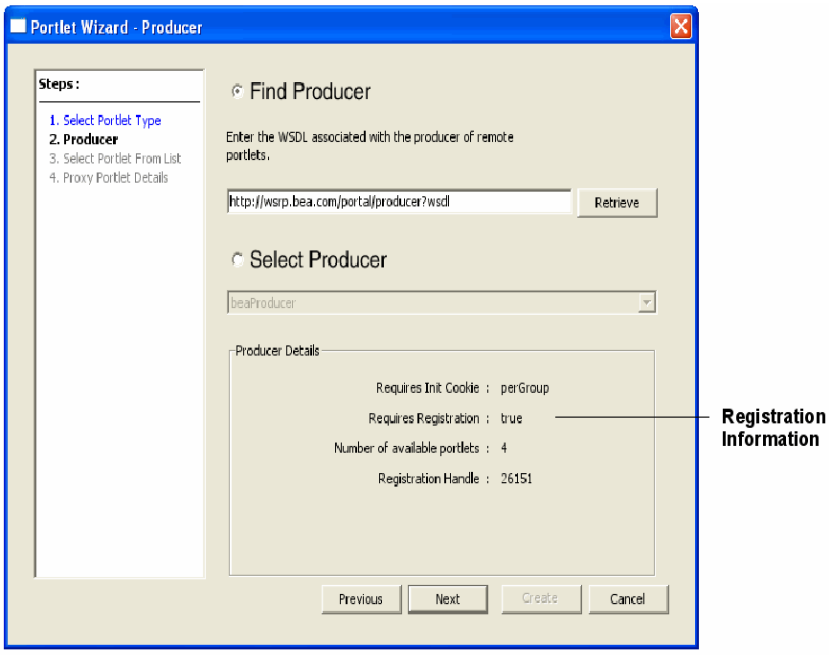
Figure 4–6 Registering the Producer



9. Click **Register**. You are returned to the Producer dialog.

   Checkpoint: At this point the WSDL data from the producer has been retrieved and is displayed in the Producer Details panel of the dialog, as shown in Figure 4–7. Note that four portlets on the producer are available to the consumer.

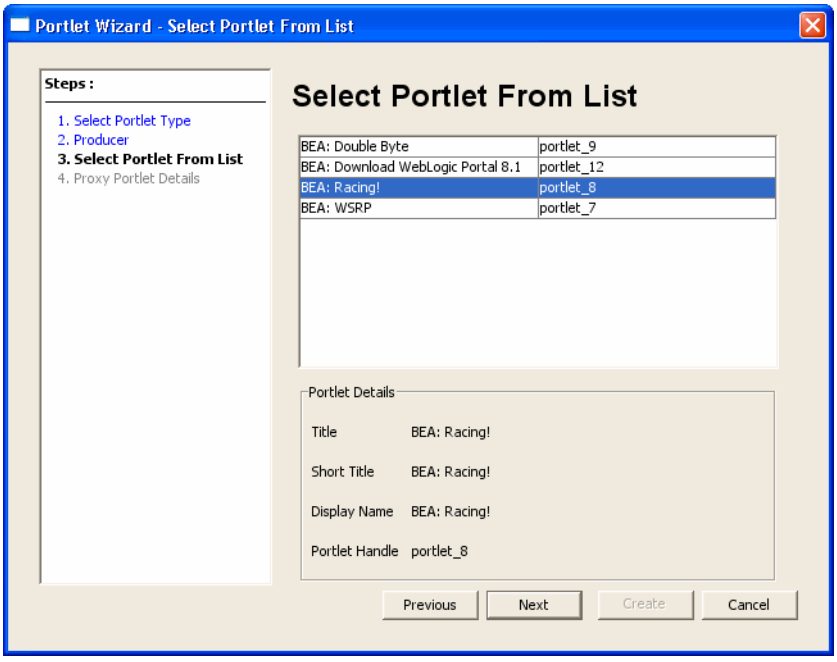
Figure 4–7 Registration Information



10. Click **Next** to proceed.

11. In the Select Portlet from List dialog, select one of the remote portlets from the list of portlets on the producer, as shown in Figure 4–8. You can pick any one of them.

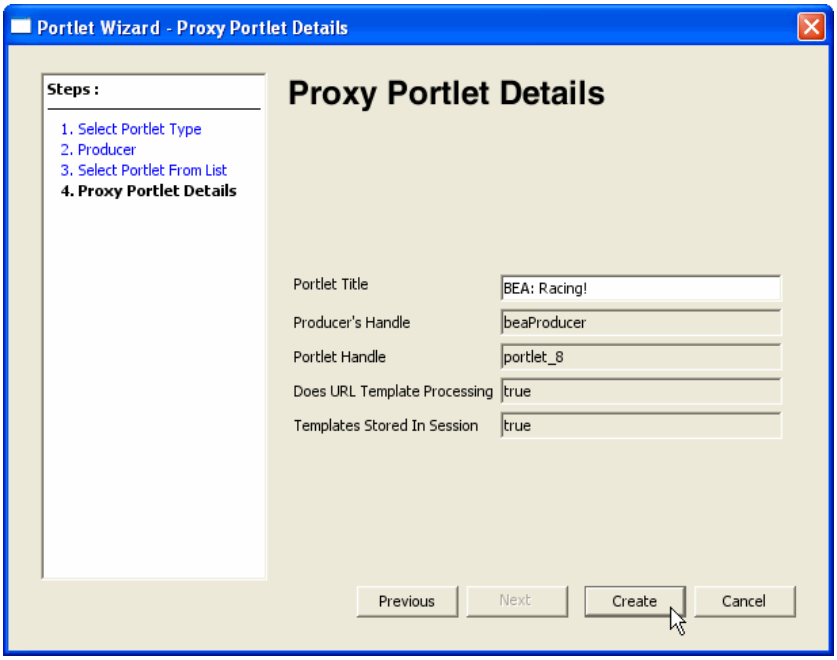
Figure 4–8 Select a Portlet on the Producer



12. Click **Next**. The Proxy Portlet Details dialog appears. The title of the portlet you selected appears in the Portlet Title field, as shown in Figure 4–9. You can change this title if you want to.

Note: If the producer does not require registration, the Producer's Handle field will appear editable and initialized. In this case, you must enter an arbitrary value for the producer handle before the **Create** button will be enabled.

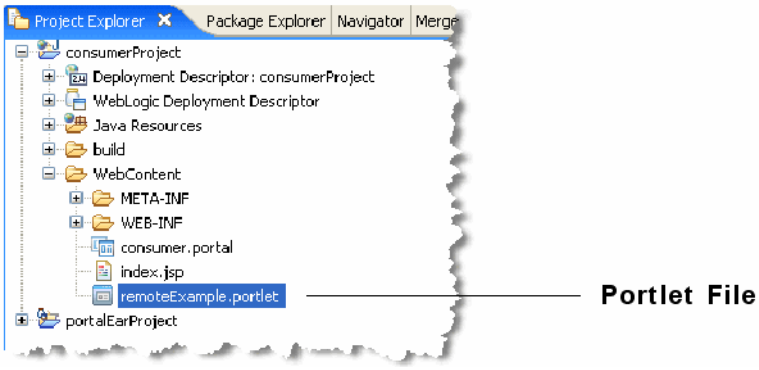
Figure 4–9 The Proxy Portlet Details



13. Click **Create**.

    The new remote portlet shows up in the Project Explorer in the WebContent folder, as shown in Figure 4–10.

Figure 4–10 Remote Portlet



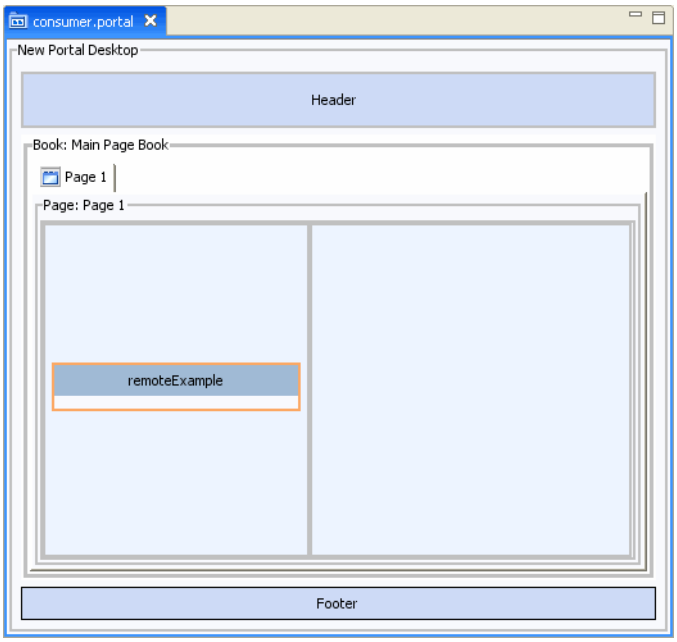
4.3.4 Viewing the Portlet

To view the portlet, you need to add it to the consumer portal, as explained in this section.

1. If it is not already open, open the consumerProject/WebContent folder.
2. Double-click the file **consumerPortal.portal** in WebContent folder. The portal editor appears in Oracle Enterprise Pack for Eclipse.

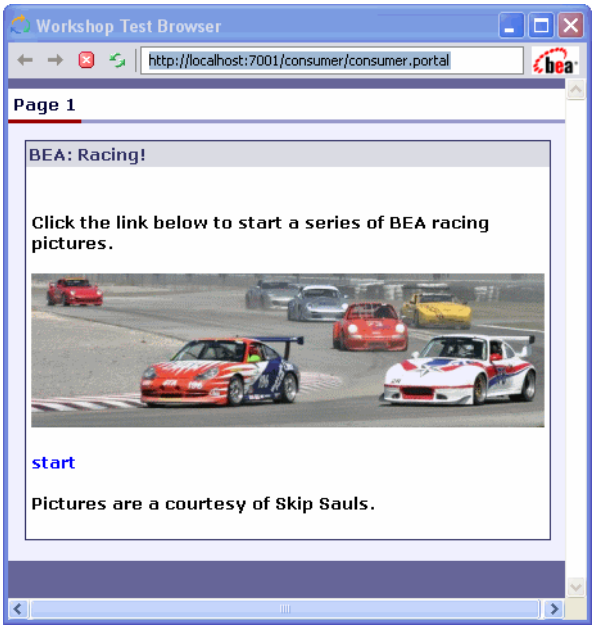
3. Drag the `remoteExample.portlet` file from the Project Explorer to the portal. The result is shown in Figure 4–11.

Figure 4–11 Remote Portlet Placed in Portal



4. To test the portal, right-click the portal filename, **remoteExample.portlet**, in the Project Explorer, and select **Run As** > **Run On Server**. The New Server – Define a Server dialog appears.
5. In the Run On Server – Define a New Server dialog, be sure the **Oracle WebLogic Server v10.x** is selected, and click **Finish**.
6. The portal containing the remote portlet appears in a browser, as shown in Figure 4–12.

Figure 4–12 Federated Portal



4.3.5 Summary

In this section you added a remote portlet to a WebLogic Portal consumer application. The consumed portlet is a proxy for a portlet that is deployed in a remote producer application. In addition to the basic setup steps, this example demonstrated the following tasks:

- Discovering the producer using its WSDL URL
- Registering the producer
- Selecting a portlet from the producer
- Adding the remote portlet to a consumer portal
- Running a consumer portal

4.4 Creating Remote Pages and Books

The primary advantage of remote books and pages is that they act as containers for other remote resources. For example, a producer can offer a remote book that contains several remoteable pages, each of which contain multiple remoteable portlets. When you consume that book, the remoteable pages and portlets it contains are consumed as well—no additional steps are required.

> **Tip:** The term remoteable refers to a book, page, or portlet that is deployed in a producer application and that is offered as remote to consumers. Application developers decide whether or not books, pages, and portlets they create are offered as remote.

For detailed information on creating remoteable pages and books in a producer application, see Chapter 6, "Offering Books, Pages, and Portlets to Consumers." If a remote book or page does not appear as you expect it to, see Section 6.4, "Rules for Creating Remoteable Books and Pages".

Remember that changes you make to a remote book or page are not reflected back to the producer; therefore, after a remote book or page is modified on the consumer, it can become inconsistent with the original book, page, or portlet in the producer application.

You can add remote books and pages to your portal as you would any other book or page.

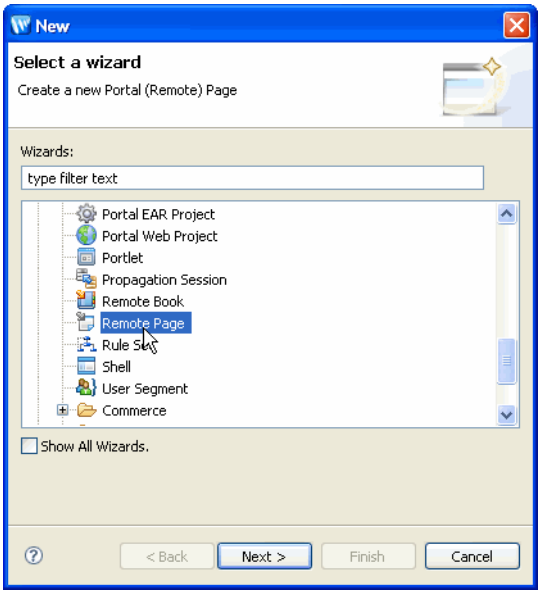
4.4.1 Basic Procedure

The procedure for creating a remote page or book is similar to the procedure for creating a remote portlet.

> **Tip:** This example explains how to create a remote page. The procedure for creating a remote book is similar.

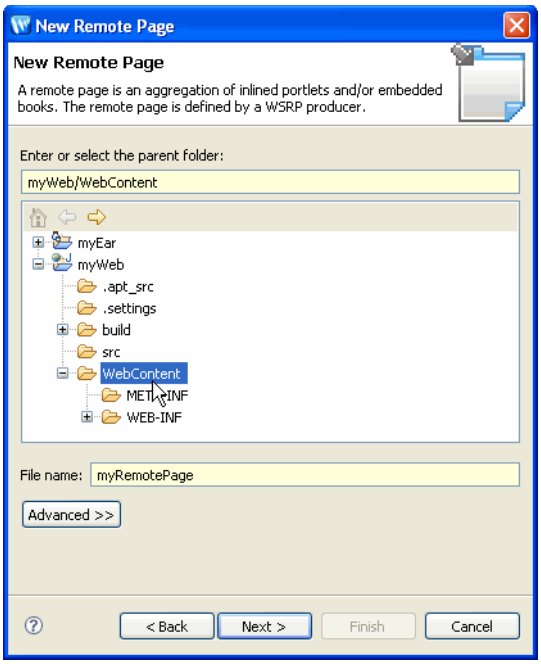
1. Select **File** > **New** > **Other**.
2. In the New dialog, select **Remote Page** (or **Remote Book**), as shown in Figure 4–13, and click **Next**.

Figure 4–13 Creating a New Remote Page



3. In the New Remote Page dialog, select a folder in which to place the resulting `.page` file, and give the file a name, as shown in Figure 4–14, and click **Next**.

Figure 4–14 Creating a Remote Page File



4. In the Page Producer dialog, select **Find Producer** and, in the field provided, enter the WSDL URL of the producer, as shown in Figure 4–15:

   The pattern for the WSDL URL is as follows:

   `http://host:port/webAppName/producer?wsdl`

   where *host* and *port* are the hostname and port number of the server on which the producer is deployed, and *webAppName* is the name of the web application in which the producer's remoteable pages and books are deployed.

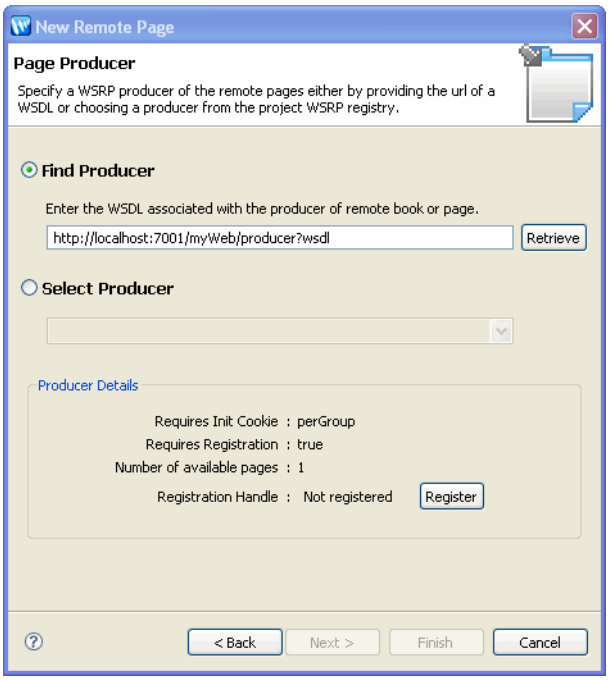
   > **Tip:** If the producer was previously added to the consumer, it will appear in the Select Producer list, and you can simply choose **Select Producer** and select the producer from the list.

5. After entering the WSDL URL, click **Retrieve**.

   Checkpoint: At this point, the consumer uses the WSDL URL to locate the producer and learn about its available remoteable pages. The Producer Details section of the wizard panel now displays information about the producer, including the number of pages that are available to the consumer, as shown in Figure 4–15.

6. Click **Register** in the Producer Details section of the wizard panel, as shown in Figure 4–15. The Register dialog appears.

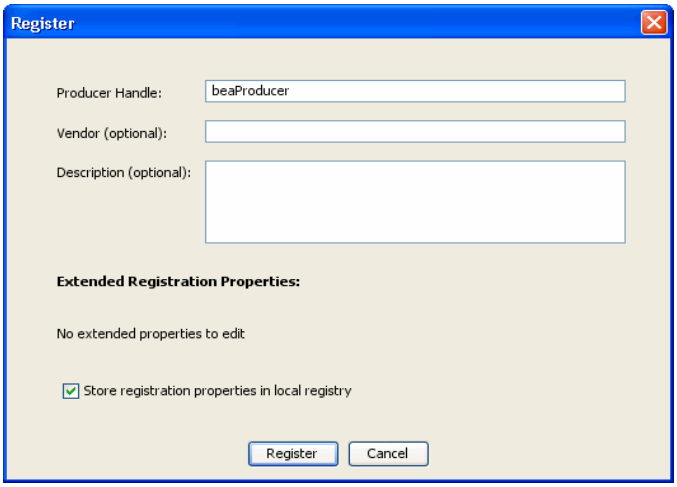
Figure 4–15 Producer Details



Tip: During registration, the producer stores information about the consumer and returns a handle to the consumer. Registration is an optional feature described in the WSRP specification. A WebLogic Portal complex producer implements this option and, therefore, requires consumers to register before discovering and interacting with books and pages offered by the producer. See Section 3.4.2.2, "Complex Producers" for more information.

7. In the Register dialog, enter a handle name in the Producer Handle field, as shown in Figure 4–6. This handle identifies the producer on the consumer.

Figure 4–16 Registering the Producer

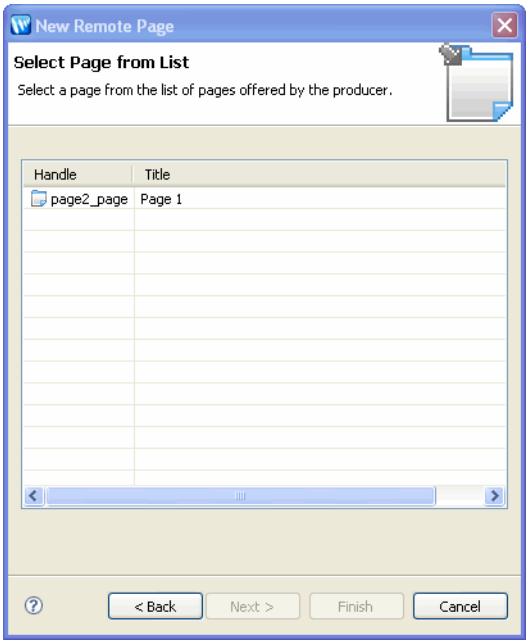


8. Click **Register**. You are returned to the New Remote Page–Page Producer dialog.

Checkpoint: At this point the WSDL data from the producer has been retrieved and is displayed in the Producer Details panel of the dialog.

9. Click **Next** to proceed.
10. In the Select Page from List dialog, select one of the remote pages from the list of pages offered by the producer, as shown in Figure 4–8, and click **Next**.

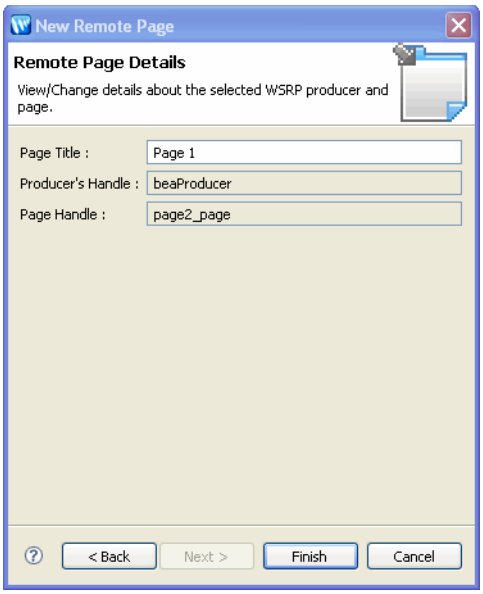
Figure 4–17 Select a Page on the Producer



11. The Remote Page Details dialog appears. The title of the page you selected appears in the Page Title field, as shown in Figure 4–9. You can change this title if you want to.

Note: If the producer does not require registration, the Producer's Handle field will appear editable and uninitialized. In this case, you must enter an arbitrary value for the producer handle before the Finish button will be enabled.

Figure 4–18 The Remote Page Details Dialog



12. Click **Finish**.

The new remote page shows up in the Project Explorer in the folder you selected. The page appears as a `.page` file, for example, `/WebContent/myRemotePage.page`. You can now add the remote page to your portal. Any remoteable portlets on the page will show up in the remote page as inlined portlets. For more information on remoteable portlets, pages, and books as remote, see Chapter 6, "Offering Books, Pages, and Portlets to Consumers." For information on inlined portlets, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

5 Configuring Remote Portlets

This chapter discusses ways you can modify and configure remote portlets within Oracle Enterprise Pack for Eclipse.

This chapter includes the following sections:

5.1 Applying a Look and Feel to a Remote Portlet

The look and feel of a portlet determines the appearance of a portlet on the portal desktop. A remote portlet's look and feel is not linked to a producer, giving you the option of modifying the portlet's appearance on the consumer. This capability allows you to match the appearance of the consumer portal in which the proxy portlet resides.

Specific procedures for applying a look and feel to a portlet are documented elsewhere. Please refer to the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal* for detailed information on these topics:

- Creating Look and Feels
- Look and Feel Architecture
- The Portal User Interface Framework
- How Look and Feel Determines Rendering
- Style Sheet Class Reference
- Creating Skins and Skin Themes
- Creating Skeletons and Skeleton Themes

5.2 Modifying Modes and States in a Remote Portlet

This section explains how to modify a remote portlet's modes and states and includes these topics:

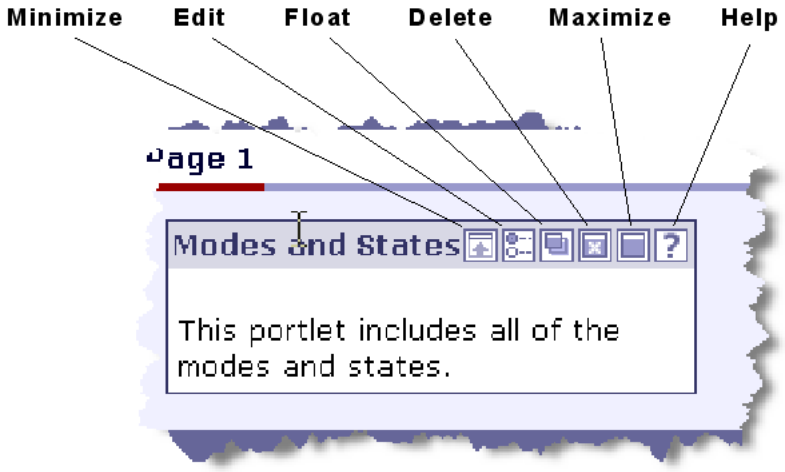
- Section 5.2.1, "What are Modes and States?"
- Section 5.2.2, "Modes and States in Remote Portlets"
- Section 5.2.3, "Changing Modes and States in Remote Portlets"

5.2.1 What are Modes and States?

A portlet's title bar can contain up to six buttons. These buttons provide convenient functions called modes and states.

Figure 5–1 shows an example portlet with all of the modes and states enabled.

Figure 5–1 Portlet with Modes and States



The modes include:

- **Edit** – Activates a custom file that lets you modify the portlet's content.
- **Help** – Activates a help file.

The states include:

- **Minimize** – Minimizes the portlet.
- **Maximize** – Maximizes the portlet.
- **Delete** – Removes the portlet from the portal.
- **Float** – Displays the portlet in a separate window.

For more detailed information on modes and states, how they work, and how to add and configure them in portlets, refer to the *Portlet User Guide*.

5.2.2 Modes and States in Remote Portlets

Table 5–4 describes how states are transferred by default from a portlet deployed on a producer to its remote proxy in a consumer application. The table also indicates whether or not the state is editable in the remote portlet.

Table 5–1 Default Behavior of States in Remote Portlets

State of Producer Portlet	Default State of Proxy Portlet	Is the Proxy Portlets's State Editable?
Delete = `true`	Delete = `true`	No
Delete = `false`	Delete = `false`	No
Maximize = `true`	Maximize = `true`	No
Maximize = `false`	Maximize = `false`	No
Minimize = `true`	Minimize = `true`	No
Minimize = `false`	Minimize = `false`	No
Float = `true`	Float = `true`	No
Float = `false`	Float = `false`	No

Table 5–4 describe how modes are transferred by default from a portlet deployed on a producer to its remote proxy in a consumer application. The table also indicates whether or not the mode is editable in the remote portlet. For instance, if the Help mode is set in the portlet deployed on the producer, it is also set in the remote proxy; however, you cannot remove it from the remote proxy. On the other hand, if Help is not set in the portlet deployed on the producer, you are free to add it to the remote portlet.

Table 5–2 Default Behavior of Modes in Remote Portlets

Mode of Producer Portlet	Default Mode of Proxy Portlet	Is the Proxy Portlets's Mode Editable?
Help = `true`	Help = `true`	No
Help = `false`	Help = `false`	Yes
Edit = `true`	Edit = `true`	No
Edit = `false`	Edit = `false`	Yes

Note: Both the help and the edit mode each reference a file that provides appropriate content for those actions. For example, the help mode references a help file. For these modes to work in a proxy portlet, the files they reference must exist on the consumer in the same relative location as they exist on the producer system.

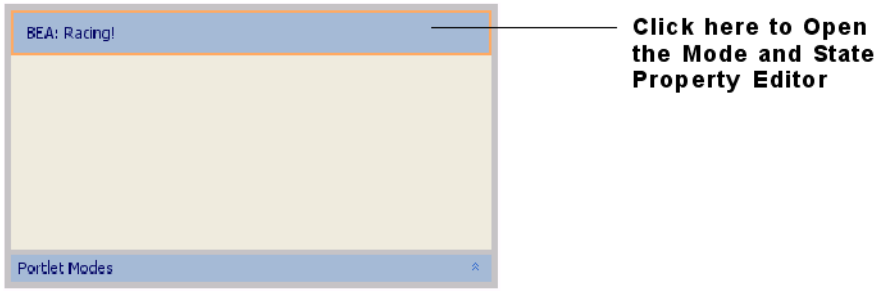
5.2.3 Changing Modes and States in Remote Portlets

All of the modes and states that are available in local portlets are available in their remote proxies. Note, however, that when you create a remote portlet, it is not possible to edit (add or remove) all of the modes and states in the remote portlet. In addition, the Float state is always turned off in a remote portlet by default; however, you are free to add it to the remote portlet in the consumer application if you wish.

The procedure for changing the default mode and state settings in a remote portlet is the same as with a local portlet.

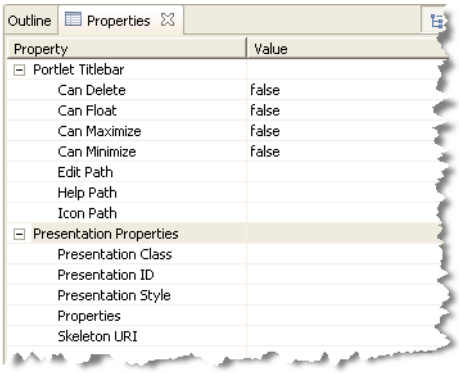
1. Double-click the portlet file in the Package Explorer view to open it in the editor.
2. Click in the header portion of the portlet in the editor, as shown in Figure 5–2. This opens the Portlet Titlebar properties in the Properties view, as shown in Figure 5–3

Figure 5–2 Click in the Header of the Portlet



3. Click on the Portlet Titlebar values to change them.

Figure 5–3 Header Properties View



5.3 Handling Errors in Remote Portlets

Under some circumstances, a remote portlet may be unable to access its producer. In this case, the consumer throws an exception. This section explains how to handle this exception by displaying an error page.

There are two ways to configure an error page for a remote portlet to be displayed if the remote portlet is unable to connect to its producer. You can configure the page in Oracle Enterprise Pack for Eclipse or in the remote portlet's XML file.

> **Tip:** For finer control of error handling, consider using interceptors. The interceptor framework is described in Chapter 9, "The Interceptor Framework."

This section includes these topics:

- Section 5.3.1, "Configuring an Error Page in Oracle Enterprise Pack for Eclipse"
- Section 5.3.2, "Configuring an Error Page in the .portlet File"

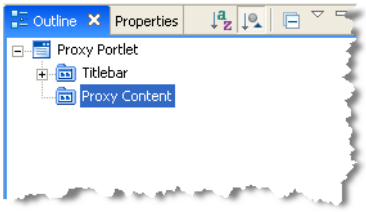
5.3.1 Configuring an Error Page in Oracle Enterprise Pack for Eclipse

To configure an error page for a remote portlet using Oracle Enterprise Pack for Eclipse:

1. In Oracle Enterprise Pack for Eclipse, display the Outline view for the remote portlet. To do this, select **Window** > **Show View** > **Other**. In the Show View dialog, select **Basic** > **Outline**.

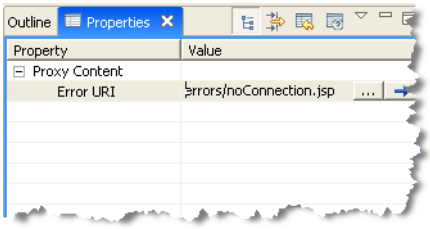
2. In the Outline View, click Proxy Content, as shown in Figure 5–4.

Figure 5–4 Selecting the Proxy Content Node



3. Click the Properties tab to display the Properties view for the Proxy Content. This view contains one property, Error URI, as shown in Figure 5–5.

Figure 5–5 Entering the Error Filename



4. In the Error URI field, enter (or browse to) the name of the error file you want to associate with the portlet. The portlet displays this page in the event of an error.

   The Error URI specifies a file path that is relative to the project in which the remote portlet is located.

5.3.2 Configuring an Error Page in the .portlet File

You can also configure an Error URI in a remote portlet's `.portlet` file. To do this, open the `.portlet` file and add the following element, where the value of the `errorUri` attribute is the name of the error file to be displayed:

```
<netuix:proxyPortletContent errorUri="errorFileName.jsp"/>
```

The `errorURI` attribute specifies a file path that is relative to the project in which the remote portlet is located.

Example 5–1 shows the complete XML file for a remote portlet, with an example `<netuix:proxyPortletContent>` element highlighted in bold.

Example 5–1 Remote Portlet XML File

```
<?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:proxyPortlet
        cacheExpires="300" definitionLabel="portlet_5_1" description=""
        doesUrlTemplateProcessing="true" forkRender="false"
        forkable="false" groupId="Consumer" portletHandle="portlet_5"
```

```
        producerHandle="consumerProducer" renderCacheable="true"
        templatesStoredInSession="true" title="Remote Preferences">
        <netuix:titlebar><netuix:maximize/><netuix:minimize/></netuix:titlebar>
        <netuix:proxyPortletContent errorUri="error.jsp"/>
    </netuix:proxyPortlet>
</portal:root>
```

5.4 Setting Preferences on a Remote Portlet

Portlet preferences function in remote portlets in much the same way as they do in local portlets. Just as with local portlets, remote portlets can take advantage of portlet preferences to allow users to customize the presentation of the portlet.

This section discusses the use of portlet preferences in remote portlets and includes these topics:

- Section 5.4.1, "What is a Portlet Preference?"
- Section 5.4.2, "Portlet Preferences and Remote Portlets"
- Section 5.4.3, "Managing Portlet Instances through Registration"

> **Note:** This section assumes that you are familiar with the concept of a portlet preference and how to create and configure portlet preferences. If you are unfamiliar with portlet preferences, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

5.4.1 What is a Portlet Preference?

Portlet preferences allow portlets to modify, store, and access pre-defined String values. When these preference values are retrieved by a portlet, they typically affect the way the portlet is displayed for a given user. For example, a stock portfolio portlet might allow users to specify which stocks they want to view. Through a user interface, users select or enter which stocks they want to view in the portlet. The list of stocks is then passed to the server and stored in the database for that particular user. As long as a portlet preference is modifiable, and an interface is provided for editing preferences, every user of a portlet can configure his or her own personal view of the portlet.

A clearly defined API exists for setting and retrieving preferences. Developers can create preferences in Oracle Enterprise Pack for Eclipse, and administrators can create and edit preferences using the WebLogic Portal Administration Console.

5.4.2 Portlet Preferences and Remote Portlets

In a federated configuration, the producer stores and manages portlet preferences. When you view or modify the preferences in a remote portlet (on a consumer), the consumer must fetch the preferences from the producer, and modifications must be sent back to the producer where they are stored.

> **Note:** Portlet preferences are included in the WebLogic Portal implementation of WSRP producers. Other WSRP producer implementations may not support portlet preferences.

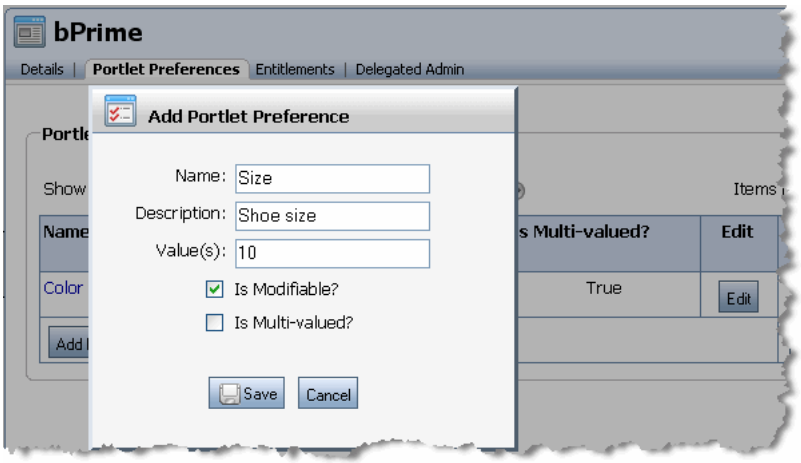
5.4.2.1 Viewing and Modifying Preferences

You can view and modify the portlet preferences for a remote portlet using the WebLogic Portal Administration Console. The Administration Console uses the Portlet Management interface of WSRP to retrieve preferences from the producer and modify them.

Note: It is not possible to create or modify portlet preferences in a remote portlet using Oracle Enterprise Pack for Eclipse.

Figure 5–6 shows the interface for creating a portlet preference in the WebLogic Portal Administration Console. A similar interface exists for editing a preference. For instance, you can change the default value for a preference, or make it read-only.

Figure 5–6 Creating a Portlet Preference in the WebLogic Portal Administration Console



Tip: Changes you make to a portlet preference in the Administration Console are scoped either at the Library level or the instance level. If you modify a portlet preference in the Library, all subsequent instances of that portlet will include the change. If you modify an instance (in the Portals folder) only that instance is affected. In other words, if the same portlet is used in several desktops, a new instance of the portlet is generated for each use. When you modify an instance of a portlet, only that instance is modified. Note that the first time a user updates a portlet preference, a new instance of the portlet is created, and the updated preferences are associated with the new instance. The WSRP registration interface provides a way for producers to keep track of new portlet instances created for remote portlets. See Section 5.4.3, "Managing Portlet Instances through Registration" for more information.

5.4.2.2 Working with Preferences Programatically

Portlets can also create, retrieve, and modify preferences programatically by obtaining a `javax.portlet.PortletPreferences` object. For instance, a page flow portlet can retrieve an instance of this object from the `PortletBackingContext` object in an action method. For example, the page flow action method shown in Example 5–2 retrieves from a `FormData` object a preference set by a user, sets the preferences in a `PortletPreferences` object, and stores the preferences in the database using the `store()` method.

> **Note:** Page flows are a feature of Apache Beehive, which is an optional framework that you can integrate with WLP. See "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

Example 5–2 Setting Portlet Preferences in an Action Method

```
/**
 * @jpf:action
 * @jpf:forward name="success" path="index.jsp"
 */
 protected Forward setColor(ColorForm form) {

 //-- Retrieve a preferences object from the context.
    PortletBackingContext context =
    PortletBackingContext.getPortletBackingContext(getRequest());
    PortletPreferences prefs = context.getPreferences(getRequest());

 //-- Set the user's preference.
 try {
  prefs.setValue("color", (String)form.getColor()[0]);
 } catch (ReadOnlyException e) {
      e.printStackTrace();
    }

    //-- Store the user's preference.
    try {
       prefs.store();
    } catch (ValidatorException io) {
       io.printStackTrace();
    } catch (IOException ioe) {
       ioe.printStackTrace();
    }

    return new Forward("success");
 }
```

As noted previously, for a remote portlet, preferences are hosted and managed on the producer. No preference information is ever stored on the consumer.

5.4.2.3 Additional Usage Notes and Restrictions

This section lists additional information about using portlet preferences in remote portlets.

- You cannot add portlet preferences to remote portlets consumed from a simple producer or from producers that have portal management disabled in the `wsrp-producer-config.xml file`.
- Portlets are not allowed to make persistent state changes during rendering. The `store()` method in `javax.portlet.PortletPreferences` throws an `IllegalStateException` if a portlet calls the `store()` method during the render phase of a portlet (that is, during the execution of the `getMarkup` operation).
- Portlets, whether remote or not, cannot be customized in any way, including the modification of portlet preferences, in either of these two cases: (a) the portlet is in a file-based portal (that is, rendered from a `.portal` file or (b) the user accessing the portlet is anonymous (not authenticated). Consumer portlets communicate this

to the producer by sending a value of `readOnly` for the `portletStateChange` element in the `performBlockingInteraction` request.

- If the instance of a remote portlet is shared among several users, WebLogic Portal consumer sends a value of `cloneBeforeWrite` for the `portletStateChange` element. This value indicates to the producer that it must clone the portlet before making changes to preferences. If a portlet does indeed modify preferences, the producer returns a new `portletHandle` to the consumer. This new `portletHandle` replaces the original `portletHandle`.
- On subsequent requests, the consumer sends a value of `readWrite` indicating that the producer can allow portlets to modify preferences.

5.4.3 Managing Portlet Instances through Registration

As discussed previously, whenever a user customizes a portlet by modifying portlet preferences, a new instance of the portlet is created. In the case of a remote portlet, the new instance is created on the producer, and the handle for that instance is returned to the consumer. Of course, as the number of users increases, the number of unique portlet instances can grow large in the producer space. If the consumer decides not to use the producer anymore, the producer needs to have a way of learning this and subsequently removing the portlet instances that are no longer needed. Portlet registration accomplishes this goal.

WebLogic Portal producers support registration by default for complex producers. If registration is enabled, consumers must register with a producer before accessing any of the producer's portlets. Once registered, the producer returns a `registrationHandle` to the consumer. The consumer must supply this handle on all future requests until the consumer is deregistered. When a consumer deregisters a portlet, the producer removes all of the portlet instances that were created for that consumer.

5.5 Using Backing Files with Remote Portlets

Backing files let you programatically add functionality to a portlet by implementing (or extending) a Java class, which enables preprocessing (for example, authentication) prior to rendering the portal controls. You can attach a backing file to a portlet using the Backing File property in the Properties View in Oracle Enterprise Pack for Eclipse.

Backing files let you implement business logic at certain points of a portlet's lifecycle. In a local portlet, backing file methods are called in the following order:

- `init()`
- `handlePostbackData()`
- `preRender()`
- `dispose()`

A producer, however, executes backing file methods in an order that reflects the type of consumer request, as shown in Table 5–4.

Table 5–3 Order of Backing File Method Execution in a Producer

Consumer Request	Order of Backing File Methods Called on the Producer
`getMarkup()`	`init()`, `preRender()`,`dispose()`
`performBlockingAction()`	`init()`,`handlePostbackData()`,`dispose()`
`handleEvents()`	`init()`, any event handler method, `dispose()`

For detailed information about backing files, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*. For an example that uses backing files with remote portlets, see Chapter 12, "Transferring Custom Data." See also Section 3.5, "Life Cycle of a Remote Portlet".

5.6 Setting a Timeout Value on a Remote Portlet

Occasionally, a producer is slow to respond to a request from a remote portlet. In this case, the portal application in which the remote portlet is located remains unresponsive until the remote portlet's response is received. This section explains how to set timeout values for remote portlets.

This section includes these topics:

- Section 5.6.1, "Overview"
- Section 5.6.2, "Setting Default Timeout Values"
- Section 5.6.3, "Setting Timeouts for Individual Remote Portlets"

5.6.1 Overview

WebLogic Portal provides two timeout settings for remote portlets:

- **Connection Establishment Timeout** – The amount of time a remote portlet will wait for a connection response from a producer.
- **Connection Timeout** – The amount of time the remote portlet will wait for a response from a producer to which it is already connected.

You can set a default timeout limit for all remote portlets and a timeout limit for an individual remote portlet. The timeout set on an individual portlet takes precedence over the default.

The remote portlet connection timeout only works when a consumer is continually connected to a producer. The timeout is effective only for cases where the producer is slow to respond to a consumer, not for cases where the producer is physically unavailable (the connection is broken), or where a new connection is made. In these cases, the operating system's TCP timeout takes effect.

5.6.2 Setting Default Timeout Values

To set default timeout values for all remote portlets in a web application, edit one or both of the elements shown in Example 5–3. These elements appear in the configuration file `wsrp-producer-registry.xml` located in the `WEB-INF` directory of each portal web application.

Example 5–3 Connection Timeout Elements

```
<connection-establishment-timeout-msecs>-1</connection-establishment-timeout-msecs>
<connection-timeout-msecs>120000</connection-timeout-msecs>
```

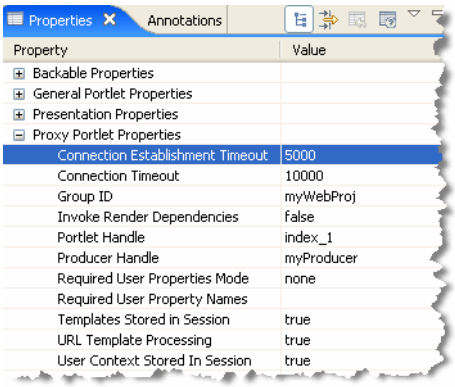
Note: Timeout values are in milliseconds.

5.6.3 Setting Timeouts for Individual Remote Portlets

To set a connection establishment and/or a connection timeout for an individual remote portlet, open the Properties view for the portlet in Oracle Enterprise Pack for

Eclipse and set values for the Connection Establishment Timeout and Connection Timeout properties, as shown in Figure 5–7. The timeout values are in milliseconds.

Figure 5–7 Setting Timeout Properties



5.7 Modifying WSRP Markup and Messages

The Interceptor Framework is a consumer-side framework that lets you programatically intercept and modify markup and user interaction-related WSRP messages sent to and received from producers. The framework exposes a set of interfaces that you can implement. These interfaces let you examine the content of a WSRP message and take specific action based on that content. For example, if a producer sends a registration error back to the consumer, an interceptor can detect that error and display an informative message to the user or, perhaps, automatically return the information required to complete the registration.

For more information on creating interceptors, see Chapter 9, "The Interceptor Framework."

5.8 Remote Portlet Properties

This section lists and describes the set of Proxy Portlet Properties and other portlet properties that of interest to federated portal developers. This section includes these topics:

- Section 5.8.1, "Proxy Portlet Properties"
- Section 5.8.2, "Other Portlet Properties"

5.8.1 Proxy Portlet Properties

Table 5–4 lists the Proxy Portlet Properties. These properties appear in the Properties list for remote (proxy) portlets.

Table 5–4 Proxy Portlet Properties

Property	Value
Connection Establishment Timeout	Optional. The number of milliseconds after which this portlet will time out when establishing an initial connection with its producer.
Connection Timeout	Optional. The number of milliseconds after which this portlet will time out when communicating with its producer. If not specified here, the default value contained in the file `WEB-INF/wsrp-producer-registry.xml` is used.

Table 5–4 (Cont.) Proxy Portlet Properties

Property	Value
Group ID	Read-only (assigned by the producer). If the producer associates this portlet within a group, the producer-assigned string appears here. Portlets with the same group ID from the same producer can share sessions.
Invoke Render Dependencies	Read-only (assigned by the producer). This boolean property allows the consumer to obtain render dependencies from the producer during the pre-render life cycle of a proxy portlet. When a portlet on a producer has a `lafDependenciesUri` value, the producer exposes the `invokeRenderDependencies` boolean in the portlet description. The value defaults to `false` if the attribute is not included in the `.portlet` file. The value is read-only, and is initialized from the producer whenever a proxy portlet is generated from the portlet wizard.
Portlet Handle	Read-only (assigned by the producer). The producer's unique identifier for the portlet that this proxy references.
Producer Handle	Required. The producer's unique identifier.
Required User Property Mode	Optional. Possible values are `none`, `all`, or `specified`. If the value is `specified`, then you must enter a list of property names in the field Required User Properties Names field.
Required User Property Names	Optional. Use this field if you entered a value of `specified` in the Required User Properties Mode field; enter a comma-delimited list of property names.
Templates Stored in Session	Read-only (assigned by the producer). Indicates whether the producer stores URL templates in the user's session on the producer side. This boolean is meaningful only when URL Template Processing boolean is set to `true`.
URL Template Processing	Read-only (assigned by the producer). 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.
User Context Stored In Session	Read-only (assigned by the producer). This boolean value defaults to `false` if the attribute is not included in the `.portlet` file. This value is initialized from the producer whenever a proxy portlet is generated from the portlet wizard.

5.8.2 Other Portlet Properties

Remote portlets also include properties that are common to other types of portlets. For a complete list and descriptions of all portlet properties, see "Portlet Properties" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

6

Offering Books, Pages, and Portlets to Consumers

WebLogic Portal producer applications can offer books, pages, and portlets to consumers. This chapter explains the procedures and best practices involved in making books, pages, and portlets remoteable.

> **Tip:** In this chapter, we use the term remoteable to refer to a book, page, or portlet that is deployed in a producer application. To be remoteable, the Offer As Remote property of the book, page, or portlet must be set to true, as explained in later in this chapter.

This chapter includes these sections:

6.1 Introduction

A complex producer can offer remoteable books, pages, and portlets. When a page or book is offered as remote from a complex producer application, the nested contents of the page or book are, by default, also offered as remote. This means that you can group multiple portlets in a page, for example, and a WebLogic Portal consumer can then consume both the page and its portlets in one operation.

> **Tip:** Portlets deployed in a simple application can also be remoteable; however, only complex producers can offer remoteable books and pages. See Chapter 8, "Configuring a WebLogic Server Producer" for more information on creating remoteable portlets in a WebLogic Server application. For information on simple and complex producers, see Section 3.4, "Understanding Producers and Consumers".

Table 6–5 summarizes which Oracle tools you can use to create and consume remote books, pages, and portlets. Although you can consume remote portlets using Oracle Enterprise Pack for Eclipse, you cannot consume remote books and pages. Oracle Enterprise Pack for Eclipse does not provide a feature for locating and consuming remote books and pages. If you want to incorporate remote books and pages into a

WebLogic Portal consumer application, you must use the WebLogic Portal Administration Console, see Chapter 18, "Adding Remote Resources to the Library."

Table 6–1 List of Oracle Tools for Creating and Consuming Remote Resources

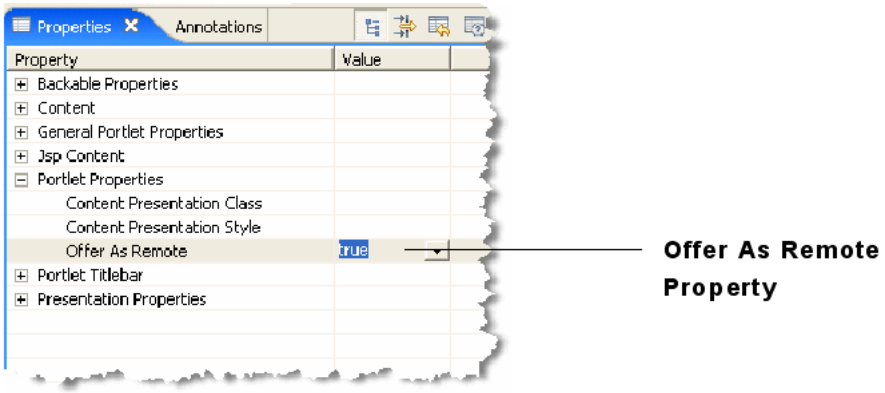
Feature	Oracle Enterprise Pack for Eclipse	Administration Console
Create remoteable books and pages	Yes	No
Create remoteable portlets	Yes	No
Consume remote portlets	Yes	Yes
Consume remote books and pages	No	Yes

6.2 Offering Portlets on a Producer

By default, all portlets deployed in a WebLogic Portal producer application are available to consumers as remote portlets. You can, however, specify which portlets are actually available to consumers by setting the Offer As Remote property in the Properties view for the portlet, as shown in Figure 6–6.

If you want a portlet to be available to consumers, set Offer As Remote to `true` (the default). If you want to hide a portlet from consumers, set Offer As Remote to `false`.

Figure 6–1 Portlet Properties View



For detailed information on creating portlets and setting properties, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

6.3 Offering Books and Pages on a Producer

If you want to create books and pages that are accessible to remote consumer applications, you must use Oracle Enterprise Pack for Eclipse.

To make a remoteable book or page in Oracle Enterprise Pack for Eclipse, as the following procedures explain, you must create the book or page as a standalone `.book` or `.page` file. In Oracle Enterprise Pack for Eclipse, you can do this by selecting **New** > **File** > **Other** > **WebLogic Portal** > **Book** (or **Page**).

> **Tip:** For more information on creating and working with pages and books, see the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

This section includes these topics:

- Section 6.3.1, "Setting Up the Example"
- Section 6.3.2, "Creating a Remoteable Page (or Book)"
- Section 6.3.3, "Summary"

6.3.1 Setting Up the Example

If you want to try the example discussed in this section, you need to run Oracle Enterprise Pack for Eclipse and perform the prerequisite tasks outlined in this section.

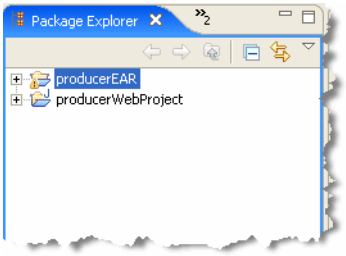
To set up the example environment, perform the prerequisite tasks outlined in Table 6–5. If you are not familiar with the specific procedures for these tasks, they are described in detail in the *Oracle Fusion Middleware Tutorials for Oracle WebLogic Portal*.

Table 6–2 Prerequisite Tasks

Task	Recommended Name
Create a WebLogic Portal domain.	`producerPortalDomain`
Create a Portal EAR Project.	`producerEAR`
Create an Oracle WebLogic Server v10.x.	N/A
Associate the EAR project with the server.	N/A
Create a Portal Web Project and add it to the EAR.	`producerWebProject`

Figure 6–2 shows the Package Explorer after the prerequisite tasks have been completed.

Figure 6–2 Package Explorer After Prerequisite Tasks are Completed



6.3.2 Creating a Remoteable Page (or Book)

> **Tip:** The procedure for creating a remoteable book is almost identical to the procedure for creating a page. Rather than reproduce both procedures here, we explain how to create a remoteable page and, where appropriate, highlight any differences between the two procedures.

To create a page in a producer application that is accessible to consumer applications:

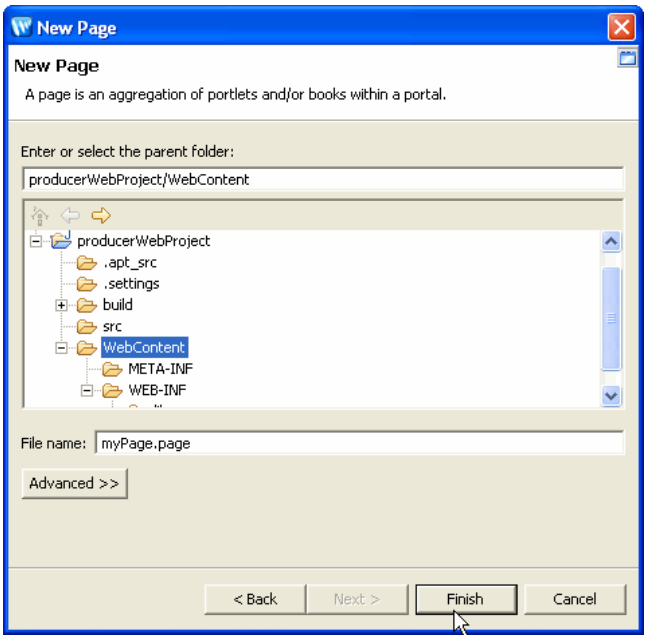
1. Start Oracle Enterprise Pack for Eclipse.
2. Create a Portal Web Project, as explained in the previous section.
3. Select **File** > **New** > **Other**.

4. In the New – Select a wizard dialog, open the WebLogic Portal folder, select **Page**, and click **Next**.

   Tip: To create a remoteable book, select **Book** instead of Page.

5. In the New Page dialog, select a parent folder for the new page and enter a name for the page, as shown in Figure 6–3. In this example, the parent folder is `WebContent`, and the filename is `myPage.page`.

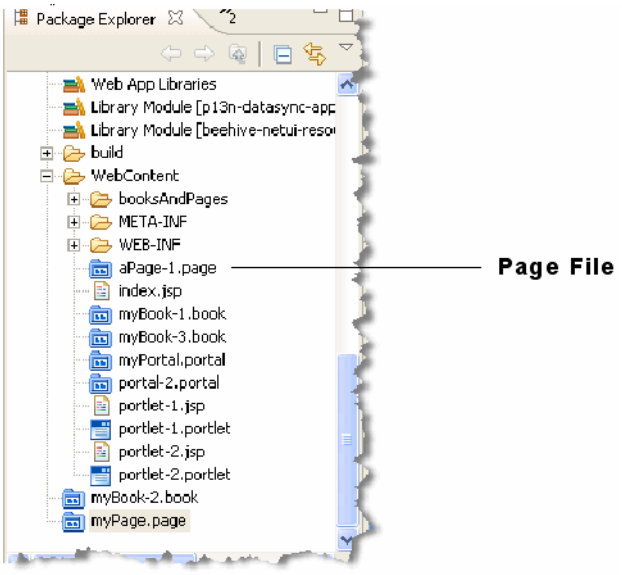
Figure 6–3 New Page Dialog



6. Click **Finish**.

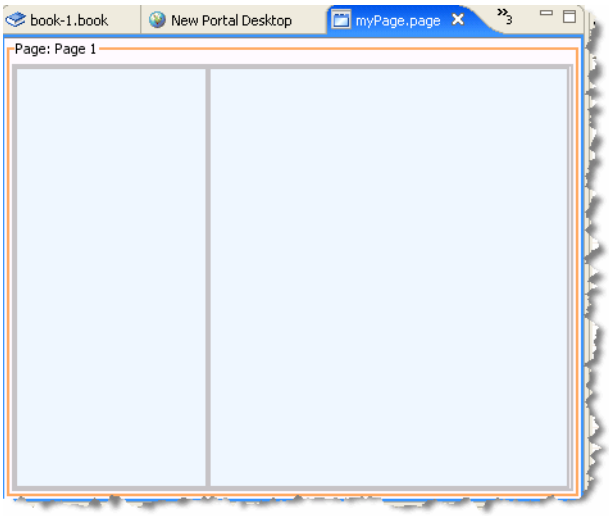
   Checkpoint: The file `myPage.page` is added to the Portal Web Project in the folder you specified, as shown in Figure 6–4.

Figure 6–4 A New Page File



In addition, the page opens in the editor, as shown in Figure 6–5.

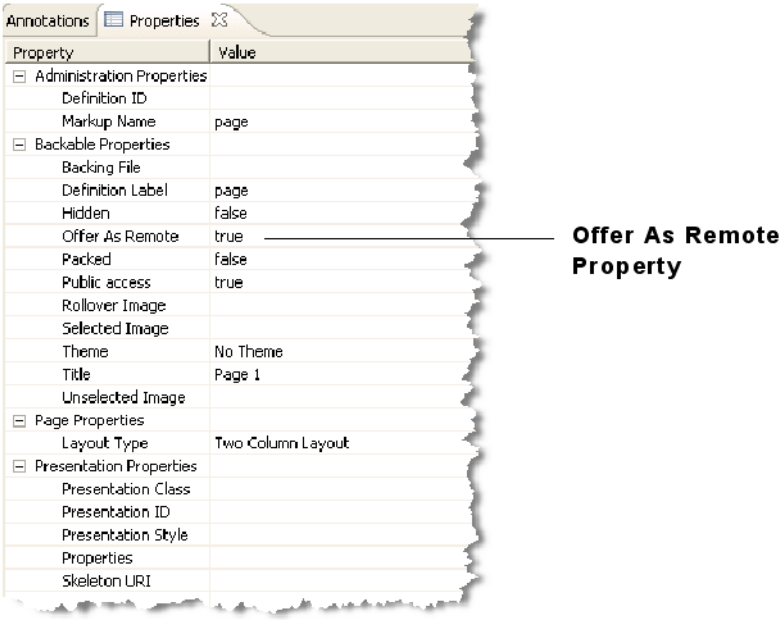
Figure 6–5 Page File Displayed in the Editor



7. Click on the border of the page to display the page's Properties view. If the Properties view is not currently available, select **Window** > **Show View** > **Properties**.

8. Note that, by default, the Offer As Remote property is set to `true` for this `.page` file, as shown in Figure 6–6. This property setting means that this page, and any books, pages, and portlets you add to it (according to the rules discussed in Section 6.4, "Rules for Creating Remoteable Books and Pages") will be visible to consumers if their respective Offer As Remote properties are also set to `true`.

Figure 6–6 Offer As Remote Property



6.3.3 Summary

You can treat the page shown in Figure 6–5 like any other page. You can add books and portlets to it and you can drag and drop the page into a portal. If you create a remote book, you can add pages to it, and those pages can in turn contain portlets and other books.

6.4 Rules for Creating Remoteable Books and Pages

The key points to remember with respect to making a page (or book) accessible to remote consumers are:

- If you have a book or page that is offered as remote, but none of the book's or page's contents (other books, pages, and portlets) are offered as remote, the book or page will not be visible to consumers. To be visible, a book or page must be offered as remote and must contain at least one other entity that is offered as remote.

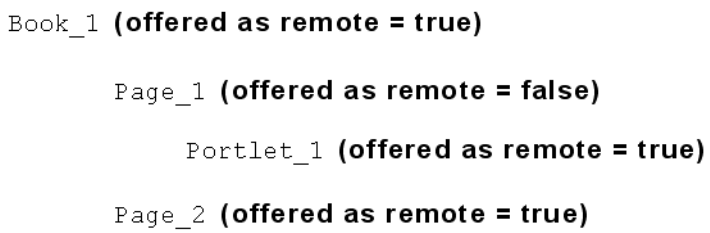
  For example, Figure 6–7 shows a sample configuration. In this configuration, consumers can locate `Book_1`. To a consumer, `Book_1` contains one page, `Page_2`. Because `Page_1` is not offered as remote, it will not be visible to consumers, nor will any of its contents.

Figure 6–7 Sample Configuration

```
Book_1 (offered as remote = true)

    Page_1 (offered as remote = false)

        Portlet_1 (offered as remote = true)

    Page_2 (offered as remote = true)

        Portlet_2 (offered as remote = true)
```

Figure 6–8 shows another sample configuration. In this case, `Book_1` is offered as remote; however, it is not visible to consumers. This is because none of its contents are offered as remote. `Page_1` is not offered as remote explicitly and `Page_2` is not offered as remote because it is empty (even though its property is set to `true`).

Figure 6–8 Sample Configuration



- Remoteable books and pages must be created as standalone `.book` and `.page` files as explained previously in Section 6.3.2, "Creating a Remoteable Page (or Book)".
- Changes to remoteable pages and books made on the producer cannot be propagated to consumers of those pages and books. This means that if you change a remoteable page or book in a producer application, and that page or book has already been consumed by consumer applications, the changes will not show up in the consumers.
- Portal look and feel elements that are used in `.page` and `.book` files must be replicated on the consumer. This means that look and feel files, such as `.layout`, `.theme`, and supporting JSP files that are used in a remoteable book or page must exist on both the producer and the consumer.
- A backing file placed on a remoteable `.book` or `.page` file in a producer application has no effect when the book or page is consumed.

7

Interportlet Communication with Remote Portlets

WebLogic Portal supports interportlet communication (IPC) between producers and consumers. For example, a remote portlet deployed in a producer application can handle a minimize event fired by a local portlet in a consumer. This chapter presents a detailed example explaining how to use interportlet communication with remote portlets.

This chapter includes these sections:

7.1 Introduction

WebLogic Portal provides an extension to the WSRP protocol that allows remote portlets to fire events during the interaction phase of their lifecycle. For detailed information on the WebLogic Portal IPC architecture for federated portlets, see Section 3.5.4, "Interportlet Communication with Events".

Communication between portlets deployed in consumer and producer applications is bi-directional. Events fired by local portlets can be handled by portlets deployed in a producer, and vice versa.

The example in this chapter demonstrates one way to implement event handling in a federated portal. In this example, the event handler is added to the portlet on the producer. When a local portlet on the consumer fires an event, the remote portlet on the producer receives the event and handles it (changes the text displayed in the portlet).

> **Note:** Whenever you implement event handling in a federated environment, remember that you must add event handlers to portlets in the producer application before you create proxy portlets in consumers. If you change a producer portlet's metadata, such as by adding an event handler, consumers are not notified of that change. The correct procedure is to add the event handler to the portlet on the producer before you create the remote portlet on the consumer.

For additional information on IPC in WebLogic Portal, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

7.2 Firing and Handling a Minimize Event

This section presents a detailed example demonstrating how to use event handling in a remote portlet. In this example, a remote portlet on the consumer fires an onMinimize event. The onMinimize event is fired when a portlet is minimized. When the event is fired from a local portlet in a consumer, the event is handled on the producer. The onMinimize event is one of several standard events supported by the WebLogic Portal framework. For a complete list of standard events, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

In this example, when the user minimizes the remote portlet on the consumer, the producer handles the event and changes some text in the portlet.

> **Tip:** Remote portlets can also handle custom events. For detailed information on using custom events with remote portlets, see Section 7.4, "Data Transfer with Custom Events".

This example includes these steps:

1. Section 7.2.1, "Setting Up Your Environment"
2. Section 7.2.2, "Creating the Portlets on the Producer"
3. Section 7.2.3, "Creating the Consumer Portlets"
4. Section 7.2.4, "Testing the Application"

7.2.1 Setting Up Your Environment

If you want to try the example discussed in this section, you need to run Oracle Enterprise Pack for Eclipse and perform the prerequisite tasks and set up the example environment.

To set up the example environment, perform the prerequisite tasks outlined in Table 7–1. If you are not familiar with the specific procedures for these tasks, they are described in detail in the WebLogic Portal tutorial *Oracle Fusion Middleware Tutorials for Oracle WebLogic Portal*.

Table 7–1 Prerequisite Tasks

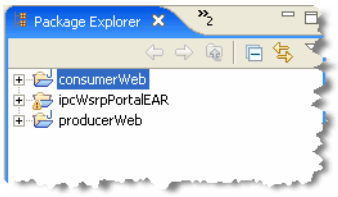
Task	Recommended Name
1. Create a WebLogic Portal domain.	`ipcWsrpDomain`
2. Create a Portal EAR Project.	`ipcWsrpPortalEAR`
3. Create an Oracle WebLogic Server v10.x.	N/A

Table 7–1 (Cont.) Prerequisite Tasks

Task	Recommended Name
3. Associate the EAR project with the server.	N/A
4. Create a Portal Web Project	`consumerWeb`
5. Create a second Portal Web Project	`producerWeb`

Figure 7–1 shows the Package Explorer after the prerequisite tasks have been completed.

Figure 7–1 Package Explorer After Prerequisite Tasks are Completed



7.2.2 Creating the Portlets on the Producer

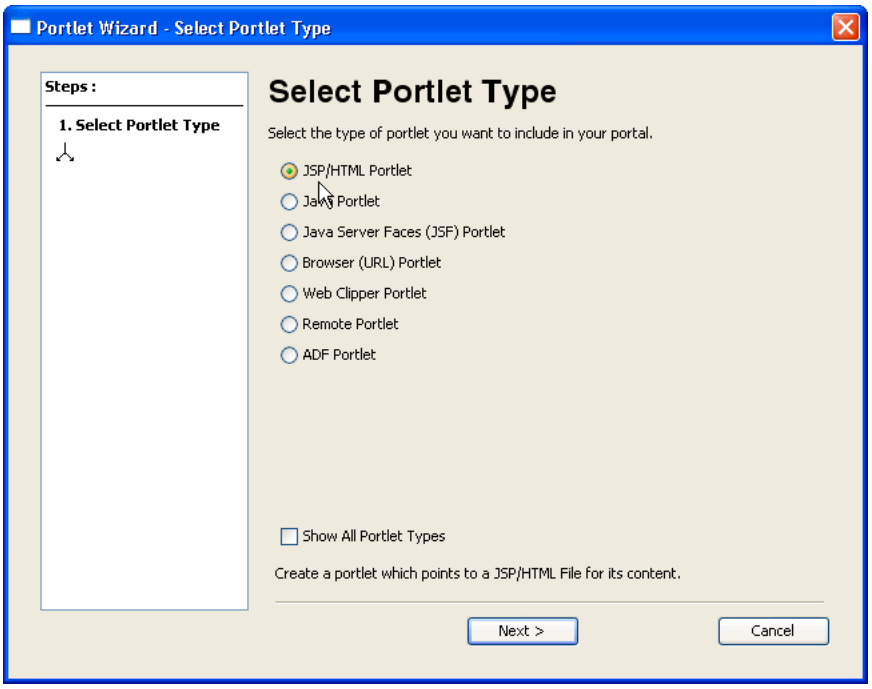
In this task, you create two JSP files on the producer-side, along with the JSP portlets that surface these files. You also create a backing file that contains the instructions necessary to complete the communication between two portlets and add an event handler to one of the portlets. Once you have created the portlets and attached the backing file, you will test the application in your browser.

7.2.2.1 Create the JSP Files and Portlets

To create the JSP files that the portlets deployed on the producer will surface:

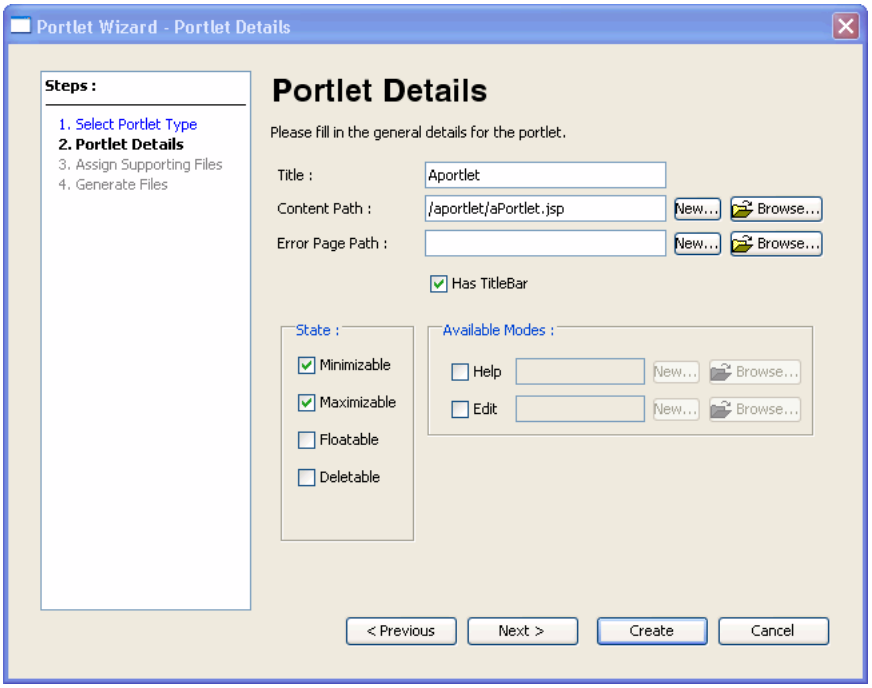
1. Be sure you have set up the example environment as explained previously in Section 7.2.1, "Setting Up Your Environment".
2. In the Project View, right-click the WebContent folder and select **New > Portlet**.
3. In the New Portlet dialog, enter the name `aPortlet.portlet` for the new portlet, and click **Next**.
4. In the Select Portlet Type wizard page, select **JSP/HTML Portlet**, and **click Next**. (See Figure 7–2.)

Figure 7–2 Select Portlet Type



5. In the Portlet Details wizard page, select the **Minimizable** and **Maximizable** states, and click **Create**. (See Figure 7–3.)

Figure 7–3 Portlet Details



6. Locate the `aPortlet.jsp` file. By default, it will be in the `WebContent/aportlet` folder. Double-click the file to open it in the editor.
7. Replace the default text in the file with the text `"Minimize Me!"` as shown in Figure 7–4.

Figure 7–4 JSP File Showing Edited Body Text

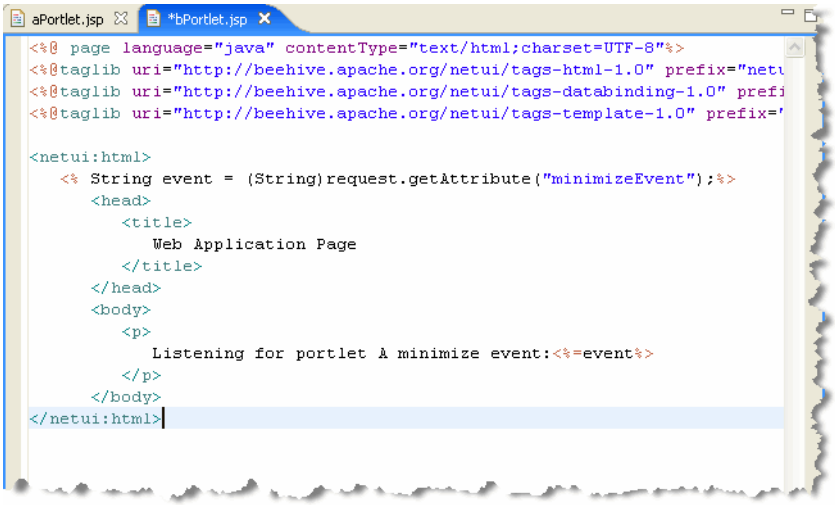
8. Save the file as `aPortlet.jsp`
9. In the same directory, make a copy of `aPortlet.jsp`, and call the copy `bPortlet.jsp`.
10. Open `bPortlet.jsp` in the editor and copy the code from Example 7–1 into the JSP, replacing everything from `<netui:html>` through `</netui:html>`. This code simply displays text placed in the request by a backing file, which you will create and attach to the portlet in a subsequent step.

Example 7–1 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>
```

Figure 7–5 shows the completed JSP source file in the editor.

Figure 7–5 Updated JSP Source



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

    Checkpoint: At this point the you have created the following files in the producerWeb/WebContent folder:

    - `aPortlet.jsp`
    - `aPortlet.portlet`
    - `bPortlet.jsp`
    - `bPortlet.portlet`.

7.2.2.2 Create the Backing File

In this example, we use a backing file attached to the portlet in the producer to handle the onMinimize event fired on the consumer.

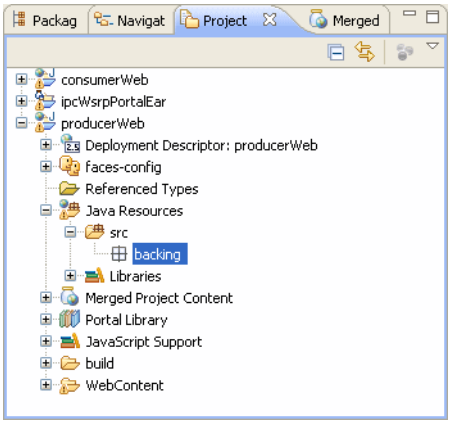
> **Tip:** For detailed information on backing files, refer to the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

To create the backing file:

1. In producerWeb, expand the Java Resources node, right-click the **src** folder, and select **New** > **Package** from the menu. The Create New Folder dialog box appears.
2. Create a source folder called `backing`.

   The `backing` folder appears under `producerWeb/Java Resources/src`, as shown in Figure 7–6.

Figure 7–6 New Backing File Package



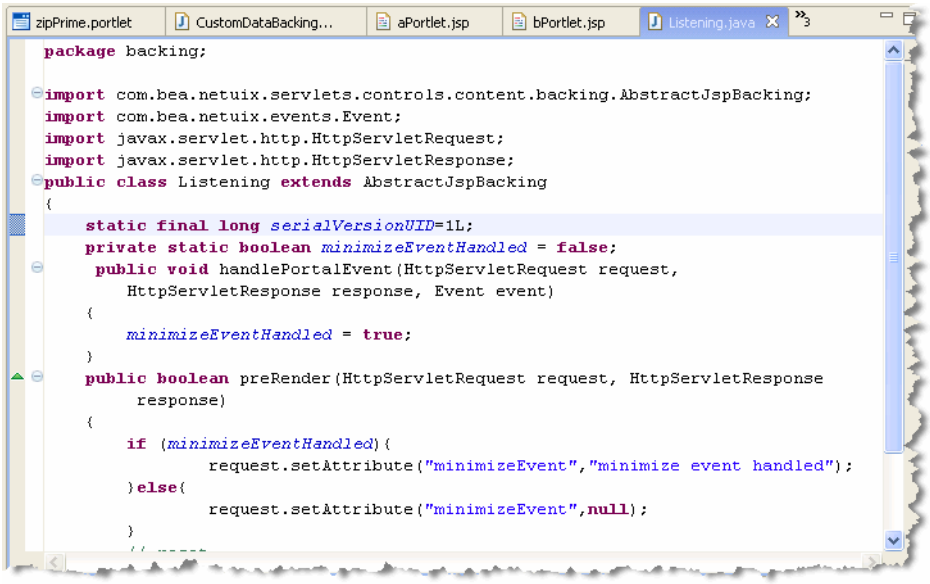
3. Right-click the **backing** package and select **New** > **Class**. The New Java Class dialog appears.
4. In the Name field, enter `Listening` and click **Finish**. The new Java class appears in the editor.
5. Delete the default contents of `Listening.java`, and copy the code from Example 7–2 into `Listening.java`.
6. Save `Listening.java`.

Example 7–2 Backing File Code for listening.java

```
package backing;
import com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking;
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;
     private static boolean minimizeEventHandled = false;
     public void handlePortalEvent(HttpServletRequest request,
        HttpServletResponse response, Event event)
    {
        minimizeEventHandled = true;
    }
    public boolean preRender(HttpServletRequest request, HttpServletResponse
         response)
    {
        if (minimizeEventHandled){
                request.setAttribute("minimizeEvent","minimize event handled");
        }else{
                request.setAttribute("minimizeEvent",null);
        }
        // reset
        minimizeEventHandled = false;
         return true;
    }
}
```

The source should now look like that shown in Figure 7–7.

Figure 7–7 Listening.java with Updated Backing File Code

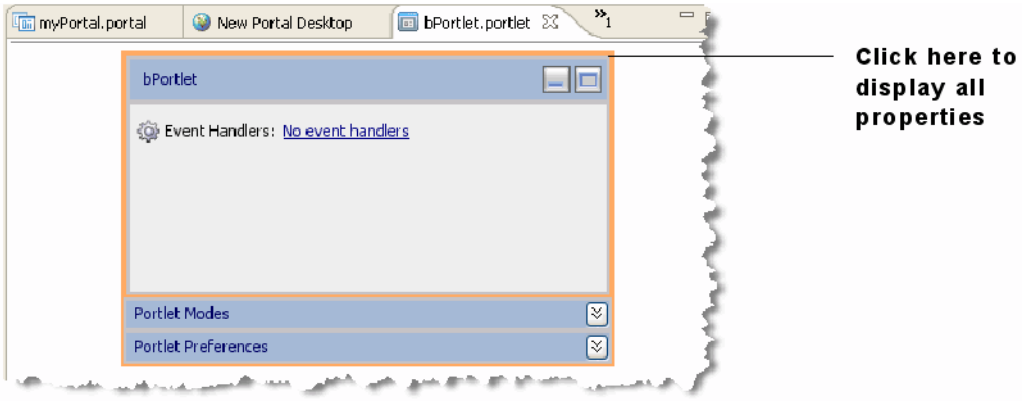


7.2.2.3 Attach the Backing File

Now you will attach the backing file created in the previous section to `bPortlet.portlet`.

1. In the Package Explorer, double-click `bPortlet.portlet` to open it.
2. Click on the portlet in the editor to display the portlet's properties. To be sure you see all the properties, click on the border of the portlet, as shown in Figure 7–8.

Figure 7–8 Click to Display All Portlet Properties

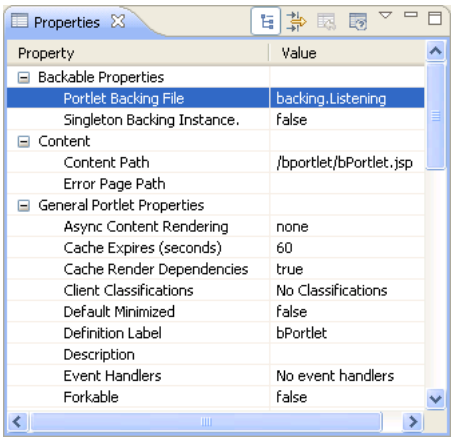


Tip: If the Properties view is not visible in your perspective, select **Window** > **Show View** > **Properties**. If you want to learn more about editing portlet properties, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

3. In the Properties view, type `backing.Listening` into the Backable Properties > Portlet Backing File field, as shown in Figure 7–9 and press Return.

Tip: You might need to expand the value column to enter text in the Portlet Backing File field.

Figure 7–9 Attaching the Backing File in the Properties View



4. Save the file.

7.2.2.4 Add the Event Handler to bPortlet

You now add the event handler to `bPortlet.portlet`. The handler will be configured to listen for an event fired by another portlet and execute an action in response to that event. To add the event handler:

1. Be sure the file bPortlet.portlet is open. If it is not, double-click it in the Package Explorer.
2. Click on the portlet in the editor to display the portlet's properties, as shown previously in Figure 7–8.
3. In the portlet editor, click the **Event Handlers** link, as shown in Figure 7–10. The Event Handlers dialog appears.

Figure 7–10 Event Handlers Link

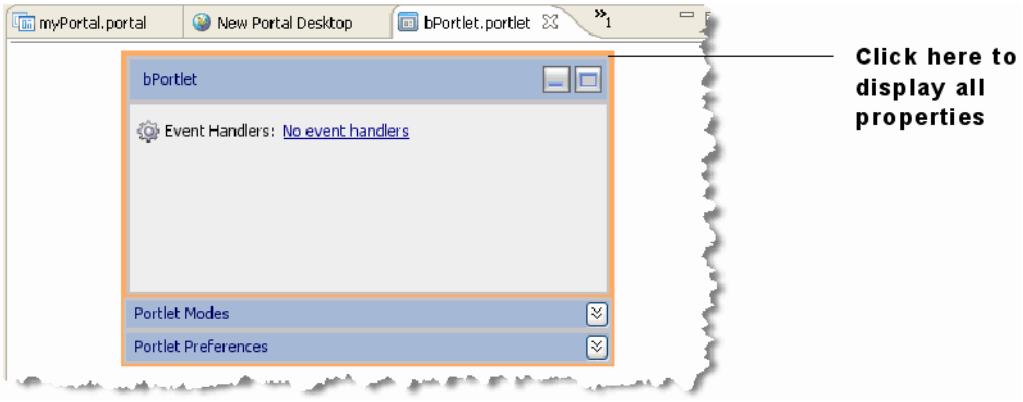
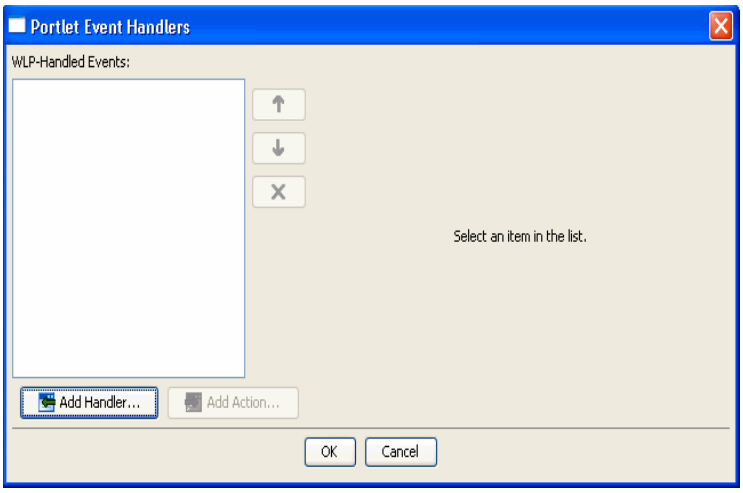


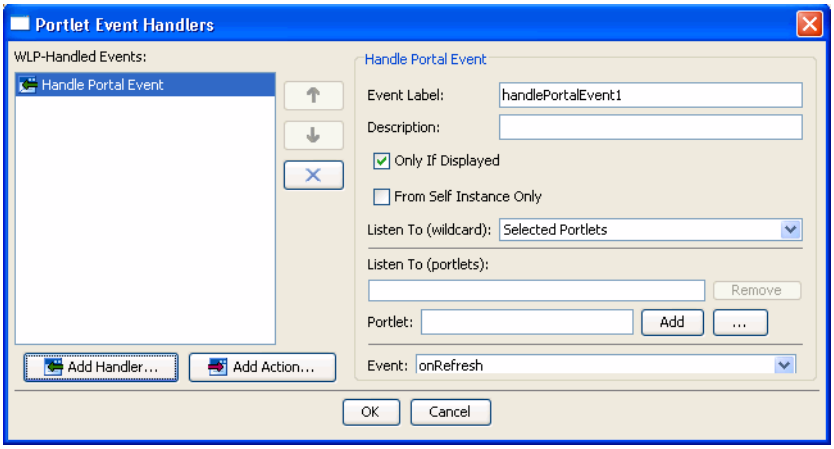
Figure 7–11 Portlet Event Handlers Dialog Box



4. Click **Add Handler** and select **Handle Portal Event** from the drop-down menu.

   The Portlet Event Handlers dialog box expands to allow entry of more details, as shown in Figure 7–12.

Figure 7–12 Event Handler Dialog Box Expanded

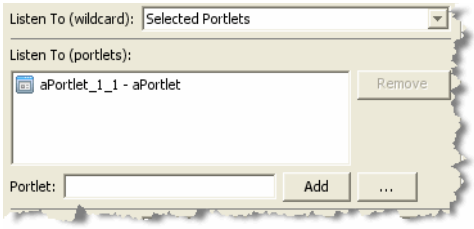


5. Accept the defaults for all fields except Portlet.
6. In the Portlet field, click the ellipses button (**...**). The Please Choose a File dialog box appears.
7. Select **aPortlet.portlet** and click **OK**.

   The dialog box closes and aPortlet_1 appears in the Listen to list and the Portlet field, as shown in Figure 7–13. The label aPortlet_1 is the definition label of the portlet to which the event handler will listen.

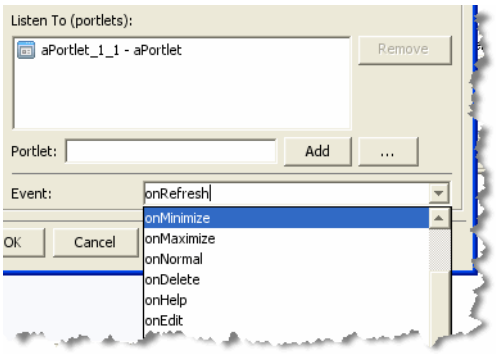
   Tip: The definition label is a unique identifier for the portlet. A default value is entered automatically, but you can change the value. Each portlet must have a unique value. See the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal* for more information.

Figure 7–13 Adding portlet_1



8. Click the **Event** drop-down menu to open the list of portal events that the handler can listen for and select **onMinimize**, as shown in Figure 7–14.

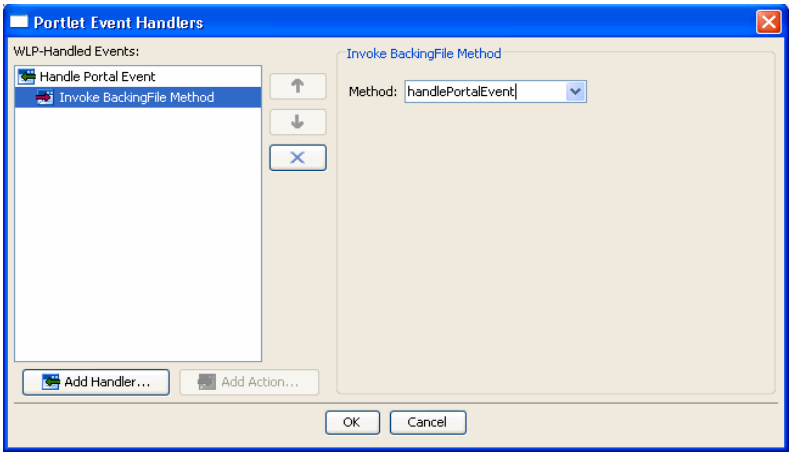
Figure 7–14 Event Drop-down List



9. Click **Add Action...** to open the action drop-down menu and select **Invoke BackingFile Method**.

10. Open the **Method** drop-down menu and enter `handlePortalEvent`, as shown in Figure 7–15. This method is defined in the backing file that is attached to bPortlet. The source code for the backing file was shown previously in Example 7–2.

Figure 7–15 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.

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 *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

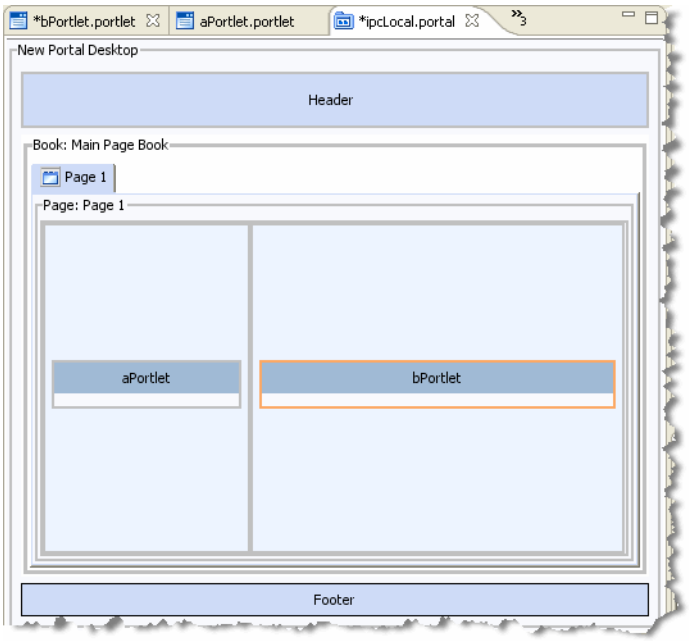
Checkpoint: You added a backing file and an event handler to bPortlet. The event handler is configured to invoke the handlePortalEvent() method in the backing file when the portlet receives an onMinimize event fired by aPortlet. In the next task, you test the application to make sure that the portlets function properly in a local environment. Then, you will create a remote portlet in a consumer application to test the interportlet communication in a federated portal environment.

7.2.2.5 Test the Application

Create a portal in the producer application called `ipcLocal.portal`:

1. In the Package Explorer, right-click **producerWeb/WebContent** and select **New** > **Portal**. The New Portal dialog appears.
2. In the File name field, enter `ipcLocal.portal` and click **Finish**. The portal is created and appears in the editor.
3. Drag both `aPortlet.portlet` and `bPortlet.portlet` from the Package Explorer onto the portal layout, as shown in Figure 7–16.

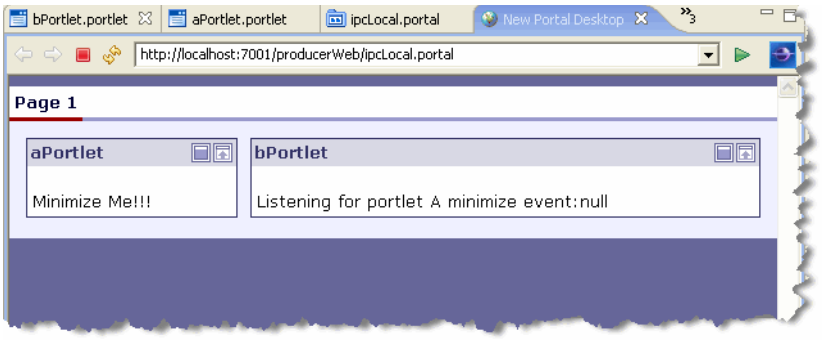
Figure 7–16 Portal Layout with Portlets Added



4. Save the portal.
5. Run the portal. To do this, right-click **ipcLocal.portal** in the Package Explorer and select **Run As** > **Run on Server**.
6. In the Run On Server – Define a New Server dialog, click **Finish**.

   The portal renders in the default browser, as shown in Figure 7–17.

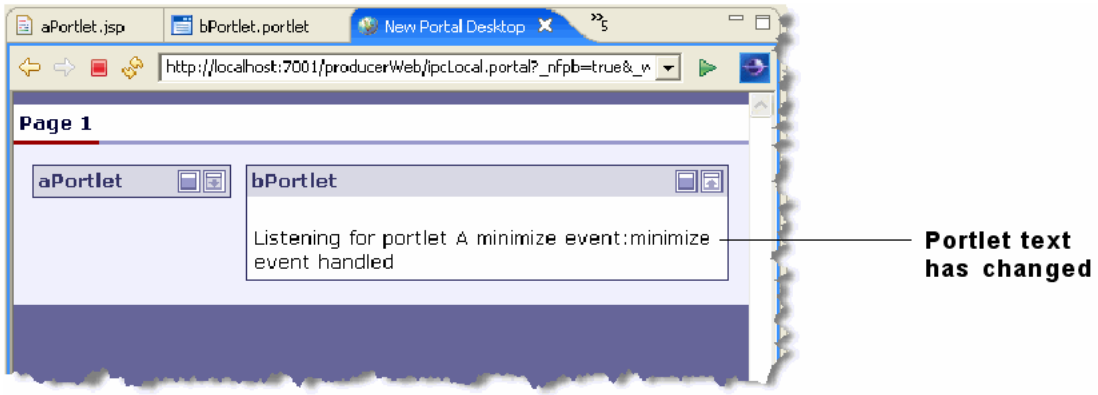
Figure 7–17 ipcLocal Portal in Browser



7. Minimize aPortlet.

   Note the content change in bPortlet.

Figure 7–18 ipcLocal Portal with aPortlet Minimized



7.2.2.6 Summary

You created a portal containing two local portlets. You configured the portlet called bPortlet to respond to an onMinimize event fired from the portlet called aPortlet. The onMinimize event is a standard event that all WebLogic Portal portlets can fire. When bPortlet receives an onMinimize event, a backing file method is called that modifies the text displayed by the portlet.

In the following steps, you will create a federated portal that uses interportlet communication.

7.2.3 Creating the Consumer Portlets

In this section, you create two portlets in the consumer application, one a JSP portlet and the other a remote portlet. The remote portlet consumes the portlet you created previously on the producer, `bPortlet.portlet`.

7.2.3.1 Setting Up the Exercise

Before you continue with this exercise:

1. In the Package Explorer, copy aPortlet.jsp from the producerWeb/WebContent folder and paste it into the consumerWeb/WebContent folder. For convenience, we reuse this portlet from the producer application. Its function in the consumer portal is simply to provide a portlet that you can minimize.

2. Right-click **consumerWeb/WebContent/aPortlet.jsp** and select **Generate Portlet**.
3. In the Portlet Details dialog, select **Minimizable**, **Maximizable**, and click **Create**. The new portlet layout appears in the editor.

7.2.3.2 Creating the Remote Portlet

To create the remote portlet:

1. Open the consumerWeb folder in the Package Explorer, right-click on the **WebContent** folder, and select **New** > **Portlet**.
2. In the New Portlet dialog, enter `bPrime.portlet` in the File name field, and click **Finish**.
3. In the Select Portlet Type dialog of the Portlet Wizard, pick **Remote Portlet**, and click **Next**.
4. In the Producer dialog, select **Find Producer**.
5. Enter the producer's WSDL URL in the text field, as shown in Figure 7–19. The WSDL URL for this example is:

   `http://host:port/producerWeb/producer?wsdl`

   for example:

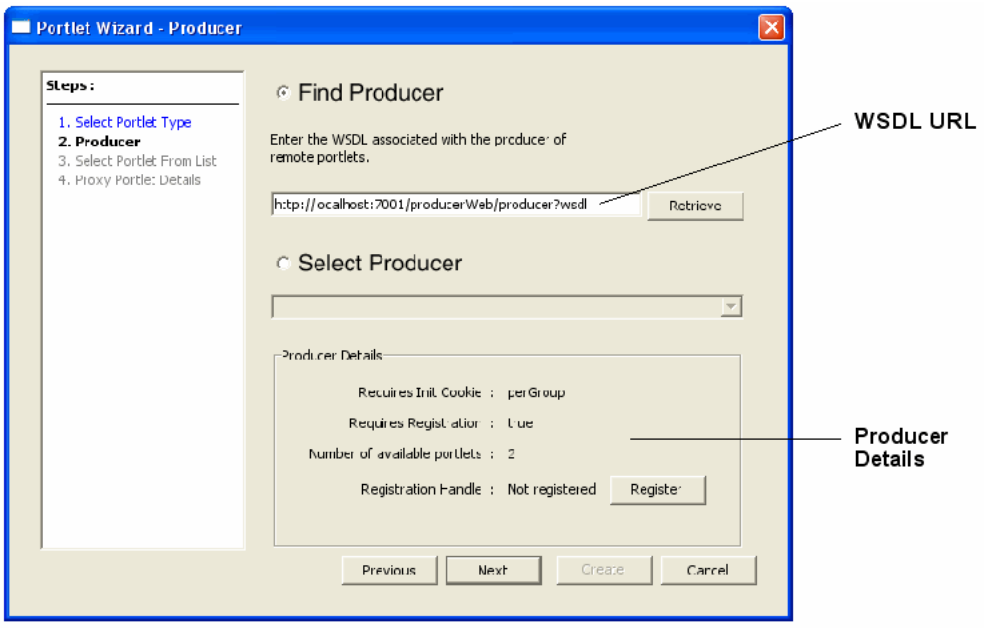
   `http://localhost:7001/producerWeb/producer?wsdl`

   > **Tip:** WSDL stands for Web Services Description Language and is used to describe the services offered by a producer. For more information, see Chapter 3, "Federated Portal Architecture."

6. Click **Retrieve**.

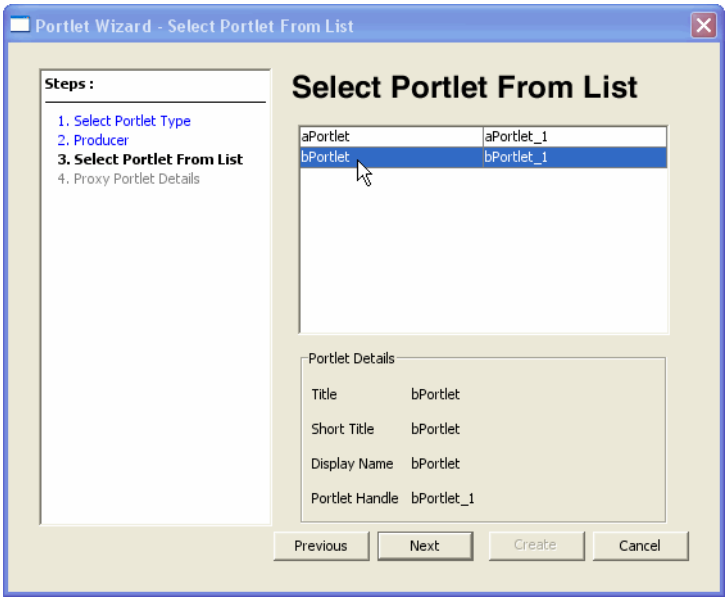
   After a few seconds, the dialog box refreshes, showing the Producer Details, as shown in Figure 7–19.

Figure 7–19 Find Producer Dialog



7. Click **Register**.
8. In the Register dialog, enter a name for the producer in the Producer Handle field, and click **Register**. You are returned to the Producer dialog.
9. In the Producer dialog, click **Next**. The Select Portlet from List dialog appears.
10. In the Select Portlet from List dialog, select bPortlet, as shown in Figure 7–20.

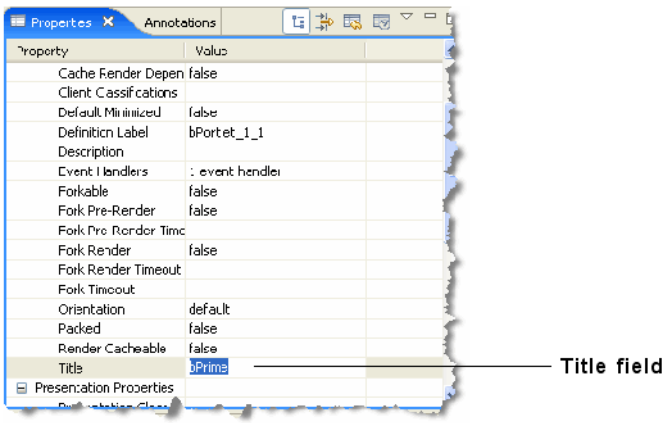
Figure 7–20 Select Portlet From List Dialog Box



11. Click **Next**. The Proxy Portlet Details dialog box appears.
12. Click **Create**.

    The remote portlet appears as `bPrime.portlet` in the consumerWeb/WebContent folder in the Package Explorer.
13. Change the portlet's title to `bPrime`. To do this, edit the Title field in the portlet's Properties view, as shown in Figure 7–21.

Figure 7–21 Changing the Portlet Title



14. Save the portlet.

Tip: In the Properties view for bPortlet (in the producerWeb/WebContent folder) be sure the Render Cacheable property is set to false.

7.2.3.3 Summary

With the completion of the two consumer portlets, you have now created all of the necessary components to demonstrate interportlet communications between a remote and a local portlet. In the next step, you will add the consumer portlets to a consumer portal and raise an event on one portlet that will cause a reaction on the other.

7.2.4 Testing the Application

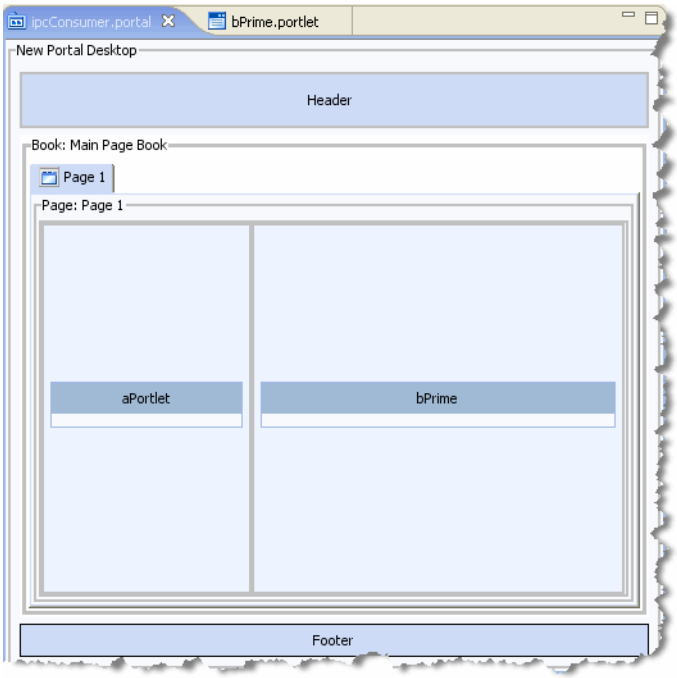
In this step, you test the consumer application to verify that minimizing aPortlet will change the content of bPrime (the remote portlet). You create a portal and add the two portlets created in Section 7.2.3, "Creating the Consumer Portlets". You then build the application and view the portal in a browser.

7.2.4.1 Build the Portal

Create a portal in the consumer application called `ipcConsumer.portal`:

1. In the Package Explorer, right-click **consumerWeb/WebContent** and select **New** > **Portal**. The New Portal dialog appears.
2. In the File name field, enter `ipcConsumer.portal` and click **Finish**. The portal is created and appears in the editor.
3. Drag both `aPortlet.portlet` and `bPrime.portlet` from the consumerWeb/WebContent folder onto the portal layout. The result is shown in Figure 7–22.

Figure 7–22 Consumer Portal Layout



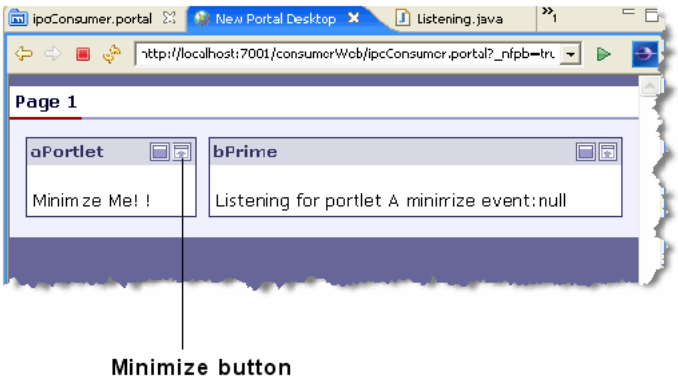
4. Save the portal.

7.2.4.2 Test the Portal

From a user's perspective, the consumer portal works exactly as if all portlets were local. The user is not aware that bPrime is a remote portlet hosted in a producer application. To test the consumer portlet, minimize aPortlet. The remote portlet, bPrime, responds changing the text it displays.

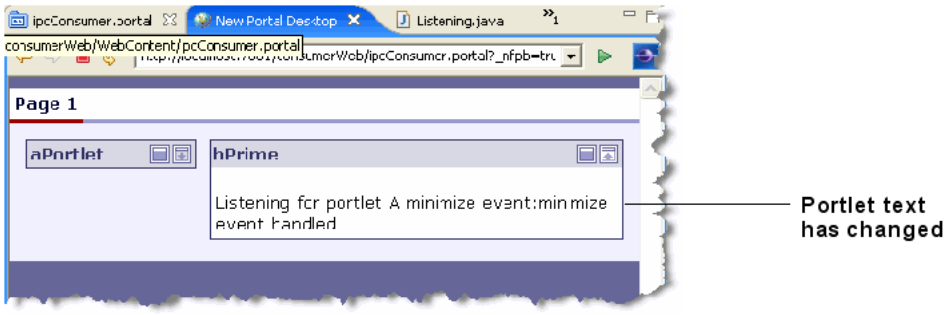
1. Run the portal. To do this, right-click `ipcConsumer.portal` in the Package Explorer and select **Run As** > **Run on Server**.
2. In the Run On Server – Define a New Server dialog, click **Finish**. A browser opens displaying the ipcConsumer portal, as shown in Figure 7–23.

Figure 7–23 Consumer Portal in a Browser



3. In aPortlet, click the **Minimize** button. The portlet aPortlet minimizes and the contents of bPortlet change, as shown in Figure 7–24.

Figure 7–24 Consumer Portal in Browser After Minimize Event



7.3 Inside the Remote Portlet File

Example 7–3 shows an excerpt from the XML content of a `.portlet` file for the remote portlet described previously in this chapter, `bPrime.portlet`. Note that the element `dispatchToRemotePortlet` is added as part of the `handleEvent` definition. This element indicates that the consumer must dispatch the event to the producer.

Example 7–3 Excerpt from the bPrime.portlet File

```
...
<netuix:handleEvent event="onMinimize" eventLabel="handlePortalEvent1"
    fromSelfInstanceOnly="false" onlyIfDisplayed="true"
```

```
    sourceDefinitionLabels="aPortlet_1"> <netuix:dispatchToRemotePortlet/>
</netuix:handleEvent>
...
```

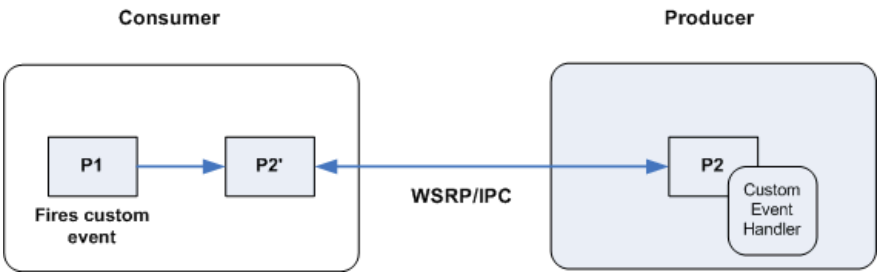
7.4 Data Transfer with Custom Events

Custom events are the recommended method for passing data between portlets deployed in consumer applications and portlets in remote producer applications. This section outlines a possible technique for passing data from a consumer to a producer using custom events. For more information on custom events, see "Custom Event Handling" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

> **Tip:** You can use custom events to pass data between any portlets, whether the portlets are local or on remote producers. For more information on event payloads that can be used with WSRP, see Section 7.5, "Event Payloads Over WSRP."

Figure 7–25 illustrates the configuration of the example discussed in this section.

Figure 7–25 Example configuration



- **P1** – A portlet on the consumer. This portlet gathers data in a form. When the user submits this form, the portlet creates a payload object containing the data and fires a custom event with the payload.
- **P2** – A portlet on the producer. This portlet is configured to listen for the custom event fired by P1. When the event is received, the portlet unpacks the payload and displays it.
- **P2'** – A remote portlet on the consumer (a proxy for P2).

7.4.1 Retrieving the Event on the Producer

This section illustrates how a portlet on the producer can be configured to handle a custom event containing a payload. In this case, the portlet is a Java portlet associated with the class shown in Example 7–4. See also Section 7.5, "Event Payloads Over WSRP."

Example 7–4 Sample Java Portlet Class

```
import java.io.IOException;
import javax.portlet.PortletException;
import javax.portlet.GenericPortlet;
import javax.portlet.RenderResponse;
import javax.portlet.RenderRequest;
import javax.portlet.EventResponse;
```

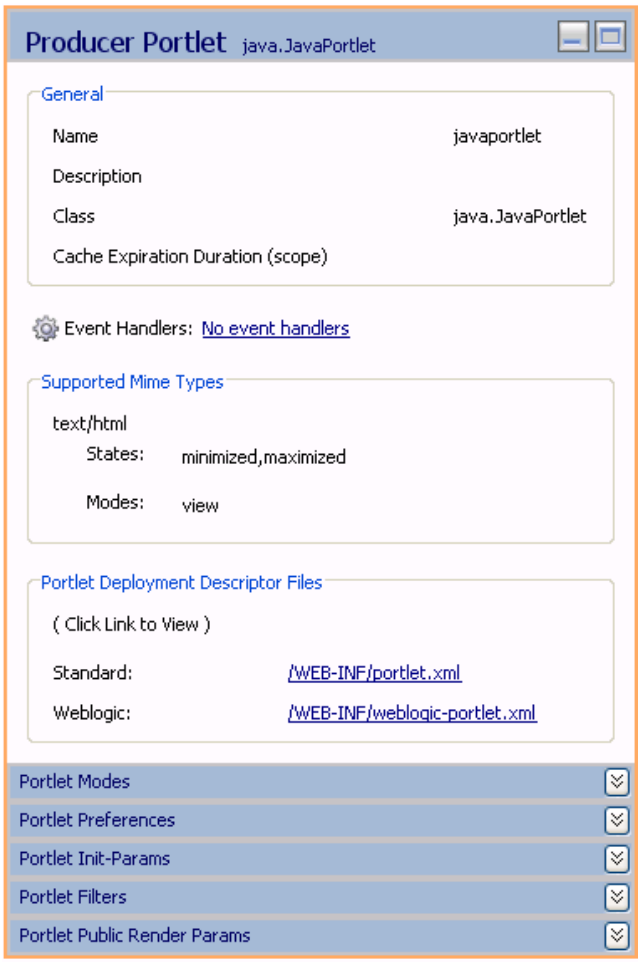
```
import javax.portlet.EventRequest;
import javax.portlet.Event;
import javax.portlet.ProcessEvent;

public class JavaPortlet extends GenericPortlet
{

@ProcessEvent(qname="{urn:com:oracle:wlp:netuix:event:custom}messageCustomEvent")
    public void getMessage(EventRequest request, EventResponse response)
        throws PortletException, IOException
    {
        Event event = request.getEvent();

        // Get the event's payload
        String message = (String)event.getValue();
        response.setRenderParameter("message0", message);
    }

    public void doView(RenderRequest request, RenderResponse response)
        throws PortletException, IOException
    {
        String message = request.getParameter("message0");
        if (message == null) message = "";
        response.setContentType("text/html");
        response.getWriter().write("<p><b>Message From Consumer: </b>" +
        message + "</p>");
    }
}
```

The getMessage() method receives the event object containing the payload sent from the consumer to the producer. In the following steps, you will configure the `portlet.xml` file to mark the portlet as processing this event. Note that in this case the custom event is fired by a portlet deployed to the consumer application.

To configure the event handler in the producer portlet:

1. In Oracle Enterprise Pack for Eclipse, create a Java portlet using the class shown in Example 7–4.
2. In the Package Explorer, double-click the Java portlet file to open the portlet in the editor, as shown in Figure 7–26.

Figure 7–26 Java Portlet in the Editor



3. Click the **No event handlers** link in the Java portlet editor to open the Event Handler dialog box.

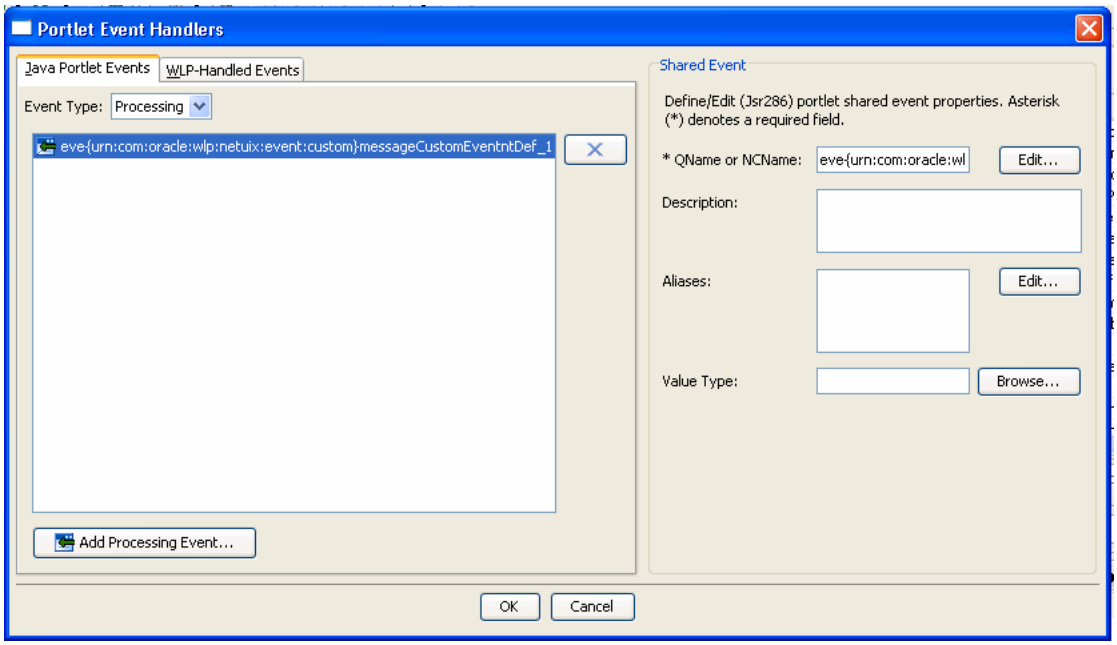
   Tip: The Portlet Event Handlers dialog box lets you create and configure event handlers for a portlet. An event handler listens for an event and takes a specified action when the event is received.

4. In the Portlet Event Handlers dialog, click **Add Processing Event...**
5. In the Define or Choose a Portlet Event Definition dialog, enter `{urn:com:oracle:wlp:netuix:event:custom}messageCustomEvent` in the QName or NCName field and click **OK**.
6. Click **OK** on the Portlet Event Handlers dialog.

   Tip: For detailed information on the Event Namespace and Alias fields, see "About QNames and Aliases" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

   Figure 7–27 shows the completed dialog.

Figure 7–27 Portlet Event Handlers Dialog



The Java portlet is now configured to handle an event called `messageCustomEvent` with the default custom event namespace of `urn:com:oracle:wlp:netuix:event:custom`. When this event is received, the portlet container will invoke the getMessage() method in the Java portlet class. This event handler provides a mechanism for interportlet communication whether the portlets are running locally or on a remote producer.

7.4.2 Firing the Event in the Consumer

A consumer portlet can be configured to fire a custom event, which is then handled on the producer. Example 7–5 illustrates code that could be used in a local portlet on the consumer to fire a custom event and attach a payload to that event.

Example 7–5 Sample Event-Firing Code

```
PortletBackingContext context =
PortletBackingContext.getPortletBackingContext(getRequest());
  context.fireCustomEvent("messageCustomEvent", form.getMessage());
return new Forward("success");
```

Refer to *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal* for more information on the `fireCustomEvent()` method. For more information on portlet development and event handling, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

7.5 Event Payloads Over WSRP

This section discusses how WebLogic Portal handles event payloads sent over WSRP, and explains the rules WLP uses to package and convert event payloads. With knowledge of these rules, you can write portlets to send and/or receive events over WSRP from any other portlet, even if the portlet is on a third-party WSRP producer.

Tip: For additional detailed information on how event payloads are handled over WSRP, review the Javadoc for the class com.bea.wsrp.model.markup.IEventContext.

7.5.1 Overview

Events can be sent from any portlet on the consumer or producer to any other portlet on the consumer or to any producer. The consumer is responsible for receiving events and distributing them to the appropriate portlets.

When events are sent over WSRP, the event payload object must be sent in XML form to be WSRP-compliant, but the WSRP 2.0 specification does not dictate a particular scheme for encoding event payloads to ensure interoperability. It is therefore possible that an event could be received from a portlet on a third-party WSRP producer, and that WebLogic Portal would not know how to convert the event's payload to a Java object. For this reason, Oracle WebLogic Portal avoids converting event payloads to Java objects whenever possible. This allows a Oracle WebLogic Portal consumer to distribute events between portlets on third-party producers even though the consumer cannot understand the event payload.

When event payload conversion into or from a WSRP SOAP message is necessary, WebLogic Portal follows the guidelines discussed in this section. Two special Java objects are used for WSRP event payloads in these cases:

- com.bea.wsrp.ext.holders.NamedStringArray – This class represents the optional NamedStringArray schema type defined in the WSRP 2.0 specification for event payloads, and represents an ordered list of name/value pairs of simple strings.
- com.bea.wsrp.ext.holders.XmlPayload – This class represents arbitrary XML for an event's payload.

7.5.2 How WLP Packages Event Payloads in XML Format

When packaging an event payload into a WSRP SOAP message, WebLogic Portal uses the following logic to convert the event payload to XML. This scenario occurs, for example, when an event is sent by a portlet on the consumer and needs to be delivered to a portlet on a producer, or when a portlet on a WLP producer sends an event.

1. If the event's payload is an instance of the com.bea.wsrp.ext.holders.NamedStringArray class, the event payload is encoded in a WSRP 2.0 NamedStringArray schema type.
2. Otherwise, if the event's payload is an instance of com.bea.wsrp.ext.holders.XmlPayload, the event payload is encoded as the XML represented by the object.
3. Otherwise, if the event's payload is a Java object with a Java Architecture for XML Binding (JAXB) binding, JAXB serialization will be used to encode the payload as XML.
4. Otherwise, the event's payload is serialized using Java serialization, base-64 encoded, and put in a WLP-specific XML event schema type.

7.5.3 How WLP Converts an Event Payload to a Java Object

When converting an event payload from a WSRP SOAP message to a Java object, WebLogic Portal uses the following logic. This scenario occurs, for example, when an event is sent by a producer portlet and delivered to a portlet on the consumer, or when an event is received by a portlet on a WLP producer.

1. If the event's payload in the SOAP message is encoded in a WSRP 2.0 NamedStringArray schema type, it is converted into a com.bea.wsrp.ext.holders.NamedStringArray object.
2. Otherwise, if the event's payload in the SOAP message is encoded in the WLP-specific serialized Java object schema, the object is deserialized. Note that this requires the appropriate Java class that is being deserialized to be in the class path.
3. Otherwise, JAXB deserialization is attempted on the XML representation of the event's payload.
4. If JAXB deserialization fails, an object of type com.bea.wsrp.ext.holders.XmlPayload is created to hold the raw XML representation of the event payload.

7.6 Using Shared Parameters

Shared parameters allow portlets (including remote portlets) to share simple String values with other portlets during all phases of the portlet lifecycle. For detailed information, see "Using Shared Parameters" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

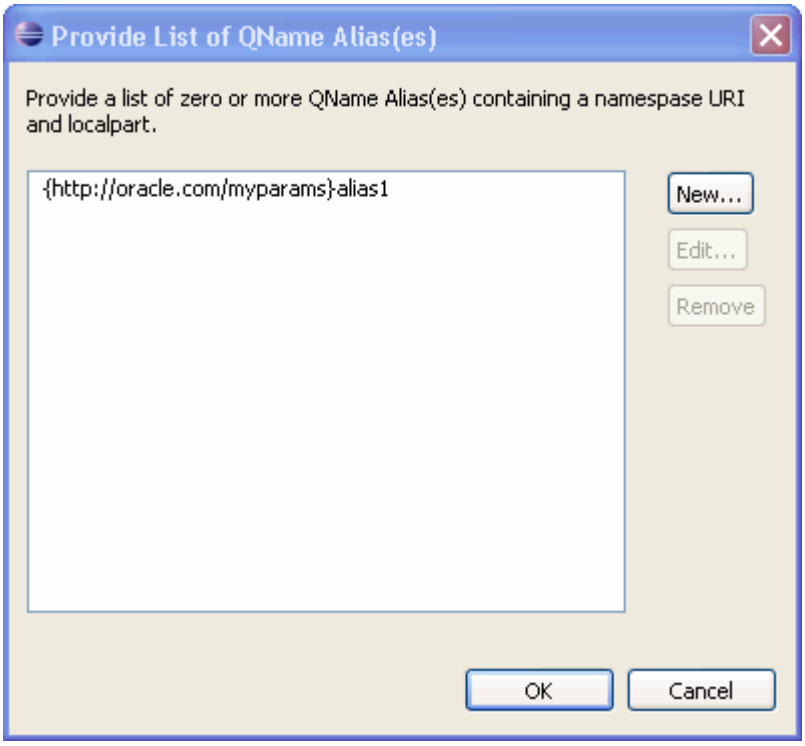
7.7 Adding Event Aliases

If a remote portlet handles a custom event, you have the opportunity to create aliases for the event in the remote portlet. Aliases provide a mechanism for renaming events as they are delivered to individual portlets, allowing communication between portlets that may not have been designed to communicate with each other. For detailed information on aliases, see "About QNames and Aliases" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

To add alias(es) to an event in a remote portlet:

1. Bring up the Portlet Event Handlers dialog. One way to do this is in the Properties view, click the **Event Handlers** button. Another way is to click the Event Handlers link in the portlet editor. If no event handlers exist for the portlet, this link is called **No event handlers**.
2. In the WLP-Handled Events column of the dialog select Handle Custom Event.
3. In the right side of the dialog, click **Edit** next to the Aliases field. The Provide List of QName Alias(es) dialog appears, as shown in Figure 7–28.

Figure 7–28 Provide List of QName Alias(es) Dialog



4. Add one or more aliases to the list. The new button brings up a dialog that helps you to construct the alias properly. For detailed information on constructing alias names, see "About QNames and Aliases" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.
5. Click **OK** to complete the task.

8

Configuring a WebLogic Server Producer

By default, WebLogic Portal projects deployed to a WebLogic Portal domain are configured to function as WSRP producers. If you want to use a Basic WebLogic Server or WebLogic Express domain as a producer, some configuration is required. This chapter explains how to configure a Basic WebLogic Server or WebLogic Express domain as a WSRP producer. Portlets deployed to the server can then be used by consumer applications.

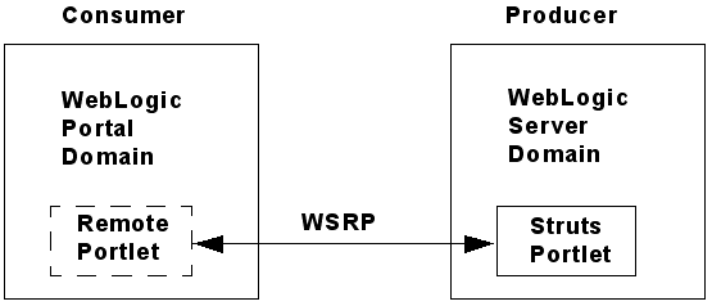
This chapter includes the following topics:

8.1 Introduction

This chapter explains how to configure a basic WebLogic Server domain as a WSRP producer. The example in this section assumes that you have a functioning Struts module deployed in a WebLogic Server domain. The goal of this procedure is to create a portlet in a producer that can be consumed remotely.

By following this procedure, you can expose a Struts application as a remote portlet that a WebLogic Portal application can consume, as illustrated in Figure 8–1.

Figure 8–1 WebLogic Server Producer



To configure a WebLogic Server domain to be a WSRP producer involves the following steps:

- Create a basic WebLogic Server domain.

- Extend the domain to include the producer components.
- Create or reconfigure a web project to include appropriate WebLogic Portal facets that are required for the project to host remoteable components, such as Struts applications.

8.2 Using WSRP in a Basic WebLogic Server Domain

This section explains how to configure a WebLogic Server domain as a producer.

> **Tip:** A producer created in this way is a simple producer. A simple producer is a producer that offers core WSRP services without requiring a full WebLogic Portal installation. In this configuration, some advanced features, such as registration and interportlet communication, are not supported. For more information on simple and complex producers, see Section 3.4, "Understanding Producers and Consumers".

The basic steps you need to perform to enable a WebLogic Server domain to be a WSRP producer are:

- Section 8.2.1, "Create a WebLogic Server Domain"

  In this step, you use the Oracle WebLogic Configuration Wizard to create a WebLogic Server domain with the appropriate elements.

- Section 8.2.2, "Extend the WebLogic Server Domain"

  In this step, you use the Oracle WebLogic Configuration wizard to extend the WebLogic Server domain using an extension template. The extension template adds WSRP producer components to the domain.

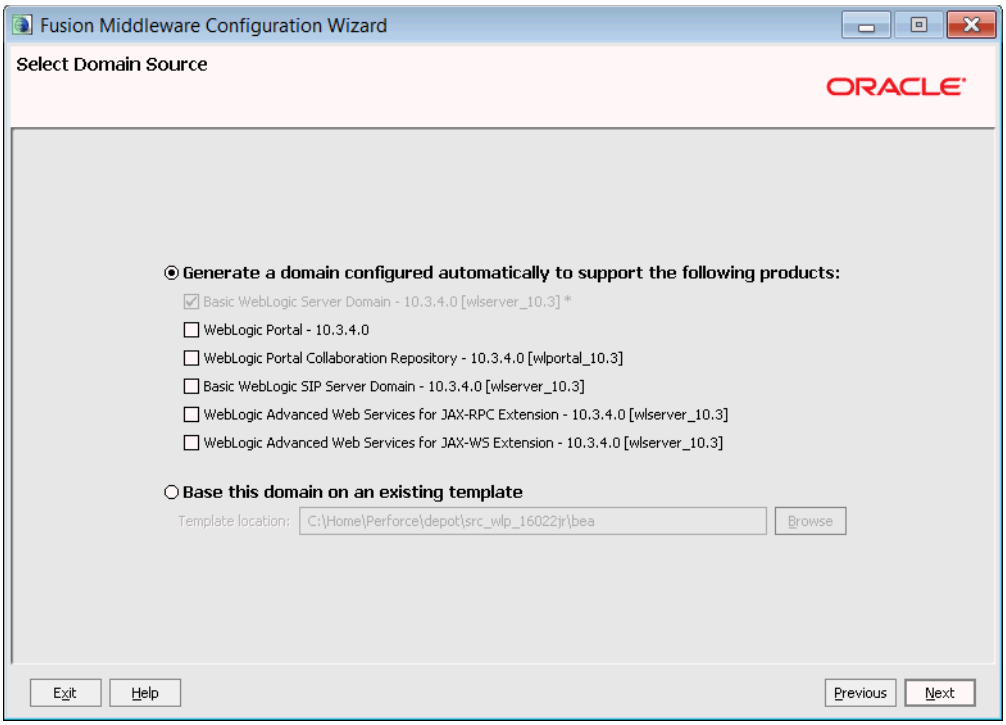
> **Note:** For deploying JPF portlets and Struts portlets, you need to use a full WebLogic Portal domain. For information about creating a full WebLogic Portal domain, see the "Creating a Domain and Server" chapter in *Oracle Fusion Middleware Quick Start Guide for Oracle WebLogic Portal*.

8.2.1 Create a WebLogic Server Domain

Create a new WebLogic Server domain using the Oracle WebLogic Configuration Wizard. You can then extend the domain to include WSRP producer components.

1. Start the Oracle WebLogic Configuration Wizard. To do this, execute the `config.cmd` (or `config.sh`) command in `<WEBLOGIC_HOME>/common/bin`.
2. In the Welcome dialog, select **Create a new WebLogic domain**, and click **Next**.
3. In the Select Domain Source dialog, **Basic WebLogic Server Domain** is selected by default. Leave the other checkboxes unselected, as shown in Figure 8–2.

Figure 8–2 Select Domain Source



4. Complete the rest of the configuration wizard steps to create the WebLogic Server domain. For detailed information on the configuration wizard, refer to *Oracle Fusion Middleware Creating Domains Using the Configuration Wizard*. See also the *Oracle Fusion Middleware Tutorials for Oracle WebLogic Portal*.

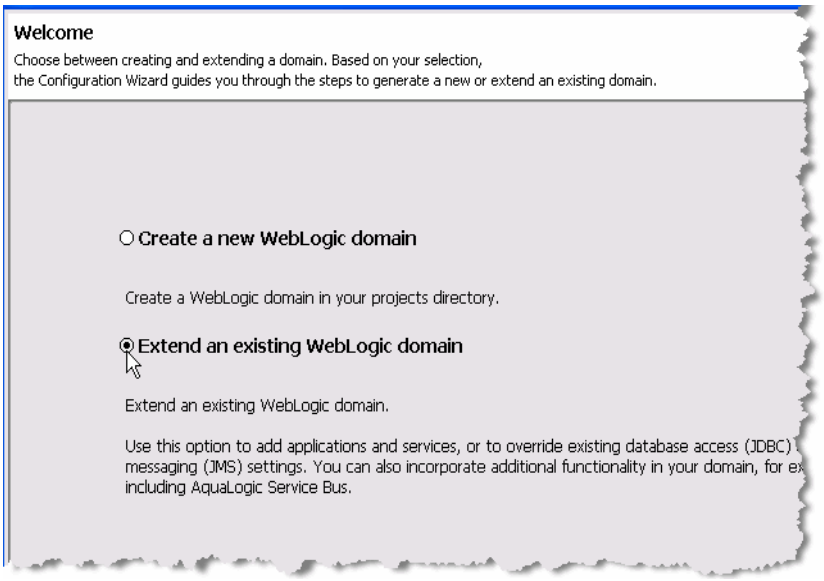
8.2.2 Extend the WebLogic Server Domain

This section explains how to extend your WebLogic Server domain to include the components of a simple producer.

You extend the domain using an extension template. An extension template defines applications and services that can be used to extend an existing domain. The extension template you will use in this example is called `wsrp-simple-producer.jar`.

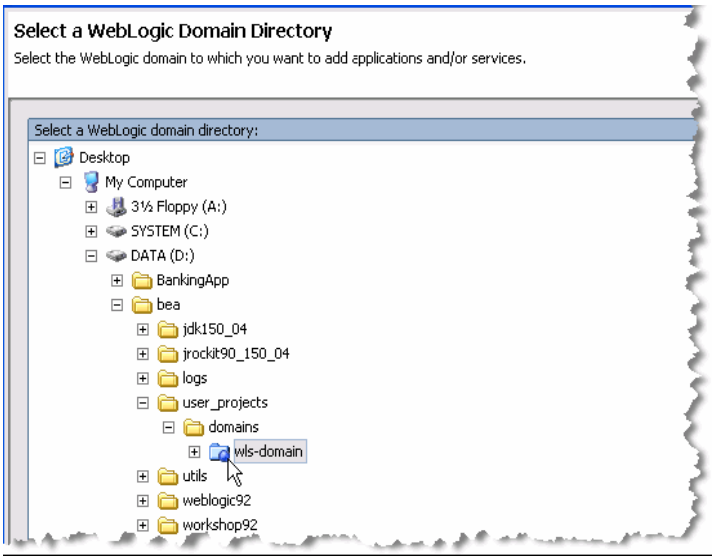
1. Start the Oracle WebLogic Configuration Wizard. To do this, execute the `config.cmd` (or `config.sh`) command in `<WEBLOGIC_HOME>/common/bin`.
2. In the Welcome dialog of the configuration wizard, select **Extend an existing WebLogic domain**, as shown in Figure 8–3, and click **Next**.

Figure 8–3 Extend a Domain



3. In the Select a WebLogic Domain Directory dialog, navigate to the WebLogic Server domain that you want to extend, select it, as shown in Figure 8–4, and click **Next**.

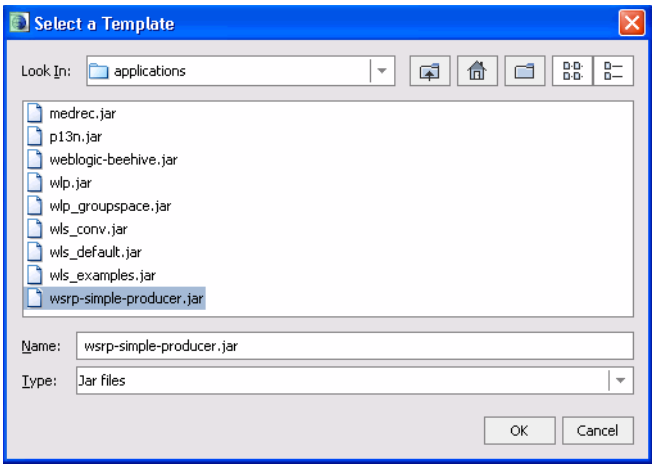
Figure 8–4 Select a Domain Directory



4. In the Select Extension Source dialog, select **Extend my domain using an existing extension template**, and click **Next**.
5. Click **Browse**.
6. In the Select a Template dialog, select the following JAR file, as shown in Figure 8–5:

   ```
   <WLPORTAL_HOME>/common/templates/applications/wsrp-simple-producer.jar
   ```

Figure 8–5 Selecting the Template



1. Click **OK** when you have selected the file.
2. In the configuration wizard, click **Next** and complete the wizard steps as appropriate. When you reach the last dialog, click **Extend**.

   Checkpoint: At this point, you have extended the WebLogic Server domain so that it can function as a simple WSRP producer. Next, you need to configure your web projects.

8.3 Configuring a Web Project

After you have a WebLogic Server domain that is configured to function as a WSRP producer, you also need to enable any web projects that you deploy to function as a WSRP producer in the domain. After you configure a web project to function as a WSRP producer, portlets you deploy in that project will be available to consumers.

8.3.1 Create a Web Project

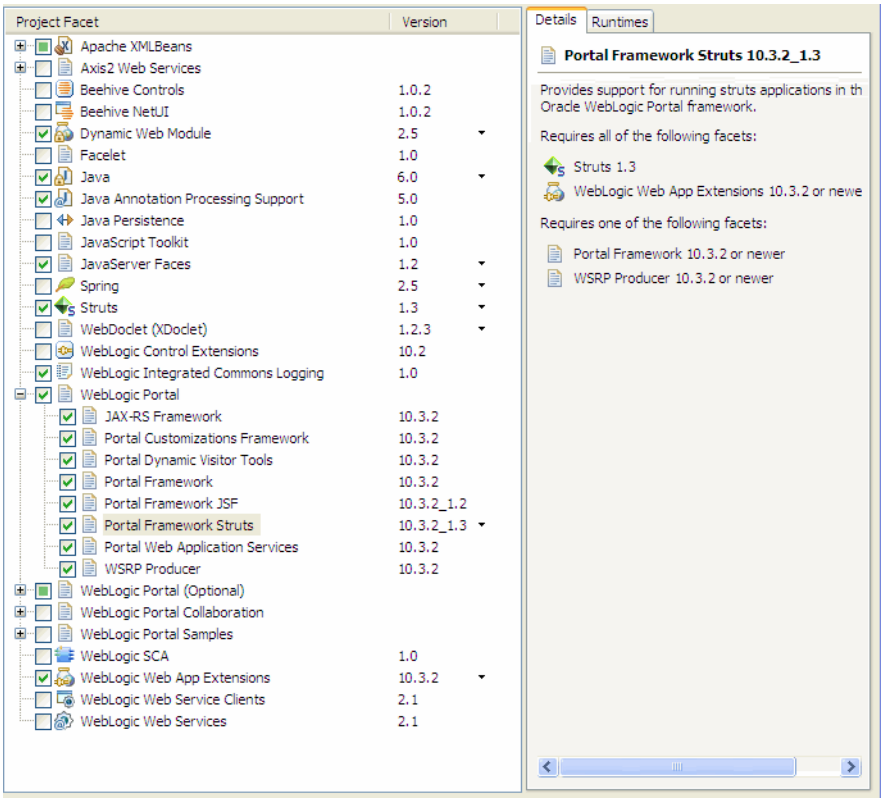
You need to create a web project that is enabled with WSRP producer components. In this example, we demonstrate how to enable a Dynamic Web Project. This type of project does not contain any WebLogic Portal components or WSRP producer components by default.

1. Open Oracle Enterprise Pack for Eclipse.
2. Select **File** > **New** > **Other**.
3. In the New – Select a wizard dialog, open the **Web** folder and select **Dynamic Web Project**. The Dynamic Web Project dialog appears.
4. Enter a name for the project and click **Next**. The Select Project Facets dialog appears.
5. In the Configuration section of the dialog, click **Modify**. The Project Facets dialog appears.
6. In the Project Facets dialog, expand the WebLogic Portal node, and select only the following facets, as shown in Figure 8–6:
   - **Portal Framework Struts**
   - **WSRP Producer**

- **WebLogic Integrated Commons Logging**

Note: Depending on the version of Portal Framework Struts you select, you may need to select other facets. The Project Facets dialog enforces these additional requirements and displays a list of any missing facets. For more information, see "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

Figure 8–6 Select Project Facets



7. Click **Finish**.

   Checkpoint: You have created a web project in which you can create portlets that will be visible to consumers.

8.4 Testing the Producer Configuration

To test the producer configuration, you can do the following:

- Section 8.4.1, "Create a Server on the Producer"
- Section 8.4.2, "Test for a Producer WSDL"
- Section 8.4.3, "Create a Portlet in the Producer Web Application"
- Section 8.4.4, "Consuming a Producer Portlet"

8.4.1 Create a Server on the Producer

If you have not done so, create a WebLogic Server in which to run the application on the producer:

1. Start Oracle Enterprise Pack for Eclipse.
2. Select **File** > **New** > **Other**.
3. In the Select dialog, open the Server folder and select **Server**.
4. Follow the wizard prompts to create the server. Use the WebLogic Server domain that you configured to function as a WSRP producer and add the WSRP-producer enabled web project to the server.
5. Start the server.

> **Tip:** For more information on creating a server using Oracle Enterprise Pack for Eclipse, see *Oracle Fusion Middleware Tutorials for Oracle WebLogic Portal*.

8.4.2 Test for a Producer WSDL

The first test to perform is to check that the producer web application returns a WSDL description when you enter the WSDL URL in a browser.

1. Start WebLogic Server.
2. Enter the WSDL URL for the web project in a browser. For example:

   ```
   http://localhost:7001/myWebProj/producer?wsdl
   ```

   If the server and web application are configured properly, the WSDL file appears in the browser. Part of a sample WSDL file is shown in Figure 8–7.

Figure 8–7 Sample WSDL File

```
- <wsdl:definitions targetNamespace="urn:oasis:names:tc:wsrp:v1:wsdl">
    <wsdl:import namespace="urn:oasis:names:tc:wsrp:v1:bind"
    location="http://www.oasis-open.org/committees/wsrp/specifications/version1/wsrp_v1_bindings.wsdl"/>
    <wsdl:import namespace="urn:bea:wsrp:ext:v1:bind" location="wlp_wsrp_v1_bindings.wsdl"/>
  - <wsdl:service name="WSRPService">
    - <wsdl:port name="WSRPBaseService" binding="urn:WSRP_v1_Markup_Binding_SOAP">
        <soap:address location="http://localhost:7001/strutsHello/producer"/>
      </wsdl:port>
    - <wsdl:port name="WSRPServiceDescriptionService"
      binding="urn:WSRP_v1_ServiceDescription_Binding_SOAP">
        <soap:address location="http://localhost:7001/strutsHello/producer"/>
      </wsdl:port>
    - <wsdl:port name="WLP_WSRP_Ext_Service" binding="urn1:WLP_WSRP_v1_Markup_Ext_Binding_SOAP">
        <soap:address location="http://localhost:7001/strutsHello/producer"/>
      </wsdl:port>
    </wsdl:service>
  </wsdl:definitions>
```

Checkpoint: If the WSDL file appears in the browser, then the server is functioning as a producer. You can now create portlets in the web application that can be consumed as remote portlets in consumer applications.

8.4.3 Create a Portlet in the Producer Web Application

You can use Oracle Enterprise Pack for Eclipse to create portlets in the web application on the producer.

For information on creating portlets, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

8.4.4 Consuming a Producer Portlet

Another test you can perform is to try to consume a portlet deployed in the producer from a WebLogic Portal application.

1. On another machine, create a WebLogic Portal Domain. You can use the WebLogic Configuration Wizard to do this. If you cannot use another machine, be sure the server's listen port does not conflict with the port used by the producer server.
2. Use Oracle Enterprise Pack for Eclipse to create a Portal Application and associate the application with the new WebLogic Portal Domain. If necessary, you can obtain a free developer's version of Oracle Enterprise Pack for Eclipse by visiting the Oracle web site.
3. Create a new Portal Web Project to the application. This application is the consumer application.
4. Create a portal in the consumer application.
5. Start the server that hosts the consumer.
6. Create a remote portlet in the Portal Web Project you just created. Point the WSDL to the web application on the producer. For example:

   `http://producerHost:producerPort/myWebApp/producer?WSDL`

   Where *`producerHost:producerPort`* is the IP address and port number of the machine hosting the producer, and *`myWebApp`* is the name of the context directory for the web application that contains the producer portlet(s) that you wish to surface. See Chapter 4, "Creating Remote Portlets, Pages, and Books" for more information.
7. On the consumer, add the remote portlet to the portal and open the portal. The portlet you created on the producer appears in the portal.

8.4.5 Summary

In this section you tested a configuration where a remote portlet in a consumer references a portlet that is deployed to a producer running in a basic WebLogic Server domain.

8.5 Disabling a WSRP Producer

To disable a WSRP producer, open the `WEB-INF/wsrp-producer-config.xml` file and set the `<service-config>` element's `enabled` attribute to `false`. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
 <wsrp-producer-config
        xmlns="http://www.bea.com/servers/weblogic/wsrp-producer-config/9.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <description>
        This configuration file disables the WSRP producer.
    </description>
    <service-config enabled="false">
 ...
```

9 The Interceptor Framework

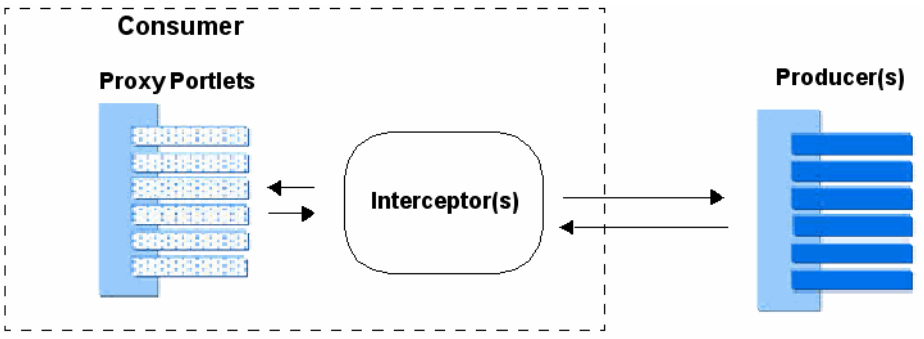
The Interceptor Framework is a consumer-side framework that lets you programmatically intercept and modify markup and user interaction-related WSRP messages sent to and received from producers. This framework exposes a set of interfaces that you can implement. These interfaces let you examine the content of a WSRP message and take specific action based on that content. For example, if a producer sends a registration error back to the consumer, an interceptor can detect that error and display an informative message to the user or, perhaps, automatically return the information required to complete the registration.

This chapter includes the following topics:

9.1 Introduction

As Figure 9–1 illustrates, interceptors are implemented in the consumer. They intercept and allow processing of incoming and outgoing WSRP messages passed between the consumer and one or more producers. Interceptors are associated with specific consumer web applications (web application scoped). You can also group together several interceptors to accommodate more complex use cases.

Figure 9–1 Interceptors Run in Consumer Applications



The interceptor framework defines five public interceptor interfaces. To work with interceptors, you implement one or more of these interfaces and register your implementation classes in a configuration file called `wsrp-consumer-handler-config.xml`. This configuration file is web application-scoped, and resides in the consumer web application's `WEB-INF` directory. See Section 9.6, "Configuring Interceptors" for more information on the configuration file.

To work with interceptors effectively, you must be familiar with basic WSRP operations, such as `getMarkup` and `performBlockingInteraction`. You need to understand the purpose of these operations and how they fit into the life cycle of proxy portlets. See Section 9.4, "Designing Interceptors".

The rest of this chapter explains how to use these interfaces and includes detailed examples and use cases.

9.2 Use Cases

If you are a consumer-side developer, you can use the Interceptor Framework for many different purposes. Some of the most common use cases for interceptors include:

- **Handling Errors** – You can use interceptors to handle errors returned from a producer. For instance, if a specific producer is not registered, you can trap the registration error and handle it as you wish. You may display an informative message to the user, or you may choose to automatically register the producer. An interceptor can also catch an I/O exception, which can occur if the producer is unavailable. In this case, you might choose to handle the error by displaying an informative message for the user, prevent future requests to the producer, or chose to redirect to another producer.
- **Caching Markup** – You can implement an interceptor to cache markup returned from a producer. This feature allows you to use any external caching system you choose. In addition, by caching markup on the consumer, you can, in some circumstances, reduce round-trip communication between the consumer and producer.
- **Validating Data** – You can use interceptors to filter user submitted data. If you detect the user's data is invalid, you can display an informational message, or you can prevent the data from being sent to the producer.
- **Replacing Markup** – An interceptor can filter, replace, modify markup data sent from the producer. An interceptor can also modify the navigational state of a remote portlet. For information on navigational state, see Section 3.5, "Life Cycle of a Remote Portlet".

- **Modifying HTTP Headers** – Interceptors can add or remove some kinds of HTTP headers, and can also inspect response headers. Refer to *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal* for details on which kinds of HTTP headers can be modified by Interceptors.

9.3 Basic Steps

This section lists the basic steps involved in creating an interceptor. More detailed information on each step is available in the other sections of this chapter. The basic steps include:

- **Determine the purpose of your interceptors.** When you know the work you want to accomplish with interceptors, you can then decide which of the interfaces to implement. For more information, see Section 9.4, "Designing Interceptors".
- **Configure the interceptors.** After you know the names of your interceptor classes, you need to specify the interceptor classes in a configuration file. See Section 9.6, "Configuring Interceptors" for detailed information.
- **Implement the interceptor interface(s)**. The interceptor interfaces are discussed in Section 9.5, "Interceptor Interfaces". For more detailed information on the interceptor interfaces, you can refer to *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*.
- **Test the interceptors.**

9.4 Designing Interceptors

When designing interceptors, you must first decide what kind of work you want to perform. Depending on the task, you can implement one or more of the interfaces. Each interface is designed to handle a particular type of WSRP operation. For instance, if you are interested in intercepting form data before it is sent to a producer, you might choose to implement the IBlockingInteractionInterceptor. If you are handling registration faults, then you might implement all of the interfaces.

Interceptors are designed to handle the following types of WSRP operations. These operations are wrapped in SOAP messages that are passed between consumers and producers using WSRP:

- `initCookie` and `initCookieResponse`
- `getMarkup` and `getMarkupResponse`
- `performBlockingInteraction` and `performBlockingInteractionResponse`
- `handleEvents` and `handleEventsResponse`
- `getRenderDependencies` and `setRenderDependencies`

To use interceptors effectively, you need to be familiar with the purpose of these operations and how they relate to the life cycle of a proxy portlet. For instance, `performBlockingInteraction` requests are sent when a user submits form data in a portlet.

> **Tip:** *If you are interested in learning more about WSRP and the preceding types of WSRP operations, see Inside WSRP* at `http://www.oracle.com/technology/pub/articles/dev2arch/2005/03/inside_wsrp.html`. For a more general overview, see Chapter 3, "Federated Portal Architecture."

When designing interceptors, also think about the number of interceptors you need to accomplish your work. You can associate more than one interceptor with a producer by creating a group of interceptors. A group is subject to specific rules that govern the order in which methods are executed. For more information see Section 9.7, "Order of Method Execution".

> **Tip:** Because every request might not have the same data available, it is important to add proper null-condition checks and take appropriate action if data is missing.

9.5 Interceptor Interfaces

This section describes the five public interceptor interfaces, their methods, method return values, and the context objects that are accessible to the interface methods. This section includes these topics:

- Section 9.5.1, "Context Objects"
- Section 9.5.2, "Interfaces"
- Section 9.5.3, "Interface Methods"
- Section 9.5.4, "Interceptor Method Return Values"

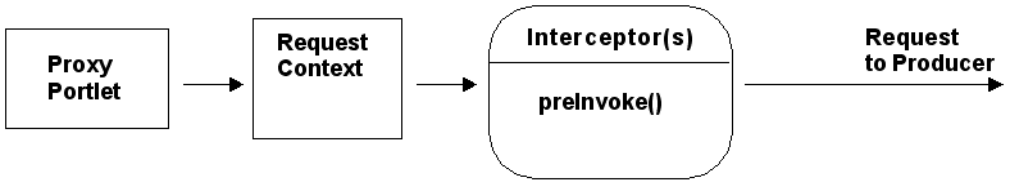
9.5.1 Context Objects

The interceptor methods receive context objects that you can use to get and set values in the intercepted SOAP messages. The context object created for each type of interceptor varies depending on the WSRP operation it represents. For instance, the `initCookie` context object does not contain the same information as the context object for the `handleEvents` operation. For detailed information on these objects, refer to *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal* for the interceptor interfaces. This section describes the flow in which request and response context objects are created and used by interceptors.

Before a message is sent to a producer, or after it is received, the interceptor framework creates an appropriate context object that is passed to the interceptor methods. This object wraps certain elements related to the message. Using methods of the context object, the interceptor can retrieve and set these elements. For example, when a user clicks a link in a remote portlet, the interceptor framework creates a request context object which it then passes to the `preInvoke()` method of the interceptors. After passing through the interceptors and possibly being modified, the request object is used to construct a message that is sent to the producer. Likewise, the interceptor framework constructs a response context object from an incoming message and passes the object the appropriate interceptor methods.

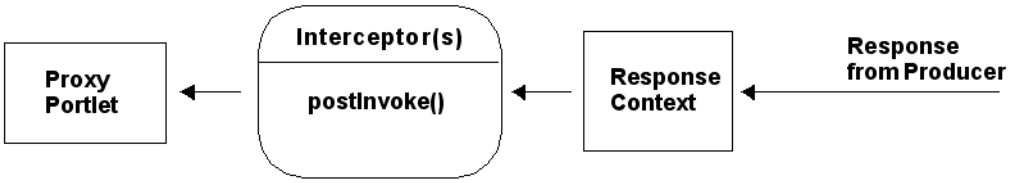
As illustrated in Figure 9–2, a request context is passed from the proxy portlet to the `preInvoke()` methods of registered interceptors. The request context contains information related to the portlet. After processing by one or more interceptors, the interceptor framework creates a message. This message includes any modifications made by the `preInvoke()` method.

Figure 9–2 Handling a Request Context Object



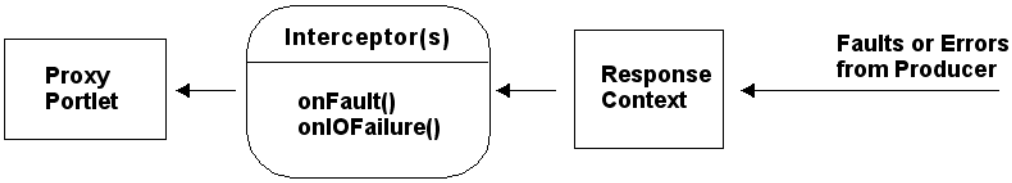
Similarly, as shown in Figure 9–3, the response context object created from an incoming message is passed to the `postInvoke()` method of the interceptors that are associated with the producer that generated the response.

Figure 9–3 Handling a Response Context Object



Finally, as shown in Figure 9–4, the response context object created from an incoming error or fault message is passed to either the `onFault()` or `onIOFailure()` method. Note that in the case of an `onIOFailure`, a response SOAP message might not be generated.

Figure 9–4 Handling an Error or Fault



9.5.2 Interfaces

The five public interceptor interfaces are summarized in Table 9–2. These interfaces are in the `com.bea.wsrp.consumer.interceptor` package. See *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal* for information on these interfaces.

Table 9–1 Interceptor Interfaces

Interface	Description
IGetMarkupInterceptor	Allows you to intercept and modify a message that is being sent in a `getMarkup` message or received in a `getMarkupResponse`.
IInitCookieInterceptor	Allows you to intercept the `initCookie` request. This request is made the first time a consumer displays a proxy portlet for a given user. The request allows the producer to initialize cookies and return them to the consumer.

Table 9–1 (Cont.) Interceptor Interfaces

Interface	Description
IBlockingInteractionInterceptor	Allows you to intercept and modify a `performBlockingInteraction` message.
IHandleEventsInterceptor	Allows you to intercept a `handleEvents` request or response.
IGetRenderDependenciesInterceptor	Allows you to intercept a `getRenderDependencies` request or response. Render dependencies include cascading stylesheet (CSS) files and scripts, such as JavaScript files, upon which the proper rendering of the portlet depend. For more information on render dependencies, see the section "Portlet Appearance and Features" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.
IGetResourceInterceptor	Allows the consumer to intercept a `getResource` operation and handle getting a resource, such as an image, PDF, or form. This feature allows a you to customize the way resources are retrieved.
IResourceServletInterceptor	Allows you to intercept requests and responses to the resource proxy servlet. The preInvoke and postInvoke parameters (see Section 9.5.3, "Interface Methods") are configured in the web.xml as a servlet init-param named `resource-servlet-interceptors`, defined using a pipe (`|`) separated list of classes. The classes must implement `com.bea.wsrp.consumer.resource.IResourceServletInterceptor`. For more information, see Section 9.9, "Using Resource Proxy Interceptors."

9.5.3 Interface Methods

Each interceptor interface includes the same four methods. Table 9–2 summarizes the interceptor methods and when each method is called. Possible return values for each method are discussed in Section 9.5.4, "Interceptor Method Return Values".

> **Tip:** The following table is a general summary only, and does not include method parameters or return values. The specific method signatures depend on the interface in which the method is used. Refer to the *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal* for a detailed description of each method and its parameters.

Table 9–2 Interceptor Methods

Method	Description
`preInvoke()`	This method is called before creating a SOAP message to send to the producer. For example, this method is called after a user clicks on a link in a proxy portlet. One use of this method is to intercept a user's input data to verify that it is complete.
`postInvoke()`	This method is called after a producer has processed its request and sent a response back to the consumer. This method can be used to intercept and filter the markup returned by the producer.
`onFault()`	This method is called when the producer returns a fault. This method can be used to examine the error and display an informational message or take another appropriate action.

Table 9–2 (Cont.) Interceptor Methods

Method	Description
`onIOFailure()`	This method is called when the there is an IOException while sending or receiving a message. This method can be used to display an informational message or take another appropriate action.

9.5.4 Interceptor Method Return Values

The following tables list the possible return values for each of the four interceptor methods:

- Table 9–2, " Interceptor Methods"
- Table 9–2, " Interceptor Methods"
- Table 9–2, " Interceptor Methods"
- Table 9–2, " Interceptor Methods"

For more information on return values, see Section 9.7.3, "How Return Status Affects Execution Order".

Table 9–3 Return Values for preInvoke()

Return Value	Description
`Status.PreInvoke.CONTINUE_CHAIN`	Indicates normal execution.
`Status.PreInvoke.ABORT_CHAIN`	Skips calling `preInvoke()` methods of the subsequent interceptors, but sends the message to the producer.
`Status.PreInvoke.SKIP_REQUEST_ABORT_CHAIN`	Skips calling `preInvoke()` methods of the subsequent interceptors and skips sending the request message to the producer.

Table 9–4 Return Values for postInvoke()

Return Value	Description
`Status.PostInvoke.CONTINUE_CHAIN`	Indicates normal execution.
`Status.PostInvoke.ABORT_CHAIN`	Skips calling `postInvoke()` methods of the subsequent interceptors.

Table 9–5 Return Values for onFault()

Return Value	Description
`Status.OnFault.CONTINUE_CHAIN`	Indicates normal execution. The consumer will handle the fault if rest of the interceptors also return `CONTINUE_CHAIN` status.
`Status.OnFault.ABORT_CHAIN`	Skips calling `onFault()` methods of the subsequent interceptors. The consumer will handle the fault.
`Status.OnFault.RETRY`	Re-sends the message that caused the fault. The `onFault()` methods of the subsequent interceptors are not called.
`Status.OnFault.HANDLED`	Skips calling `onFault()` methods of the subsequent interceptors and assumes that fault has been consumed by the interceptor. The interceptor is responsible for providing all response data.

Table 9–6 Return Values for OnIOFailure()

Return Value	Description
`Status.OnIOFailure.CONTINUE_CHAIN`	Indicates normal execution. The consumer will handle the IO failure if the rest of the interceptors also return `CONTINUE_CHAIN` status.
`Status.OnIOFailure.ABORT_CHAIN`	Skips calling `onIOFailure()` methods of the subsequent interceptors. The consumer will handle the fault.
`Status.OnIOFailure.RETRY`	Re-sends the message that caused the IO failure. The `onIOFailure()` methods of the subsequent interceptors are not called.
`Status.OnIOFailure.HANDLED`	Skips calling `onIOFailure()` methods of the subsequent interceptors and assumes that the IO failure is consumed by the interceptor. The interceptor is responsible for providing all response data.

9.6 Configuring Interceptors

The interceptors are configured in `wsrp-consumer-handler-config.xml`, a web application scoped configuration file. This configuration file requires two entries: `interceptor` and `interceptor-group`. Both of these entries must be present in the configuration file.

> **Tip:** You'll find the `wsrp-consumer-handler-config.xml` file in the Eclipse Project Explorer view under **Merged Project Content/WEB-INF**. To edit this file, right-click it and select **Copy to Project**. Then, open the file in the editor.

The `<interceptor>` element specifies the fully qualified interceptor class name and provides an arbitrary, unique name. The interceptor class must also be in the web application's class path or another accessible class path, such as a system-defined class path. Each interceptor specified by an `<interceptor>` element must be referenced in a group, therefore, you must configure at least one `<interceptor-group>`.

The `<interceptor>` element includes the following elements.

- `name` – A unique name within the scope of a web application.
- `producer-handle` – (Optional) If you specify the handle for a registered producer, the intereceptor(s) in the group will only be called on messages received from or sent to that producer. If you do not specify a producer handle, then the interceptor(s) in the group will be called for all producers associated with the consumer.
- `interceptor-name` – The name(s) of the interceptors you want to include in the group. Use the name(s) specified in the interceptor element(s).

The `<interceptor-group>` element includes the following elements.

- `name` – A unique name within the scope of a web application.
- `producer-handle` – (Optional) If you specify the handle for a registered producer, the intereceptor(s) in the group will only be called on messages received from or sent to that producer. If you do not specify a producer handle, then the interceptor(s) in the group will be called for all producers associated with the consumer.

- `interceptor-name` – The name(s) of the interceptors you want to include in the group. Use the name(s) specified in the interceptor element(s).

For more information on groups, and the order in which methods in groups are called, see Section 9.7, "Order of Method Execution".

Example 9–1 shows a simple configuration, including two interceptors and one group.

Example 9–1 Sample Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<wsrp-consumer-handler-config ...>
    <interceptor>
        <name>AutoRegisteringInterceptor</name>
        <class-name>myInterceptors.AutoRegistrationInterceptor</class-name>
    </interceptor>

    <interceptor>
        <name>ErrorMessageCustomizer</name>
        <class-name>myInterceptors.ErrorMessageCustomizer</class-name>
    </interceptor>
    <interceptor-group>
        <name>Group_1</name>
        <producer-handle>MyProducer</producer-handle>
        <interceptor-name>AutoRegistrationInterceptor</interceptor-name>
        <interceptor-name>ErrorMessageCustomizer</interceptor-name>
    </interceptor-group>
</wsrp-consumer-handler-config>
```

9.7 Order of Method Execution

This section discusses the factors that affect the order of method execution in interceptors and groups of interceptors.

- Section 9.7.1, "Overview"
- Section 9.7.2, "Basic Order Of Execution in a Group"
- Section 9.7.3, "How Return Status Affects Execution Order"
- Section 9.7.4, "Instance Creation and Reuse"
- Section 9.7.5, "Example Chains"

9.7.1 Overview

An interceptor group is a collection of interceptors whose methods are called in a well-defined order. A group can be associated with a specific producer or not associated with any producer. If associated with a single producer, then the interceptors in the group will be called only when requests and responses occur between the consumer and that specific producer. If no producer is associated with a group, then the group's interceptors are called when communication occurs between the consumer and all producers associated with it. For detailed information on configuring a group, see Section 9.6, "Configuring Interceptors".

9.7.2 Basic Order Of Execution in a Group

This section describes the order in which interceptor methods are called if all methods return a status value of `CONTINUE_CHAIN`.

Recall that all interceptors contain four methods: `preInvoke()`, `postInvoke()`, `onFault()`, and `onIOFailure()`. In an interceptor chain, all of the `preInvoke()` methods are executed, then the `postInvoke()` methods, the `onFault()` methods, and finally the `onIOFailure()` methods.

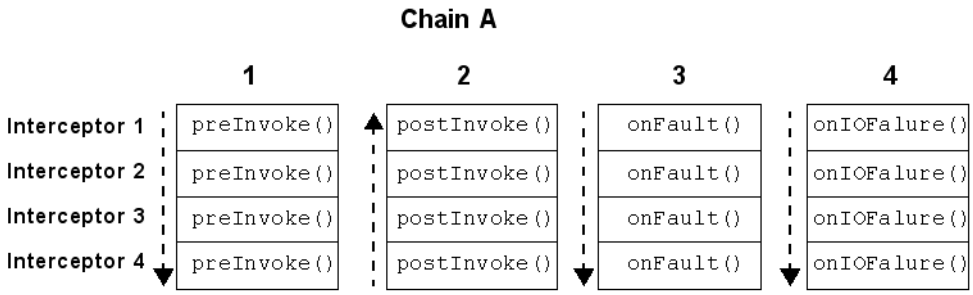
Figure 9–5 illustrates the order in which methods in an interceptor chain are called for the following chain definition:

Example 9–2 Example Interceptor Chain Definition

```
<interceptor-chain>
    <name>Chain-A</name>
    <producer-handle>myProducer</producer-handle>
    <interceptor-name>Interceptor2</interceptor-name>
    <interceptor-name>Interceptor3</interceptor-name>
    <interceptor-name>Interceptor3</interceptor-name>
    <interceptor-name>Interceptor4</interceptor-name>
</interceptor-chain>
```

The illustration assumes that all methods return the `CONTINUE_CHAIN` status. Note that all of the `preInvoke()` methods are called first in the order in which the interceptors appear in the chain configuration, then the `postInvoke()` methods are called in the reverse order. After all the `postInvoke()` methods are called, the `onFault()` methods are called in the order shown in Figure 9–5. Finally, the `onIOFailure()` methods are called in the order shown in Figure 9–5. If `onFault()` or `onIOFailure()` are called, then `postInvoke()` is not called.

Figure 9–5 Default Method Order in Interceptor Chains



Tip: Be aware that you can define interceptors in the configuration file that are associated with specific producers or not associated with any specific producer. An unassociated interceptor does not have a <producer-handle> element defined with it. Unassociated interceptors are always called first for all producer transactions, before the interceptors that are associated with a specific producer are called. Unassociated interceptors are called in the order in which they appear in the configuration file. See Section 9.6, "Configuring Interceptors" for more information.

9.7.3 How Return Status Affects Execution Order

The return status of interceptor methods also affects the order in which interceptor methods are executed. It's helpful to think of chains of interceptor methods. It's easier to understand the way interceptor chains work if you think of four separate chains: a `preInvoke()` chain, a `postInvoke()` chain, an `onFault()` chain, and an

onIOFailure() chain. If you think of chains this way, it's easier to understand the effect of return status on the execution of the chain.

Table 9–2 summarizes the possible return values for interceptor methods and how they affect the order of execution in a chain.

Table 9–7 Interceptor Method Return Values

Return Value	Description
CONTINUE_CHAIN	If all methods return a CONTINUE_CHAIN status, interceptors in a chain are executed in order.
ABORT_CHAIN	Skips calling methods of the subsequent interceptors in the chain, but sends the message on to the producer. A use case for ABORT_CHAIN is when you trap a registration error. If the interceptor is able to fix the error, it can then be re-submitted to the producer.
SKIP_REQUEST_ABORT_CHAIN	Skips calling methods of the subsequent interceptors in the chain and skips sending the request message to the producer. A use case for SKIP_REQUEST_ABORT_CHAIN is when the interceptor performs caching. If markup exists in the cache, there may be no reason to perform further processing and return a message to the producer.
HANDLED	Skips calling the fault-handling methods of the subsequent interceptors in the chain and assumes that fault has been consumed by the interceptor. The interceptor is responsible for providing markup data inputstream, in the absence of it will result in rendering "no markup found error" error message in the portlet.
RETRY	Re-sends the message that caused the fault. The fault-handling methods of the subsequent interceptors in the chain are not called. Only one retry is permitted per message.

Note: If ABORT_CHAIN or SKIP_REQUEST_ABORT_CHAIN is returned from preInvoke(), all of the interceptors will still be called, in reverse order, during the postInvoke() phase.

9.7.4 Instance Creation and Reuse

A new instance of an interceptor implementation class is created for every message before calling preInvoke(). This same instance is reused to call postInvoke(), onFault(), and onIOFailure(). This allows you to set and use instance variables within the scope of a request. For a given instance, all methods are called once; however, preInvoke() and postInvoke() can be called one more time if the RETRY status is returned by either onFault() or onIOFailure(). Only one retry is permitted per message.

9.7.5 Example Chains

This section includes several examples that illustrate the flow of method execution in an interceptor chain. Refer to Table 9–2 for details on interceptor return values referred to in these examples.

Figure 9–6 illustrates the flow in an interceptor chain when the preInvoke() method is called on the chain. When a status of ABORT_CHAIN returned, a message is immediately returned to the producer. The preInvoke() methods of subsequent interceptors in the chain are not called.

Figure 9–6 preInvoke() Chain with ABORT_CHAIN Return Value

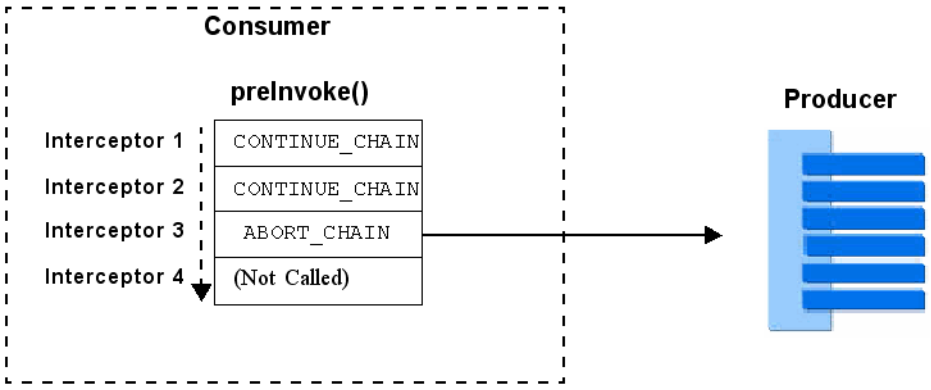


Figure 9–7 illustrates another example of the flow in an interceptor chain when the `preInvoke()` method is called on the chain. When a status of `SKIP_REQUEST_ABORT_CHAIN` is returned, no message is sent to the producer. The `preInvoke()` methods of subsequent interceptors in the chain are not called.

Figure 9–7 preInvoke() Chain

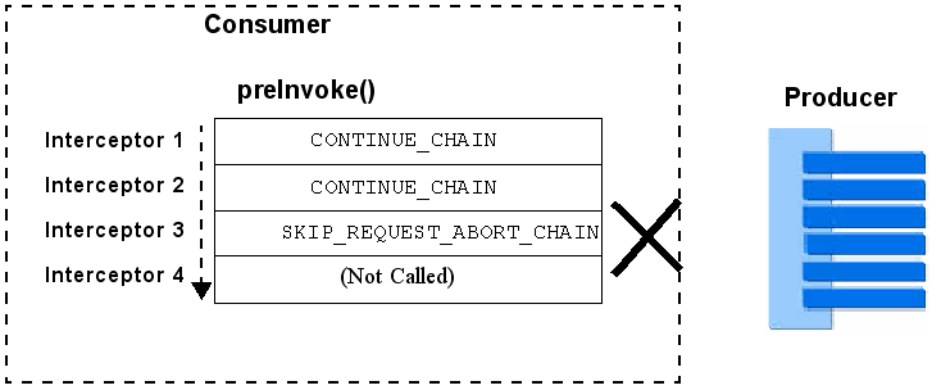


Figure 9–8 illustrates the flow in an interceptor chain when the `onFault()` method is called on the chain. When a status of `RETRY` is returned, the same message that caused the failure, with possible modifications inserted by the interceptor, is returned to the producer. The `onFault()` methods of subsequent interceptors in the chain are not called. Only one retry is permitted. If the same fault is returned, the interceptor framework assumes that the error is handled by the interceptor, and a status of `HANDLED` is returned.

Figure 9–8 onFault() Chain with RETRY Return Value

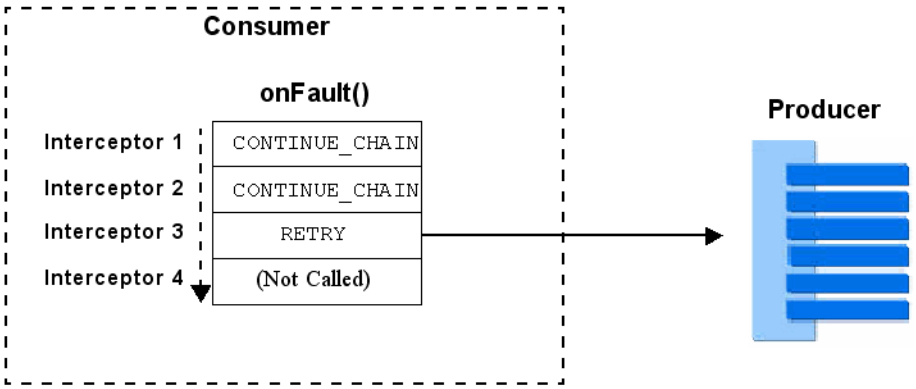
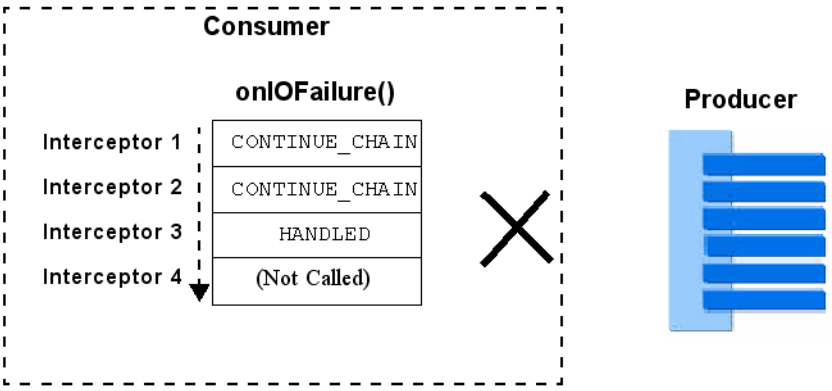


Figure 9–9 illustrates the flow in an interceptor chain when the `onIOFailure()` method is called on the chain. In this case, the no message is returned to the producer, and the framework assumes that fault has been consumed by the interceptor. The `onIOFailure()` methods of subsequent interceptors in the chain are not called. Only one retry is permitted. The second retry is not honored, and the fault or exception is passed to a proxy portlet. If the same fault is returned, the interceptor framework assumes that the error is handled by the interceptor, and a status of `HANDLED` is returned.

Figure 9–9 onIOFailure() Chain with HANDLED Return Value



9.8 Implementing an Error-Handling Interceptor

This section illustrates two simple interceptor implementations. The first implements the `onFault()` method and modifies the error message that is returned to the producer. The second implements `onFault()` and redirects portlet to display an error page.

This section includes these sections:

- Section 9.8.1, "Modifying an Error Message"
- Section 9.8.2, "Including an Error JSP Page"

9.8.1 Modifying an Error Message

You can use interceptors to retrieve and modify exceptions thrown from the producer. In Example 9–3, the `onFault()` method retrieves a Throwable from the response.

You can design an `onFault()` method to examine the exception and take any appropriate action. In this case, the error message is retrieved, modified, and written back to the IGetMarkupResponseContext object. The return status `HANDLED` has the following effects:

- If the interceptor is part of a chain, it skips calling subsequent `onFault()` methods in the chain.
- Returns markup data to the producer. This markup is then displayed in the portlet. If you do not return markup data to the producer, the portlet displays the message "No Markup Found Error."

Example 9–3 ErrorMessageCustomizer

```
import com.bea.wsrp.consumer.interceptor.IGetMarkupInterceptor;
import com.bea.wsrp.model.markup.IGetMarkupRequestContext;
import com.bea.wsrp.model.markup.IGetMarkupResponseContext;
import com.bea.wsrp.consumer.interceptor.Status;
import weblogic.xml.util.StringInputStream;

public class ErrorMessageCustomizer implements IGetMarkupInterceptor
{
    public Status.PreInvoke preInvoke(IGetMarkupRequestContext requestContext)
    {
        return Status.PreInvoke.CONTINUE_CHAIN;
    }

    public Status.PostInvoke postInvoke(IGetMarkupRequestContext requestContext,
                                        IGetMarkupResponseContext responseContext)
    {

        return Status.PostInvoke.CONTINUE_CHAIN;
    }

    public Status.OnFault onFault(IGetMarkupRequestContext requestContext,
                                  IGetMarkupResponseContext responseContext,
                                  Throwable t)
    {
        String message = "This Message is Customized by ErrorMessageCustomizer\n";
        message = message + t.getMessage();
        StringInputStream stringInputStream = new StringInputStream(message);
        responseContext.setMarkupData(stringInputStream);

        return Status.OnFault.HANDLED;
    }

    public Status.OnIOFailure onIOFailure(IGetMarkupRequestContext requestContext,
                              IGetMarkupResponseContext responseContext, Throwable t)
    {
        return Status.OnIOFailure.CONTINUE_CHAIN;
    }
}
```

9.8.2 Including an Error JSP Page

In this example, the `onFault()` method is implemented to include an error JSP page in the portlet.

Example 9–4 DisplayErrorPage Class

```
import com.bea.wsrp.consumer.interceptor.IGetMarkupInterceptor;
import com.bea.wsrp.model.markup.IGetMarkupRequestContext;
import com.bea.wsrp.model.markup.IGetMarkupResponseContext;
import com.bea.wsrp.consumer.interceptor.Status;
import weblogic.xml.util.StringInputStream;
import myClasses.MyError;

public class DisplayErrorPage implements IGetMarkupInterceptor
{
    public Status.PreInvoke preInvoke(IGetMarkupRequestContext requestContext)
    {
        return Status.PreInvoke.CONTINUE_CHAIN;
    }

    public Status.PostInvoke postInvoke(IGetMarkupRequestContext
                    requestContext, IGetMarkupResponseContext responseContext)
    {

        return Status.PostInvoke.CONTINUE_CHAIN;
    }

    public Status.OnFault onFault(IGetMarkupRequestContext requestContext,
                              IGetMarkupResponseContext responseContext,
                              Throwable t)
    {
        try
        {
           if (t instanceof MyError) {
                responseContext.render(requestContext.getHttpServletRequest(),
                                   requestContext.getHttpServletResponse(),
                                   "/redirectTarget/myTarget.jsp");
            } else {
                responseContext.render(requestContext.getHttpServletRequest(),
                                   requestContext.getHttpServletResponse(),
                                   "/redirectTarget/defaultTarget.jsp");
          }
        }
        catch (ServletException e)
        {
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }

        return Status.OnFault.HANDLED;
    }

    public Status.OnIOFailure onIOFailure(IGetMarkupRequestContext
                requestContext, IGetMarkupResponseContext
                responseContext, Throwable t)
    {
        return Status.OnIOFailure.CONTINUE_CHAIN;
    }
}
```

9.9 Using Resource Proxy Interceptors

This section explains how to use the WSRPResourceServletInterceptor and ClipperResourceServletInterceptor.

9.9.1 What is the ResourceProxyServlet

WLP employs a ResourceProxyServlet to proxy resource requests from the consumer portal to the producer. This servlet makes it possible for a consumer to request resources, like images, from a producer (which may be behind a firewall and otherwise unavailable to the consumer). The producer maps resource requests (URLs) to the ResourceProxyServlet on the consumer. When a browser tries to resolve a resource, it makes the request to ResourceProxyServlet on the consumer, which then requests the resource from the producer. By default, ResourceProxyServlet is added to a WLP application's `web.xml` file.

9.9.2 The IResourceServletInterceptor

The IResourceServletInterceptor intercepts requests and responses from the ResourceProxyServlet. This interceptor can be used for:

- Setting up a 2-way SSL
- Setting up an HTTP or SOCKS proxy
- Modifying the target URL
- Adding or removing headers (on both requests and responses)
- Rewriting markup

The IResourceServletInterceptor is used by both WSRP (for proxied and served resources) and the WLP Web Clipper Portlet. Default implementations are provided with WLP for both of these cases.

9.9.3 Configuring the Resource Proxy Interceptors

IResourceServletInterceptor follows the same programming model as the other WSRP interceptors described in this chapter. Like the other interceptors, IResourceServletInterceptor includes preInvoke() and postInvoke() methods. The major difference is that resource proxy interceptors are configured as a servlet `init-param` in the `web.xml` file. This servlet init-param is called resource-servlet-interceptors.

Unlike the other interceptors, the preInvoke and postInvoke parameters are configured in the web application's `web.xml` file. To configure these parameters, use a servlet `init-param` named `resource-servlet-interceptors`. Add the parameter values using a pipe (`|`) to separate the list of interceptor classes. The classes must implement `com.bea.wsrp.consumer.resource.IResourceServletInterceptor`.

Example 9–5 illustrates a resource-servlet-interceptor parameter for a WSRP resource proxy. Example 9–6 illustrates a resource-servlet-interceptor parameter for a Web Clipper Portlet resource proxy.

Example 9–5 Configuring a WSRP Resource Proxy in web.xml

```
<servlet>
        <servlet-name>com.bea.wsrp.consumer.resource.ResourceProxyServlet</servlet-name>
        <servlet-class>com.bea.wsrp.consumer.resource.ResourceProxyServlet</servlet-class>
```

```
        <init-param>
            <param-name>resource-servlet-interceptors</param-name>
            <param-value>com.bea.wsrp.consumer.resource.WsrpResourceServletInterceptor|
               com.bea.wsrp.qa.TestInterceptor</param-value>
        </init-param>
   </servlet>
```

Example 9–6 Configuring a Clipper Portlet Resource Proxy in web.xml

```
<servlet>
        <servlet-name>com.bea.netuix.clipper.ClipperResourceProxyServlet</servlet-name>
        <servlet-class>com.bea.netuix.clipper.ClipperResourceProxyServlet</servlet-class>
        <init-param>
            <param-name>resource-servlet-interceptors</param-name>
            <param-value>com.bea.netuix.clipper.ClipperResourceServletInterceptor|
                 com.bea.netuix.clipper.qa.TestInterceptor</param-value>
        </init-param>
   </servlet>
```

9.9.4 Default Interceptors

If not otherwise configured, both WSRP and Web Clipper Portlets use a default interceptor. The defaults perform most of the basic functions required by the resource proxy servlet.

> **Caution:** Oracle does not recommend that the default resource proxy interceptors be replaced. Instead, add additional interceptors before or after the default interceptor. See Example 9–5 and Example 9–6. You can also extend (subclass) the interceptor.

Some functions performed by the default interceptors include:

- Setting up the request and response contexts:
  1. Parsing the URL
  2. Transferring headers
  3. Rewriting the response markup
- Performing security checks
- Adding the appropriate cookies.

Because the default interceptors perform these functions, Oracle recommends that custom interceptors operating on the request (preInvoke) be placed after the base interceptor, and interceptors operating on the response should be placed before the base interceptor. If using the approach of extending the base interceptor, call the `super` method on any overridden method, as shown in Example 9–7.

Example 9–7 Overriding Base interceptor Methods

```
public class MyInterceptor extends WsrpResourceServletInterceptor {
    @Override
    protected PreInvoke preInvoke(IResourceServletRequestContext requestContext) throws IOException
{
       PreInvoke superPreInvoke = super.preInvoke(requestContext);
       if (superPreInvoke == PreInvoke.CONTINUE_CHAIN) {
```

```
            final HttpURLConnection connection = requestContext.getURLConnection;
            connection.addRequestProperty("X-MY-SECURITY-TOKEN", generateSecurityToken());
        }

        return superPreInvoke;
    }

}
```

9.9.5 More Information

For more information, refer to the Javadoc for WsrpResourceServletInterceptor and ClipperResourceServletInterceptor.

10

Federating User Profiles

WebLogic Portal enables user profile information to be passed from consumers to producers. This feature allows many of the Personalization features available in WebLogic Portal to function in a federated portal. This chapter explains how to work with user profile information in a federated portal. Before a federated portal can use user profile information, some configuration is required in both the consumer and producer applications.

This chapter includes the following topics:

10.1 Introduction

This section summarizes the purpose of user profile propagation and how WebLogic Portal propagates user profile data in a federated environment.

10.1.1 What are User Profiles?

A user profile is a collection of property sets that contain user-specific information. WebLogic Portal provides many features that rely on user profiles. For example, the WebLogic Portal Personalization features rely on user profiles to deliver customized content to specific types of users.

For example, you could create a property set in Oracle Enterprise Pack for Eclipse called human resources that contains properties such as gender, hire date, and e-mail address. This information can be used to personalize the user's experience in your portal. When users log into a portal, the portal can access the property values and target them with personalized content, e-mails, pre-populated forms, and discounts based on the Personalization rules you set up.

See the *Oracle Fusion Middleware Interaction Management Guide for Oracle WebLogic Portal* for more information on personalization. For detailed information on creating user profiles, see *Oracle Fusion Middleware Interaction Management Guide for Oracle WebLogic Portal*.

10.1.2 User Profiles in Federated Portals

For a WebLogic Portal producer to return personalized content to a consumer, user information must be conveyed from the consumer to the producer. The basic requirements for using user profile information in a federated portal include:

- On the producer, declare the user properties to request from the consumer. The best practice is to request only those properties that are required by the portlets that are deployed on the producer. See Section 10.3, "Configuring the Producer" for more information.
- On the consumer, provide a mapping file, if necessary, that maps the requested user properties with equivalent properties that exist on the consumer. The consumer uses the WebLogic Portal Personalization (P13N) API to retrieve the requested user properties on the consumer. See Section 10.4, "Configuring the Consumer" for more information.

  Tip: Once retrieved, the list of the properties required for a specific portlet is stored in the consumer database for future access.

As shown in Figure 10–1, after a consumer first contacts a producer, the producer responds with a list of the portlets it offers and with a request for the user information that each portlet requires.

Figure 10–1 Producer Requests User Information from Consumer

If a portlet requires user information, the consumer will attempt to supply that information as part of the `getMarkupRequest()` to the producer before the portlet can be rendered, as illustrated in Figure 10–2. WebLogic Portal uses the P13N API, typically in conjunction with a mapping file, to retrieve the requested user properties on the consumer.

Figure 10–2 Producer Returns Personalized Content

10.1.3 Platform for Privacy Preferences (P3P)

The WSRP protocol specifies a standard format for storing and exchanging user information. This format, called Platform for Privacy Preferences (P3P), is an internet standard. You can configure WebLogic Portal applications to accept user information presented in this format as well as in the WebLogic Portal user profile format.

See Section 10.5, "P3P Examples" for more information. The P3P specification is available on the W3C web site, `http://www.w3.org/TR/P3P`.

10.2 When to Use this Feature

Use this feature if the user properties defined on the producer and consumer do not match. When exactly the same user properties exist on the consumer and producer, you do not need to use this feature.

> **Tip:** In a production environment, the best practice is to specify a property set and property name for each user property you want to propagate. Retrieving all properties is inefficient when only a small subset of properties is needed.

10.3 Configuring the Producer

To use user profile information in a federated portal, you need to declare on the producer which user properties are required by the portlets deployed on the producer. The declared properties are marshalled in a response to the consumer and returned to the consumer application, which must then return the requested user property values when registering the producer.

The procedures for configuring portlets deployed in a producer to use user profile information differs depending on whether you are configuring Java portlets or non-Java portlets.

This section includes the following topics:

- Section 10.3.1, "Configuring Java Portlets"
- Section 10.3.2, "Configuring Non-Java Portlets"

10.3.1 Configuring Java Portlets

The Java Portlet Specification specifies how Java portlets access user attributes such as the name, e-mail address, phone number, and other attributes of the user. This section explains how to specify user attributes for Java portlets deployed in a producer application, and how Java portlets retrieve user information.

> **Tip:** For detailed information on how user information is accessed by Java portlets, refer to the User Information section of the Java Portlet Specification.

10.3.1.1 Configuring the Deployment Descriptor (portlet.xml)

The Java Portlet Specification defines the `<user-attribute>` element for specifying user attributes required by a deployed Java portlet. Figure 10–1 shows an excerpt of a `portlet.xml` file with user properties specified. The `<name>` elements specify user attribute names.

Example 10–1 Specifying User Properties in portlet.xml File

```
<portlet-app>
...
    <user-attribute>
        <name>Employee/Language</name>
    </user-attribute>
    <user-attribute>
        <name>Employee/Role</name>
    </user-attribute>
...
</portlet-app>
```

See also Section 10.3.1.3, "Creating Default User Property Sets".

10.3.1.2 Retrieving User Information in a Java Portlet

The Java Portlet Specification also specifies how Java portlets retrieve user information from the portal environment in which they are deployed. The portlet can retrieve a Map object that contains the user attributes of the user who initiated the request. You can retrieve this Map object from the request using the `PortletRequest.USER_INFO` constant.

The example code in Example 10–2 shows how a Map of user information is retrieved from the request in a JSP associated with a Java portlet. User property values are retrieved from the Map using the user property names as keys.

Example 10–2 Retrieving User Information in a Java Portlet

```
...

    Map<String, Object> props;
    PortletRequest portletRequest = (PortletRequest)
    request.getAttribute("javax.portlet.request");
    if (portletRequest != null) {
        props = (Map<String, Object>)
       portletRequest.getAttribute(PortletRequest.USER_INFO) ;
    } else {
        props = null ;
    }

    if (props == null) {%>
        <p>Empty Profile</p>
    <%} else {%>
       <p><%= props.get("Employee/Language") %></p>
       <p><%= props.get("Employee/Role") %></p>
    <%}%>
...
```

10.3.1.3 Creating Default User Property Sets

Example 10–3 shows a sample user attribute specified in a `portlet.xml` file. This section explains how you can streamline the `<user-attribute>` property by creating default user property sets. For example, by creating a default user property called "Employee," the name attribute in Example 10–3 could be shortened to `<name>Language</name>`.

Example 10–3 User Attribute Specified in portlet.xml

```
<user-attribute>
    <name>Employee/Language</name>
</user-attribute>
```

To create a default user property set, first create a `weblogic-portlet.xml` file in the `WEB-INF` directory of your portal web application. You can then use the `<user-property-set>` attribute to configure default user property sets.

Example 10–4 shows how to create a default user property set called "Employee" for all portlets in the web application:

Example 10–4 Default Property Set Applied to All Portlets

```
<portal-container>
```

```
    <user-property-set>Employee</user-property-set>
</portal-container>
```

Example 10–5 shows how to create a default user property set for a specific portlet:

Example 10–5 Default Property Set Applied to Specific Portlets

```
<portlet>
    <name>portletName</name>
    <user-property-set>Employee</user-property-set>
    ....
</portlet>
```

With the default user property set "Employee" specified in `weblogic-portlet.xml`, you can then code the `<user-attribute>` value shown Example 10–3 as follows in the `portlet.xml` file.

Example 10–6 User Attribute

```
<user-attribute>
    <name>Language</name>
</user-attribute>
```

For more information on the `weblogic-portlet.xml` file, see "Building Portlets" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

10.3.1.4 Mapping User Properties

If the user properties on the consumer and producer do not match, you can create a mapping file on the consumer. A mapping file allows the consumer to retrieve user properties that map to the properties requested by the producer. For detailed information on mapping user properties, see Section 10.4, "Configuring the Consumer".

10.3.2 Configuring Non-Java Portlets

This section explains how to specify user attributes for non-Java portlets deployed in a producer application.

10.3.2.1 Configuring the Deployment Descriptor File

For non-Java portlets, you specify required user properties in the descriptor file `wsrp-producer-config.xml`. This file is located in the `WEB-INF` directory of your producer web application. Example 10–7 shows a sample `wsrp-producer-config.xml` file. The `<requiredUserProperties>` element specifies the required user properties for portlets deployed in the producer web application (shown in bold type). In the example, the value `All` specifies that consumer must supply all available user profile information to the producer. Other possible values are discussed in this section.

Example 10–7 Sample wsrp-producer-config.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<wsrp-producer-config
    xmlns="http://www.bea.com/servers/weblogic/wsrp-producer-config/9.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:uddi="urn:uddi-org:api_v2"

xsi:schemaLocation="http://www.bea.com/servers/weblogic/wsrp-producer-config/9.0
```

```
wsrp-producer-config.xsd">
    <description></description>
    <service-config>
        <registration required="true" secure="false"/>
        <service-description secure="false" supports-method-get="true"/>
        <markup secure="false" rewrite-urls="true" transport="string"/>
        <portlet-management required="true" secure="false"/>
    </service-config>
    <supported-locales>
        <locale>en</locale>
        <locale>en-US</locale>
    </supported-locales>
    <requiredUserProperties properties="All">
    </requiredUserProperties>
</wsrp-producer-config>
```

The `<requiredUserProperties>` element contains one attribute, called `properties`, which takes one of these three values:

- **`All`** – Instructs the consumer to send all user profile information. For example:

  ```
  <requiredUserProperties properties="All">
  ```

- **`None`** – Instructs the consumer to send no user profile information. For example:

  ```
  <requiredUserProperties properties="None">
  ```

- **`Specified`** – Instructs the consumer to send only specified user profile information. Use the `<specifiedProperties>` sub-element to list the user information required by the portlet. For example:

  ```
  <requiredUserProperties properties="specified">
   <description>These are required properties</description>
   <specifiedProperty name="Employee/name"/>
   <specifiedProperty name="Employee/gender"/>
   <specifiedProperty name="Employee/number"/>
  </requiredUserProperties>
  ```

The value given for the `name` property can take one of these forms:

- *`propertySet/propertyName`* – The name of a property set defined on the producer and the name of a property in that property set. For example:

  ```
  <requiredUserProperties properties="specified">
   <specifiedProperty name="Employee/gender"/>
  </requiredUserProperties>
  ```

- *`propertySet/*`* – The name of a property set defined on the producer and an asterisk (*), which specifies that all properties in that property set are required. For example:

  ```
  <netuix:requiredUserProperties properties="specified">
   <specifiedProperty name="Employee/*"/>
  </requiredUserProperties>
  ```

- *`p3pName`* – Specify P3P user properties. For example:

  ```
  <requiredUserProperties properties="specified">
  ```

```
    <specifiedProperty name="name/given"/>
    <specifiedProperty name="gender"/>
   </requiredUserProperties>
```

If no user information is specified in `wsrp-producer-config.xml`, the behavior is the same as if a value of `None` were specified in `<requiredUserProperties>`.

Retrieving User Information in a Portlet

The code excerpt in Example 10–8 shows how user properties are retrieved in a portlet's JSP file using the P13N tag `<profile:getProperty>`.

Example 10–8 Retrieving Values in a Portlet

```
...
<%
    if (request.getUserPrincipal() != null) {
  %>
    <profile:getProfile profileKey="<%= request.getUserPrincipal().getName() %>"
/>
  <%
  } else { %>
      <profile:getProfile profileKey="anonymous" groupOnly="true" />
  <%
  }
  %>

  <tr>
<td>Name</td>
<td id="wsrp_date"><profile:getProperty propertySet=
"Employee" propertyName="name"/></td>
  </tr>
  <tr>
    <td>Gender</td>
    <td id="wsrp_int_code"><profile:getProperty propertySet=
    "Employee" propertyName="gender"/></td>
    </tr>
  <tr>
...
```

10.3.2.2 Handling User Property Extensions

If a WebLogic Portal or non-WebLogic Portal consumer sends extended P3P user profile information, the portlet can retrieve the extensions as a List object obtained from the `<profile:getProperty>` tag. Example 10–9 shows example code that extracts a List containing telephone extensions. In this case, the property `homeInfo/postal/extensions` is an extended WSRP user property.

Example 10–9 Retrieving User Profile Extensions

```
<profile:getProperty propertySet="<%= UserProperty.P3P_PROPERTY_SET_NAME %>"
propertyName="homeInfo/postal/extensions" id="postalExtsObj"/>
        <%
            List<Element> teleExts = (List<Element>) postalExtsObj;
            if (teleExts != null) {
                for (int i = 0 ; i < teleExts.size() ; i++) {
                   String extStr = teleExts.get(i)

        %>
```

```
                <tr> <td>Postal Extension[<%= i %>]</td>
                <td colspan="2"
                id="postal_extensions[<%=i%>]"><%= extStr %></td> </tr>
<%          }
      }%>
```

10.3.2.3 Mapping User Information on the Consumer

Consumers may map the user properties requested by producers to properties that exist on the consumer. For detailed information on mapping user properties, see Section 10.4, "Configuring the Consumer".

10.4 Configuring the Consumer

In many cases, the user property set and property names that exist on a producer do not match those on the consumer. Therefore, WebLogic Portal allows you to map these names appropriately. This section explains how to map property set and property names using a configuration file or programatically with a mapping class.

This section includes these topics:

- Section 10.4.1, "Using a Mapping File"
- Section 10.4.2, "Using a Mapping Class"
- Section 10.4.3, "Mapping Constants"

10.4.1 Using a Mapping File

Specify user profile mappings in the `wsrp-user-property-config.xml` file. This file is located in the `WEB-INF` directory of the consumer web application.

As shown in Example 10–10, the element `<wsrp-user-property-map-bean>` is the top-level element that can appear in this configuration file. The elements that can fall under `<wsrp-user-property-map-bean>` are shown in bold type and include:

- **`<user-property-map>`** – Creates a producer to consumer mapping that applies to all producers registered with the consumer.
- **`<producer-user-property-map>`** – Creates a mapping tied to a specific producer, indicated with the `<producer-handle>` element.
- **`<mapper-class-name>`** – Lets you supply a class that performs mappings programatically. You must specify the fully qualified class name of the mapping class. For more information on creating a mapping class, see Section 10.4.2, "Using a Mapping Class".

As shown in Example 10–10, the `<producer-user-property-map>` element can be used to create producer-specific mappings directly or with a mapping class.

Example 10–10 Example wsrp-user-property-config.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<wsrp-user-property-map-bean xmlns="http://www.bea.com/ns/portal/90/wsrp-user-property-config">

    <!-- Maps ldap/name -> Employee/name for all registered producers -->
    <user-property-map>
        <producer-property-name>Employee/name</producer-property-name>
        <consumer-property>ldap/name</consumer-property>
    </user-property-map>
```

```
    <!-- Specifies a mapper class to apply to all registered producers -->
    <mapper-class-name>myClasses.MyUserPropertyMapper1</mapper-class-name>

    <!-- User Property Map for specific producer -->
    <producer-user-property-map>
        <producer-handle>complexProducer</producer-handle>
        <user-property-map>
            <producer-property-name>Employee/number</producer-property-name>
            <consumer-property>"xxxxxx"</consumer-property>
        </user-property-map>
    </producer-user-property-map>

    <!-- Specifies a mapper class for specific producer -->
    <producer-user-property-map>
        <producer-handle>complexProducer2</producer-handle>
        <mapper-class-name>myClasses.MyUserPropertyMapper2</mapper-class-name>
    </producer-user-property-map>
</wsrp-user-property-map-bean>
```

The `<producer-property-name>` sub-element of `<user-property-map>` specifies the *`propertySet/propertyName`* pair of the requested producer property, and the `<consumer-property>` sub-element specifies the equivalent pair that exists on the consumer.

The `<producer-property-name>` and `<consumer-property>` pairs can take the following forms:

- *`propertySetName/propertyName`* – The name of a property set and the name of a property in that property set. For example:

  ```
  <producer-property-name>propertySetName-A/propertyName-A</producer-property-nam
  e>
  <consumer-property>propertySetName-B/propertyName-B</consumer-property>
  ```

- *`propertySetName/*`* – The asterisk (*) specifies that all properties in that property set are mapped. This pattern assumes that the same property names exists in the mapped property sets on the consumer and the producer.

  For example, the following lines map all properties in `propertySetName-A` on the producer to `propertySetName-B` on the consumer.

  ```
  <producer-property-name>propertySetName-A/*</producer-property-name>
  <consumer-property>propertySetName-B/*</consumer-property>
  ```

- *`propertyValue`* – Maps a property name from the producer to a constant value. For example, the following lines map the property called `propertyName-A` from the producer to an arbitrary string constant. In addition to strings, you can specify other types of constants. For more information, see Section 10.4.3, "Mapping Constants".

  ```
  <producer-property-name>propertySetName-A/propertyName-A
  </producer-property-name>
  <consumer-property>"aStringValue"</consumer-property>
  ```

10.4.2 Using a Mapping Class

In addition to using a mapping file to map requested producer properties to consumer properties, you can create a mapping class to programmatically map and set user property values on the consumer. To use a mapping class, you need to do the following:

- Write the mapping class.
- Configure the mapping class in the `wsrp-user-properties-config.xml` file.

10.4.2.1 Writing the Mapping Class

To create a mapping class:

1. Extend the com.bea.wsrp.consumer.userproperty.DefaultUserPropertyMapper class in *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*.
2. Override the `getProducerProperties` method to implement the mapping functions that you want to create. For detailed information on this method, refer to the *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*. The mapper class example in Example 10–11 sets the gender property for a user based on the user's name.

   Note: Extending DefaultUserPropertyMapper and overriding getProducerProperties is the simplest and best practice, although it is not required. You can also extend its abstract base class if you want to.

3. Configure the mapper class in the `wsrp-user-property-config.xml` file. To do this, add lines to `wsrp-user-property-config.xml` that follow the pattern shown in Example 10–11, where *`producerHandle`* is the unique name that identifies the producer on the consumer, and *`myClasses.MyMapperClass`* is the full class name of the mapper class.

Example 10–11 Example Mapper Class

```
package com.bea.portlet.qa.wsrp.userprops;

import java.util.Arrays;
import java.util.Collection;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import com.bea.p13n.property.EntityPropertyCache;
import com.bea.wsrp.consumer.userproperty.DefaultUserPropertyMapper;
import com.bea.wsrp.consumer.userproperty.RequiredUserProperties;
import com.bea.wsrp.consumer.userproperty.UserProperty;

public class TestUserPropertyMapper extends DefaultUserPropertyMapper {
    private final static Set<String> MALE_NAMES = new HashSet<String>() ;
    private final static Set<String> FEMALE_NAMES = new HashSet<String>() ;

    static {
        final String[] maleNames = {"Nate","Nathan","Eric","Subbu","Scott"};
        MALE_NAMES.addAll(Arrays.asList(maleNames)) ;
        final String[] femaleNames = {"Mandy","Geeta","Jenn","Jen","Jenny"} ;
        FEMALE_NAMES.addAll(Arrays.asList(maleNames)) ;
    }

    /**
     * Map set the user's gender if user.name.given is set
     * @param requiredProperties the properties requested by the producer
     * @param map  A map where the key is the producer's name and
     * the value is the consumer's name
     * @param profile the User's profile on the consumer
```

```
     * @return the properties mapped to the producer
     */
    public Collection<UserProperty> getProducerProperties(
            RequiredUserProperties requiredProperties,
            Map<String, String> map,
            EntityPropertyCache profile) {

        final Collection<UserProperty> properties =
        super.getProducerProperties(requiredProperties, map, profile) ;
        if (requiredProperties.isPropertyRequired("HR", "gender")) {
            final String givenName = (String) getProperty(profile, "HR", "name.given") ;
            if (MALE_NAMES.contains(givenName)) {
                addUserProperty(properties, "HR", "gender", "M") ;
            } else if (FEMALE_NAMES.contains(givenName)) {
                addUserProperty(properties, "HR", "gender", "F") ;
            }
        }
        return properties ;
    }
}
```

10.4.2.2 Configuring the Mapping Class

You need to declare mapping classes in the `wsrp-user-properties-config.xml` file. To do this, use the `<mapper-class-name>` element. This element takes a fully qualified class name as its property, as shown in the following example:

```
<mapper-class-name>myClasses.MyMapperClass</mapper-class-name>
```

You can place this element directly under the `<wsrp-user-property-map-bean>` element or the `<producer-user-property-map>` element. For detailed information on the configuration file, see Section 10.4.1, "Using a Mapping File".

10.4.3 Mapping Constants

In addition to mapping user properties to user properties, you can map user properties to constant values. You can map to constants in the configuration file or in a mapper class. Example 10–12 shows part of a `wsrp-user-properties-config.xml` file where a property called `long` is mapped to a constant of type long, which is enclosed in `/L` delimiters.

Example 10–12 Mapping User Properties to Constant Values

```
...
<user-property-map>
    <producer-property-name>map/long</producer-property-name>
    <consumer-property>/L42/L</consumer-property>
</user-property-map>
...
```

Table 10–1 includes the full set of constant delimiters.

Table 10–1 Constant Delimiters

Type	Delimiter	Example
String	"	"Hello World"
Boolean	/B	/Btrue/B

Table 10–1 (Cont.) Constant Delimiters

Type	Delimiter	Example
Long	`/L`	`/L42/L`
Double	`/D`	`/D3.14159/D`
Date	`/T`	`/T1975-09-27T14:38:11-07:00/T`

If you create a mapping class, you can specify constants using the delimiters shown in Table 10–1 or use the constants defined in the `com.bea.wsrp.consumer.userproperty.UserProperty` interface. For details on this interface, refer to the *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*.

10.5 P3P Examples

This section recasts some of the examples given previously in this chapter to show how to use P3P attributes instead of WebLogic Portal user attributes. This section includes the following examples:

This section includes these examples:

- Section 10.5.1, "Example: portlet.xml file with P3P Attributes"
- Section 10.5.2, "Example: Retrieving P3P User Information in a Java Portlet"
- Section 10.5.3, "Example: Retrieving User Information in Other Portlets"

10.5.1 Example: portlet.xml file with P3P Attributes

The portlet.xml file is a standard deployment descriptor for Java portlets. Example 10–13 shows a `portlet.xml` file that includes P3P attributes. For more information on this file, see Section 10.3.1, "Configuring Java Portlets".

P3P attribute names always begin with the prefix `user`, and by convention, a dot (.) separator is used to separate elements of a name (for example: `user.name.given`). For a complete set of names used by Java portlets, refer to the Java Portlet Specification.

Example 10–13 Specifying User Properties in portlet.xml File

```
<portlet-app>
…
    <user-attribute>
        <description>User Given Name</description>
        <name>user.name.given</name>
    </user-attribute>
    <user-attribute>
        <description>User Last Name</description>
        <name>user.name.family</name>
    </user-attribute>
    <user-attribute>
        <description>User eMail</description>
        <name>user.home-info.online.email</name>
    </user-attribute>
    <user-attribute>
        <description>Company Organization</description>
        <name>user.business-info.postal.organization</name>
    </user-attribute>
```

```
...

</portlet-app>
```

10.5.2 Example: Retrieving P3P User Information in a Java Portlet

The example code in Example 10–14 shows how a Map of user information is retrieved from the request in a JSP associated with a Java portlet. Note that standard P3P user property names, such as `user.bdate`, are used in the file.

Example 10–14 Retrieving User Information in a Java Portlet

```
...
Map<String, Object> props;
    PortletRequest portletRequest = (PortletRequest)
request.getAttribute("javax.portlet.request");
    if (portletRequest != null) {
        props = (Map<String, Object>)
portletRequest.getAttribute(PortletRequest.USER_INFO) ;
    } else {
        props = null ;
    }

    if (props == null) {%>
        <p>Empty Profile</p>
    <%} else {%>
       <p><%= props.get("user.bdate") %></p>
       <p><%= props.get("user.business-info.telecom.telephone.intcode") %></p>
    <%}%>
...
```

10.5.3 Example: Retrieving User Information in Other Portlets

The code excerpt in Example 10–15 shows how P3P properties are retrieved in a portlet's JSP file using the P13N tag `<profile:getProperty>`. WebLogic Portal recognizes the constant `com.bea.wsrp.consumer.userproperty.UserProperty.P3P_PROPERTY_SET_NAME` to be the set of standard P3P user properties.

Example 10–15 Retrieving P3P Values in a non-Java Portlet

```
<%@ page import = "com.bea.wsrp.consumer.userproperty.UserProperty" %>
...
<%
    if (request.getUserPrincipal() != null) {
  %>
      <profile:getProfile profileKey="<%= request.getUserPrincipal().getName() %>"
     />
  <%
  } else { %>
      <profile:getProfile profileKey="anonymous" groupOnly="true" />
  <%
  }
  %>

  <tr>
    <td>Date</td>
    <td id="wsrp_date"><profile:getProperty propertySet=
```

```
    "<%= UserProperty.P3P_PROPERTY_SET_NAME %>" propertyName="bdate"/></td>

  </tr>
  <tr>
    <td>Int Code</td>
    <td id="wsrp_int_code"><profile:getProperty propertySet=
    "<%= UserProperty.P3P_PROPERTY_SET_NAME %>" propertyName=
    "businessInfo/telecom/telephone/intcode"/></td>
  </tr>
...
```

11 Consumer Entitlement

Consumer entitlement allows producers to decide which portlets to offer to consumers based on registration properties.

This chapter includes these topics:

11.1 Introduction

WSRP allows consumers to pass information to producers during registration. Through the User Management features of WebLogic Portal, you can create roles based on this registration information. The roles, in turn, can be used to entitle specific portlets for specific consumers. This feature allows producers to control which portlets are offered to specific consumers.

To entitle consumers based on registration properties:

1. Define one or more application-defined property sets using Oracle Enterprise Pack for Eclipse. See Section 11.2.1, "Creating an Application Property Set".
2. Modify the `wsrp-producer-config.xml` configuration file for each portal web application on the producer. See Section 11.2.2, "Editing the Producer Configuration File".
3. Define user entitlements based on the application-defined property set(s). See Section 11.2.3, "Defining Consumer Entitlements".

No configuration is required by consumers. All of the configuration takes place on the producer. Once the producer is properly configured, required registration information is sent to the consumer in response to a service description request. The consumer simply prompts the user to enter the registration information requested by the producer through either the WebLogic Portal Administration Console or Oracle Enterprise Pack for Eclipse.

A typical use case for consumer entitlements is to provide one consumer access to a set of portlets and another consumer access to another set of portlets. For example, suppose your company has several partners. The administrator of the producer could create a registration property with a set of unique values. The administrator could then give each partner their own unique registration property value. When partners register the producer, they are required to enter their value as a registration property, which entitles them to receive a specific set of portlets.

11.2 Configuring a Producer

This section explains how to configure a producer to entitle consumers based on registration properties.

> **Tip:** You can only define consumer entitlements in a complex producer. For information on complex producers, see Section 3.4, "Understanding Producers and Consumers".

The basic steps include:

- Section 11.2.1, "Creating an Application Property Set"
- Section 11.2.2, "Editing the Producer Configuration File"
- Section 11.2.3, "Defining Consumer Entitlements"

11.2.1 Creating an Application Property Set

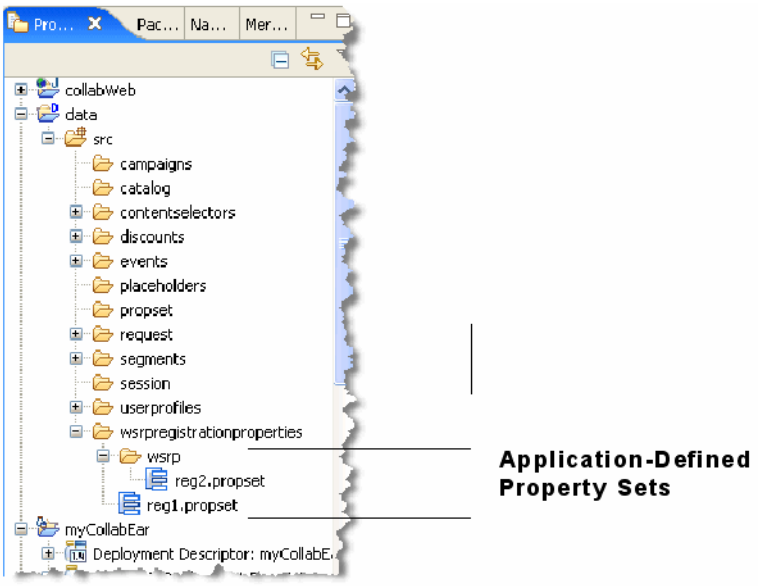
The first step in creating entitlements for consumers based on registration properties is to create one or more Application-Defined Property Sets using Oracle Enterprise Pack for Eclipse. These property sets are used to specify the values a consumer must supply to a producer at registration time.

> **Tip:** Application-Defined Property Sets must be created in a datasync project. To start the datasync creation wizard in Oracle Enterprise Pack for Eclipse, select **File** > **New** > **Other** > **WebLogic Portal** > **Datasync**. For detailed information on creating Application-Defined Property Sets, see the *Oracle Fusion Middleware Interaction Management Guide for Oracle WebLogic Portal*.

For example, if you create a property set containing a set of identification keywords. When the producer receives the registration information from the consumer, it can evaluate the keyword it receives and, based on a visitor entitlement, return a specific set of portlets. If the user registers with a different keyword, a different set of portlets could be returned.

The property sets you create appear in Oracle Enterprise Pack for Eclipse in the datasync project's src/propset folder. Figure 11–1 shows a sample `propset` folder containing two property sets.

Figure 11–1 Application-Defined Property Sets



11.2.2 Editing the Producer Configuration File

After you create property sets containing consumer registration properties and optional default values, you need to make the producer aware of these property sets. To do this, you must edit the configuration file `wsrp-producer-config.xml`. Only the property sets listed in this configuration file are sent to consumers for registration. This configuration file includes a `<registration>` element. This element includes the `<property-uri>` element, which specifies paths to each of the property sets you defined for that producer.

By default, a producer includes a path `/wsrpregistrationproperties`. You can either put `.propset` files in that directory or create other directories as needed and list them in the `<property-uri>` element.

> **Tip:** By default, the wsrp-producer-config.xml file is stored in a J2EE Shared Library. To edit the file, you must first copy it from the J2EE Shared Library to your workspace. To do this, switch to the Merged Projects view in Oracle Enterprise Pack for Eclipse. In the WEB-INF directory of the producer web application, right-click the wsrp-producer-config.xml file (it appears in an italic font) and select Copy to Project. The configuration file is then copied from the J2EE Shared Library to your file system, where you can edit it. Any local changes you make to the file take precedence over the J2EE Shared Library version.

Example 11–1 shows a sample `<registration>` element in a `wsrp-producer-config.xml` file. A directory called `/wsrpregistrationproperties` is created and configured by default in the `wsrp-producer-config.xml` file. Any property sets placed in this directory are automatically sent to the consumer as registration properties. In addition, all property sets in the `/wsrp` directory will be sent to the consumer as registration properties. The paths specified in the `<property-uri>` element are relative to the `META-INF/data` directory of each producer web application. Note that the directory name `/wsrp` is an example only; you can create property sets under any directory name you choose. If you do not provide specific property sets using `<property-uri>` elements,

WebLogic Portal imports all Application-Defined Property Sets for registration properties.

Example 11–1 Registration Element

```
<service-config>
        <registration required="true" secure="false">
     <property-uri>/wsrpregistrationproperties</property-uri>
           <property-uri>/wsrp</property-uri>
        </registration>
        <service-description secure="false" supports-method-get="true"/>
        <markup secure="false" rewrite-urls="true" transport="string"/>
        <portlet-management required="true" secure="false"/>
    </service-config>
```

The `isStrict` keyword is a `<registration>` keyword that causes registration to fail in a specific case, as specified in Table 11–3.

Table 11–1 isStrict Keyword

Value of isStrict	Explanation
`isStrict = true`	Causes registration to fail if both of these are true: ■ a consumer provides a property value for a registration property that has a restricted set of values defined *and* ■ the provided value(s) do not fall within the restricted set of values.
`isStrict = false`	(Default) If the consumer sends a value that lies outside of the set of values associated with a registration property set, the user can register, but the registration value(s) are not saved. In this case, entitled portlets that require these values will not be offered to the consumer.

The `isStrict` keyword is part of the `<registration>` element. The `isStrict` keyword is show in bold type in Example 11–2.

Example 11–2 isStrict Keyword

```
<service-config>
   <registration required="true" secure="false" isStrict="true">
      <property-uri>/wsrpregistrationproperties</property-uri>
      <property-uri>/wsrp</property-uri>
</registration>
   </service-config>
```

11.2.3 Defining Consumer Entitlements

After you have created property sets for consumer registration and added them to the `wsrp-producer-config.xml` file, you can create visitor entitlements based on the property sets.

In the WebLogic Portal Administration Console, you can define Role Expressions for consumer registration. Figure 11–2 shows the Role Expressions tab selected for a Visitor Role called role1.

Figure 11–2 Role Expressions Tab

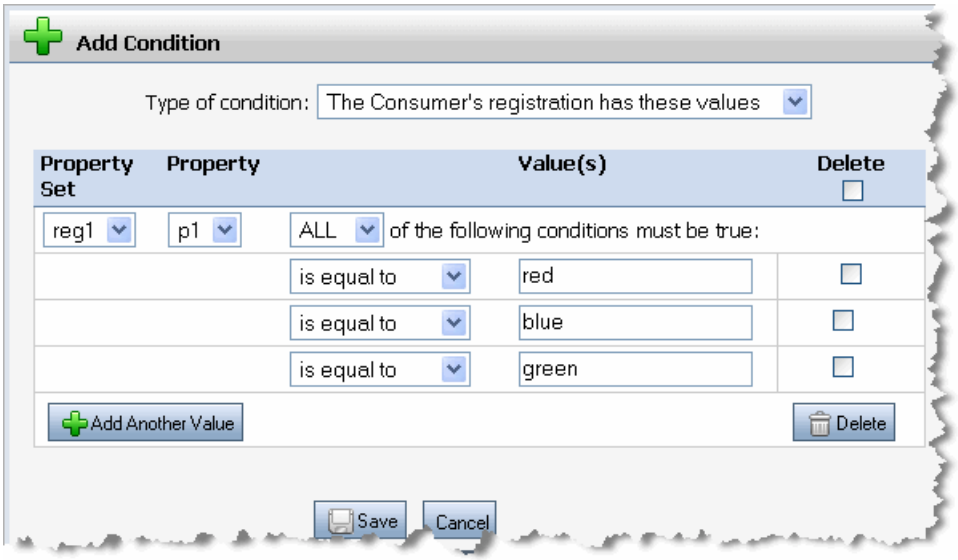
The role expressions are used to dynamically determine whether or not a consumer belongs to a visitor entitlement role. Individual portlets can then be offered based on membership in that role.

To create a visitor entitlement in the Administration Console:

1. Create a visitor role.
2. Define one or more consumer registration expressions in the role.
3. Entitle specific portlets based on the role.

For example, you can define a role with the following Role Expression: If the property p1 equals red, blue, or green, then consumer is considered to be a role member. The producer then returns all portlets that are entitled for that role to the consumer. Figure 11–3 shows the Add Conditions dialog in the WebLogic Portal Administration Console. This dialog is used to define Role Expressions. When creating entitlements for consumer registration, select **The Consumer's registration has these values** from the Type of condition drop-down menu.

Figure 11–3 Setting Registration Properties



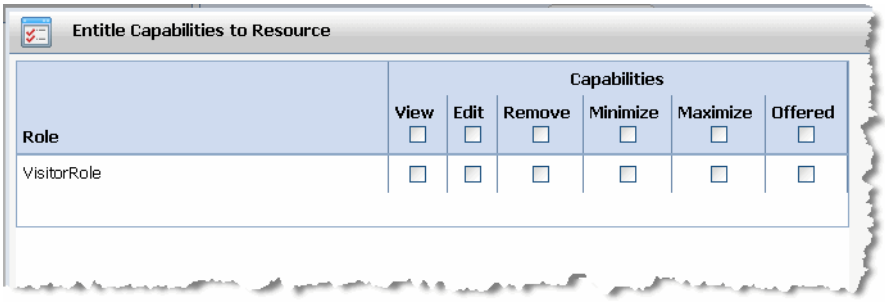
Detailed information on setting up visitor entitlements is beyond the scope of this guide. See the *Oracle Fusion Middleware Security Guide for Oracle WebLogic Portal* for information on this topic.

To entitle a portlet with a consumer registration role:

1. In the Administration Console, select the portlet you want to entitle.
2. Select the Entitlements tab.

3. Click **Add Role**.
4. Use the Add Role dialog to select the role to add to the portlet, and click **Save**.
5. Complete the Entitle Capabilities to Resource dialog (Figure 11–4) to assign capabilities to the role and click **Save**.

Figure 11–4 Entitling Capabilities to Resource Dialog

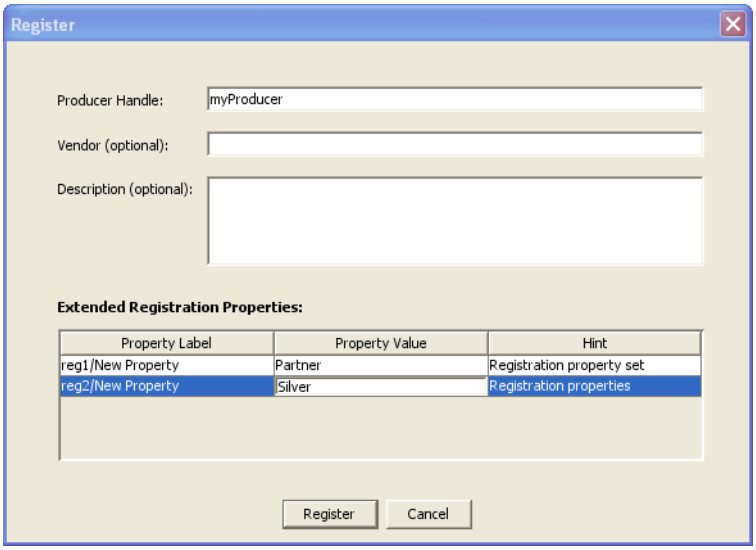


11.3 Registering a Consumer

When you register a consumer with a producer that has been configured to request registration properties, the producer asks the consumer to provide values for those properties. In Oracle Enterprise Pack for Eclipse, the set of extended registration properties and optional default values are added to the Register dialog, which appears when you attempt to create a remote portlet. On each subsequent request by the consumer, the producer retrieves these registration properties and uses them to make entitlement decisions.

Figure 11–5 shows the registration dialog for a producer that requires registration properties. In this example, two properties are requested: reg1 and reg2. The property values entered by the user are sent to the producer. The producer retrieves the values and stores them. On subsequent requests, the producer compares the stored registration values to the registration values it requires. If the registration values are accepted, the producer uses them to determine to which role the consumer belongs. Once the consumer's role is established, the producer returns only entitled portlets to the consumer.

Figure 11–5 Register Dialog



Tip: When you register a producer, the consumer sends a getServiceDescription request to the producer. The producer's response includes the <registration> element. This element includes a list of the types of registration properties the producer requires. The consumer then populates the registration dialog with the appropriate fields. The user fills out these fields and submits them with the registration request. For information on getServiceDescription and other WSRP operations, see Chapter 3, "Federated Portal Architecture."

11.4 Modifying Registration Properties

Using the WebLogic Portal Administration Console, you can modify the registration properties for a producer that has already been registered with a consumer. When the consumer re-registers the producer, some portlets that were previously in use might not be available or some additional portlets might be available to the consumer. For detailed information on modifying the registration properties for a producer using the Administration Console, see Section 20.2, "Modifying Producer Registration Properties".

12 Transferring Custom Data

WebLogic Portal supports a relatively simple technique for passing custom data between consumers and producers. A set of interfaces is provided that let you attach arbitrary data to request and response objects. This chapter explains how to use these interfaces to achieve custom data transfer and includes detailed examples.

This chapter includes the following topics:

12.1 What is Custom Data Transfer?

Custom data transfer allows portlet developers to exchange arbitrary data between producers and consumers. The primary use cases for custom data transfer are:

- To send and receive data that WebLogic Portal does not usually send or receive.
- To send and receive data that the WSRP protocol does not allow.

> **Note:** It is recommend that you use this technique only after trying other techniques for data transfer. The preferred technique for transferring data between producers and consumers is to use custom events. For more information, see Section 7.4, "Data Transfer with Custom Events".

Some example use cases for custom data transfer include:

- You are a portal developer building a portal with a set of location-sensitive portlets deployed on one or more producers. You would like to supply a zip code to each of these portlets in a request so that each portlet can use this zip code to generate location-aware markup.
- You want to send arbitrary data such as theme or style information or user profile data to portlets.

Custom data transfer allows you to easily resolve these situations and many others like them. The technique for using custom data transfer is straightforward, and involves these primary tasks:

1. Create one or more "holder" classes that implement the interfaces listed in the following section, "Section 12.2, "Custom Data Transfer Interfaces"." A serializable default implementation of the interfaces, called SimpleStateHolder is provided with WebLogic Portal.
2. Place a serializable holder object in a request or response object, as appropriate. For example, in a consumer application, you can set a holder object as a request parameter and retrieve it in the producer application. See Section 12.3.1, "Custom Data Transfer with a Complex Producer" for a detailed example of this technique.

Both simple producers and complex producers can take advantage of this feature.

12.2 Custom Data Transfer Interfaces

The following interfaces enable the transfer of data between producers and consumers. To perform custom data transfer, implementations of these interfaces must be deployed on both the consumer and producer.

Section 12.3, "Performing Custom Data Transfer" includes a detailed example demonstrating how to use these interfaces. For more information on these interfaces, refer to their *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*descriptions.

Note: These interfaces are not supported for events and render dependencies requests.

- com.bea.wsrp.ext.holders.InteractionRequestState

  Allows the consumer to send some arbitrary data to the producer when an interaction (such as a form submission) occurs.

- com.bea.wsrp.ext.holders.InteractionResponseState

  Allows the producer to return some arbitrary data to the consumer after an interaction occurs.

- com.bea.wsrp.ext.holders.MarkupRequestState

  Allows the consumer to send some arbitrary data to the producer when a portlet is being refreshed.

- com.bea.wsrp.ext.holders.MarkupResponseState

  Allows the producer to return some arbitrary data to the producer after portlet is rendered.

- com.bea.wsrp.ext.holders.XmlPayload

  Transfers XML data between consumers and producers. You can place an instance of this class directly in request and response objects. For more information, see Section 12.4, "Transferring XML Data".

  Tip: If you do not want to create your own implementations of these interfaces, the serializable com.bea.wsrp.ext.holders.SimpleStateHolder class provides a default implementation. The examples in this chapter use SimpleStateHolder to pass custom data.

12.3 Performing Custom Data Transfer

This section presents examples that illustrate how to use custom data transfer between consumers and producers. Both examples use the serializable com.bea.wsrp.ext.holders.SimpleStateHolder class, which implements the five interfaces listed previously in Section 12.2, "Custom Data Transfer Interfaces". This class provides a default implementation of the above interfaces that lets you exchange simple name-value pairs of data.

The examples include:

- Section 12.3.1, "Custom Data Transfer with a Complex Producer"

  This example demonstrates custom data transfer between a consumer and a complex producer.

- Section 12.3.2, "Custom Data Transfer in a Simple Producer"

  This example demonstrates custom data transfer between a consumer and a simple producer.

12.3.1 Custom Data Transfer with a Complex Producer

This example explains how to transfer data from a consumer to a complex producer. For information on complex producers, see Section 3.4.2, "WebLogic Portal Producers".

12.3.1.1 Example Overview

In this example, a backing file in the consumer application packages arbitrary data in a com.bea.wsrp.ext.holders.SimpleStateHolder object. This object is attached to a request using the setAttribute() method. The producer retrieves the data from the request and places it in a JSP page. The modified page is then displayed by the consumer application.

The example consists of these steps:

1. Section 12.3.1.2, "Setting Up the Example"
2. Section 12.3.1.3, "Creating the Producer JSP and Portlet"
3. Section 12.3.1.4, "Federating zipTest.portlet to the Consumer"
4. Section 12.3.1.5, "Creating a Backing File"
5. Section 12.3.1.6, "Testing the Consumer Application"

12.3.1.2 Setting Up the Example

If you want to try the example discussed in this chapter, you need to run Oracle Enterprise Pack for Eclipse and perform the prerequisite tasks listed in Table 12–1. For detailed information on performing these basic setup tasks, see "Setting Up Your Portal Development Environment" in *Oracle Fusion Middleware Tutorials for Oracle WebLogic Portal*.

Table 12–1 Prerequisite Tasks

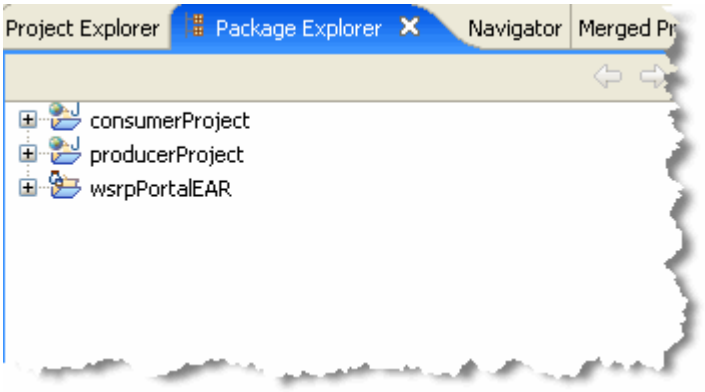
Task	Recommended Name
Create a Portal domain.	`wsrpPortalDomain`
Create a Portal EAR Project.	`wsrpPortalEAR`
Create an Oracle WebLogic Server v10.x.	N/A
Associate the EAR project with the server.	N/A

Table 12–1 (Cont.) Prerequisite Tasks

Task	Recommended Name
Create a Portal Web Project and add it to the EAR.	`consumerProject`
Create a second Portal Web Project and add it to the EAR.	`producerProject`

Figure 12–1 shows the Package Explorer after the prerequisite tasks have been completed.

Figure 12–1 Package Explorer After Prerequisite Tasks are Completed



We also assume that you know how to view and edit portlet properties in the Properties view. For information, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

12.3.1.3 Creating the Producer JSP and Portlet

With the example environment in place, create a JSP file on the producer and a portlet to surface that file. Code placed in the JSP file retrieves a SimpleStateHolder object from the request, retrieves its data payload, and displays the data.

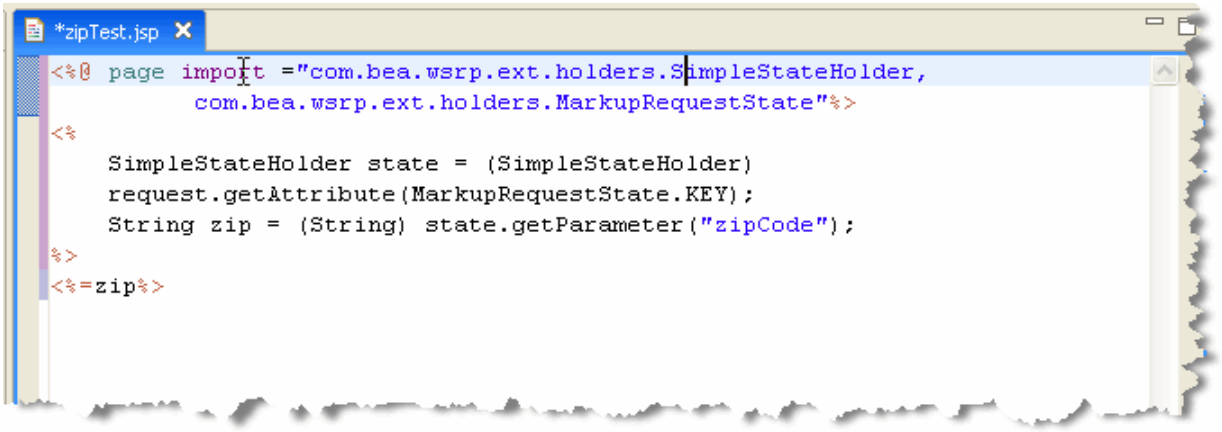
1. Be sure you have set up the example environment as explained previously in Section 12.3.1.2, "Setting Up the Example".
2. Right-click **producerProject/WebContent** in the Package Explorer and select **New** > **JSP**. The New JavaServer Page dialog appears.
3. In the dialog, enter `zipTest.jsp` in the File name field, and click **Finish**.
4. Replace the entire contents of the JSP source file with the code in Example 12–1.

Example 12–1 Code to Get State from the Request

```
<%@ page import ="com.bea.wsrp.ext.holders.SimpleStateHolder,
         com.bea.wsrp.ext.holders.MarkupRequestState"%>
<%
    SimpleStateHolder state = (SimpleStateHolder)
    request.getAttribute(MarkupRequestState.KEY);
    String zip = (String) state.getParameter("zipCode");
%>
<%=zip%>
```

Figure 12–2 shows the editor with the new source code.

Figure 12–2 New JSP Source for zipTest.jsp



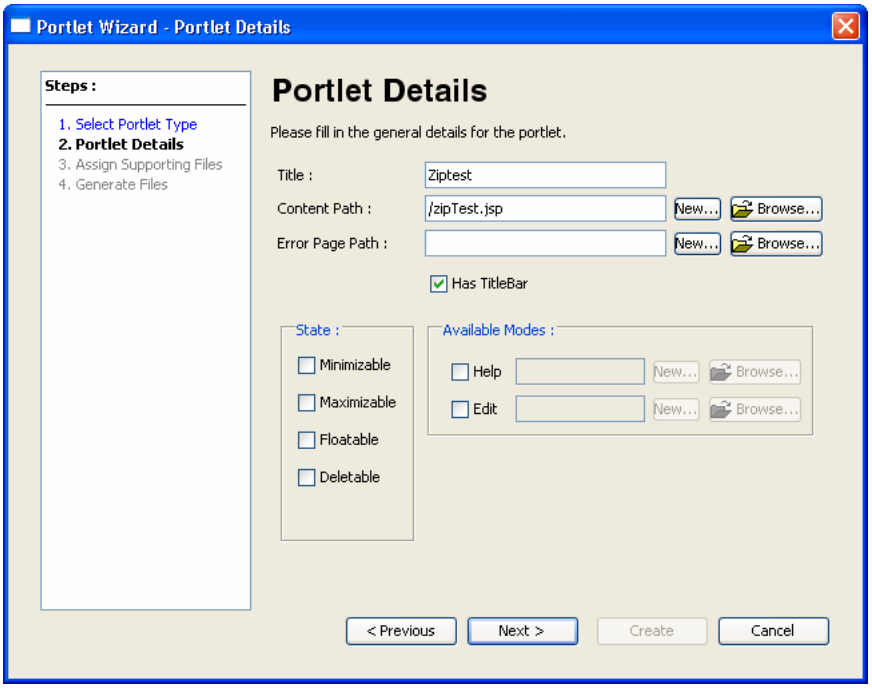
5. Save the file.

   Tip: Later in this example, you will add a backing file to the proxy portlet in the consumer web application. This backing file creates the SimpleStateHolder object, adds some data to it, and puts the object into the request that is sent from the consumer to the producer. For more information on SimpleStateHolder, refer to its description in *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*.

6. In the Package Explorer view, open the `producerProject/WebContent` folder. Right-click **zipTest.jsp** in the WebContent folder and select **Generate Portlet**.

   The Portlet Details dialog box appears. Note that `zipTest.jsp` already appears in the Content Path field, as shown in Figure 12–3.

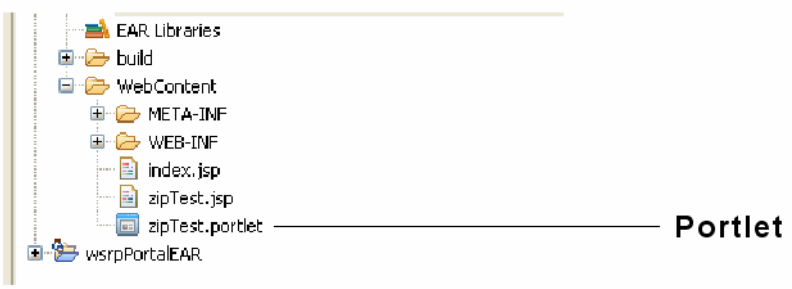
Figure 12–3 Portlet Details with zipTest.jsp Included



7. In the State checkbox, select **Minimizable** and **Maximizable**, then click **Next** and then click **Create**.

   The portlet `zipTest.portlet` appears in the Package Explorer, as shown in Figure 12–4.

Figure 12–4 New JSP Portlet

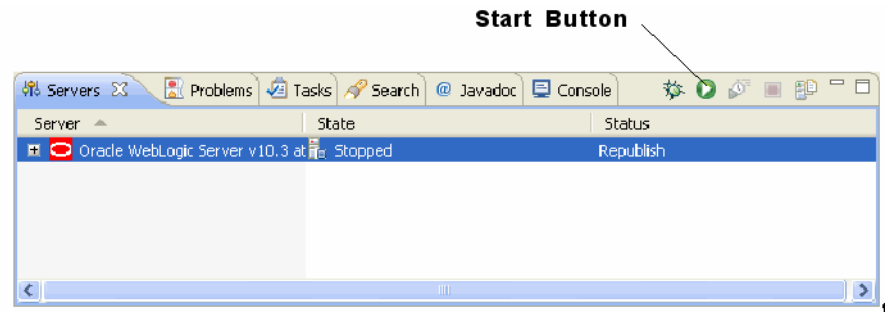


12.3.1.4 Federating zipTest.portlet to the Consumer

Next, create a remote portlet in the consumer application to surface in `zipTest.portlet` from the producer:

1. Be sure that WebLogic Server is running. If not, select the Servers tab. Make sure the **Oracle WebLogic Server v10.x** is selected, and click the **Start** button, as shown in Figure 12–5.

Figure 12–5 Click the Start Button to Start the Server

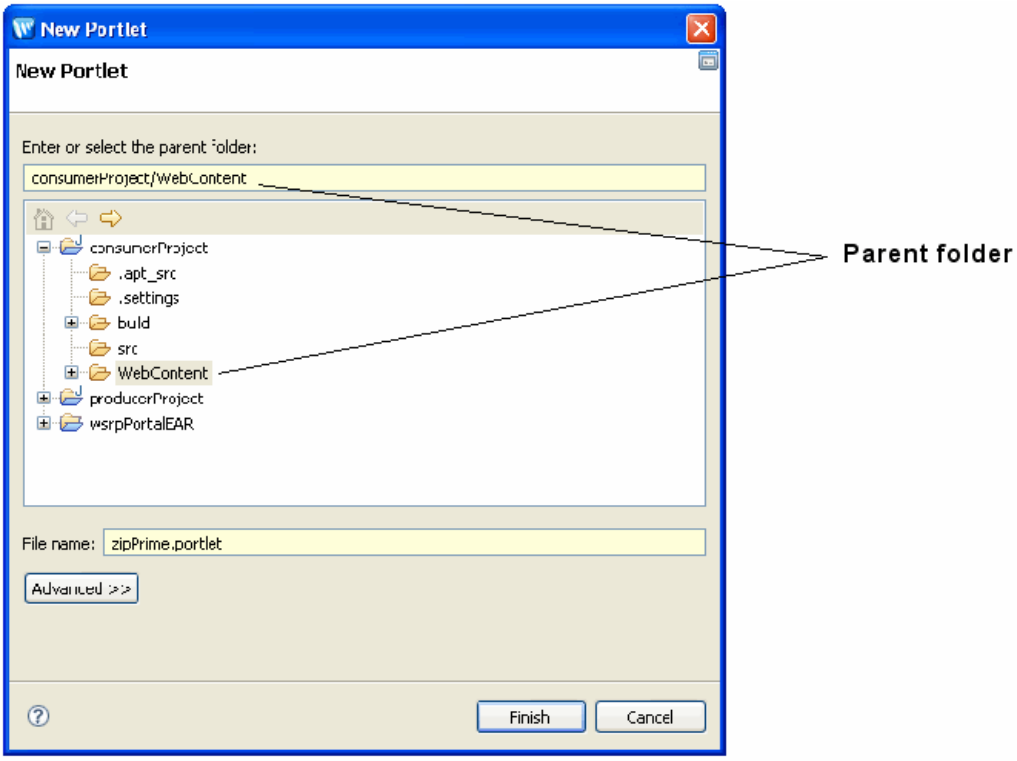


2. In the Package Explorer, open the consumerProject folder.
3. Right-click the **WebContent** folder, and select **New** > **Portlet**.

   Tip: The Portlet selection only appears on the New menu if you are using the Portal perspective. Switch to the Portal perspective if Portlet does not appear on the menu.

   The New Portlet dialog box appears, as shown in Figure 12–6.

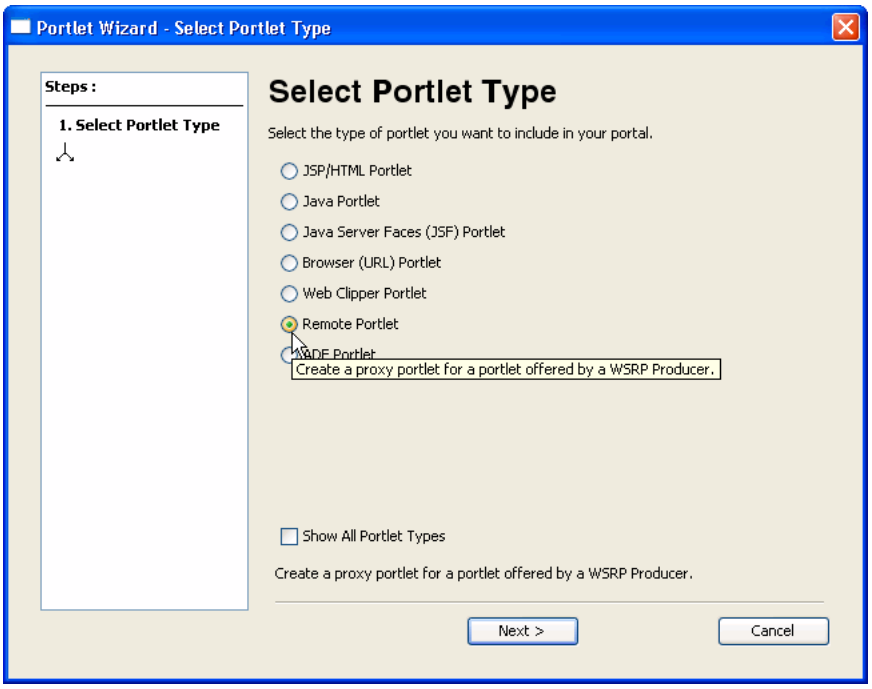
Figure 12–6 New Portlet Dialog



4. In the New Portlet dialog, select **WebContent** as the parent folder, enter `zipPrime.portlet` in the File name field, and click **Next**.

   The Select Portlet Type dialog box appears as shown in Figure 12–7.

Figure 12–7 Select Portlet Type Dialog

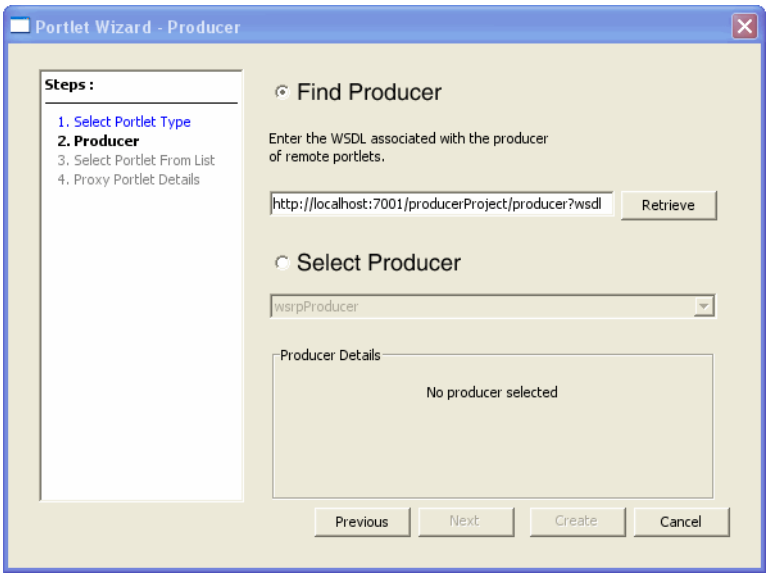


5. Select **Remote Portlet** and click **Next**. The Portlet Wizard – Producer dialog box appears.
6. In the Portlet Wizard – Producer dialog, select **Find Producer** and, in the field provided, enter the following WSDL URL, as shown in Figure 12–8:

   ```
   http://localhost:7001/producerProject/producer?wsdl
   ```

   Tip: Of course, the host name localhost is only appropriate if the producer is running on the same server as the consumer. We co-located the consumer and producer to simplify the presentation of this example. Typically, producers and consumers do not run in the same server.

Figure 12–8 The WSDL URL



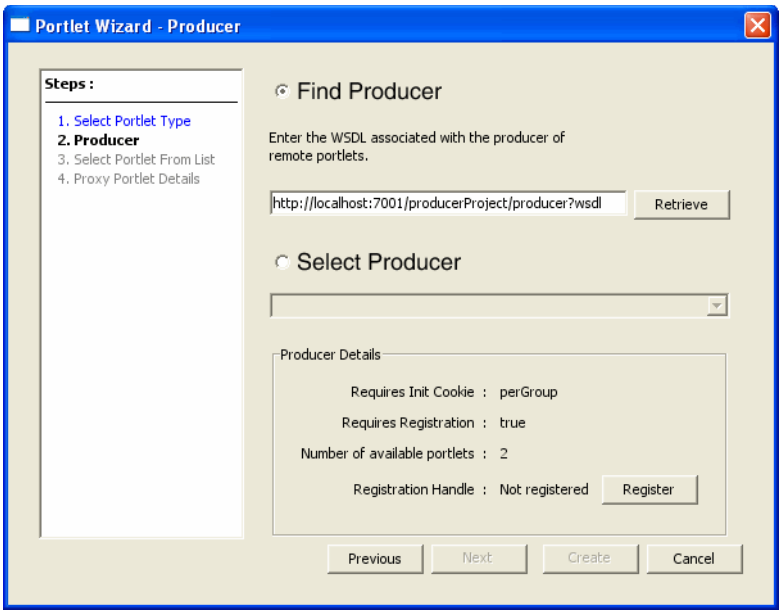
7. After entering the WSDL URL, click **Retrieve**.

   Tip: WSDL stands for Web Services Description Language and is used to describe the services offered by a producer. For more information, see Section 3.4.3, "WebLogic Portal Consumers".

   After a few moments, the Portlet Wizard – Producer dialog box refreshes, and registration information appears in the Producer Details panel, as shown in Figure 12–9.

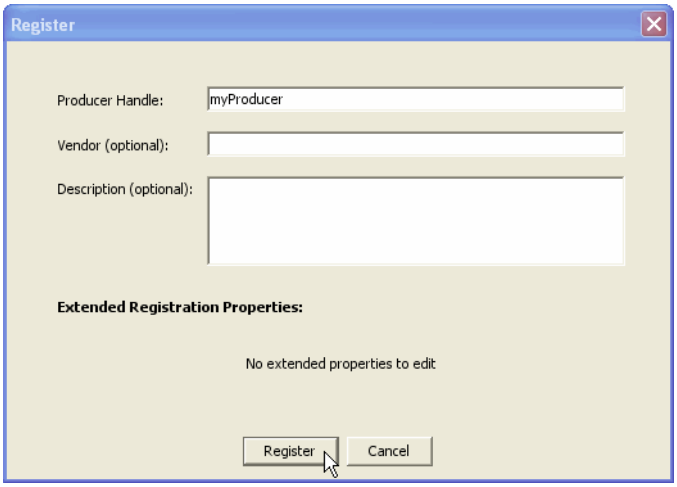
   Tip: Registration is an optional feature described in the WSRP specification. A WebLogic Portal complex producer implements this option and, therefore, requires consumers to register before discovering and interacting with portlets offered by the producer. See Section 3.4.2.2, "Complex Producers" for more information.

Figure 12–9 Producer Retrieved



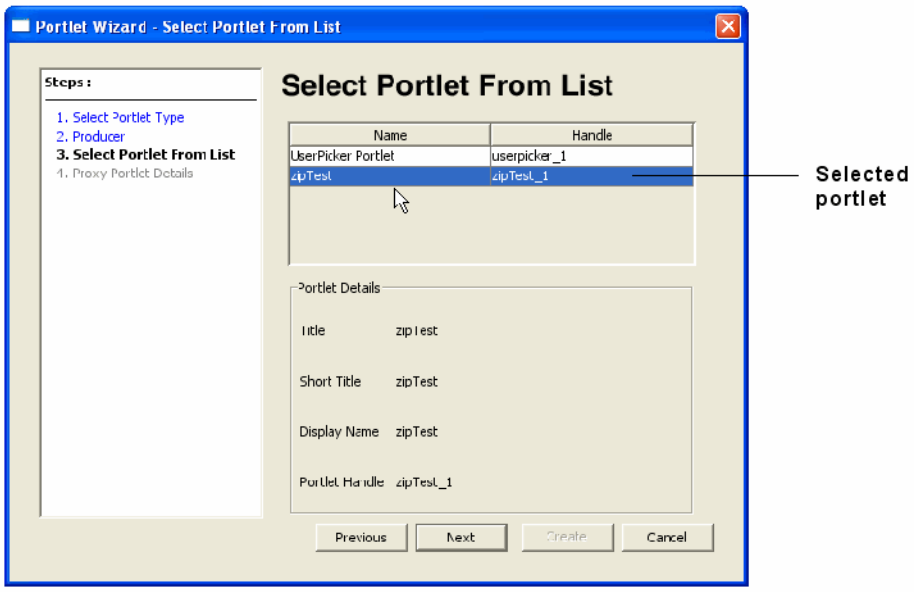
8. Click **Register**. The Register dialog appears.
9. In the Register dialog, enter `myProducer` in the Producer Handle field, as shown in Figure 12–10, and click **Register**. The handle is stored on the consumer and is used to identify the producer.

Figure 12–10 The Register Dialog



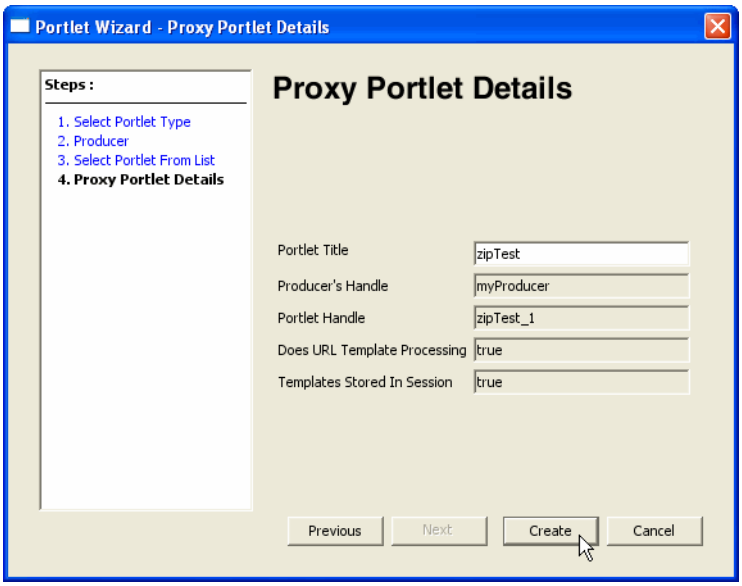
10. In the Portlet Wizard – Producer dialog, click **Next**. The Select Portlet from List dialog box appears.
11. From the portlet list, select **zipTest**, as shown in Figure 12–11.

Figure 12–11 Select Portlet from List Dialog Box



12. Click **Next**. The Proxy Portlet Details dialog box appears, as shown in Figure 12–12.

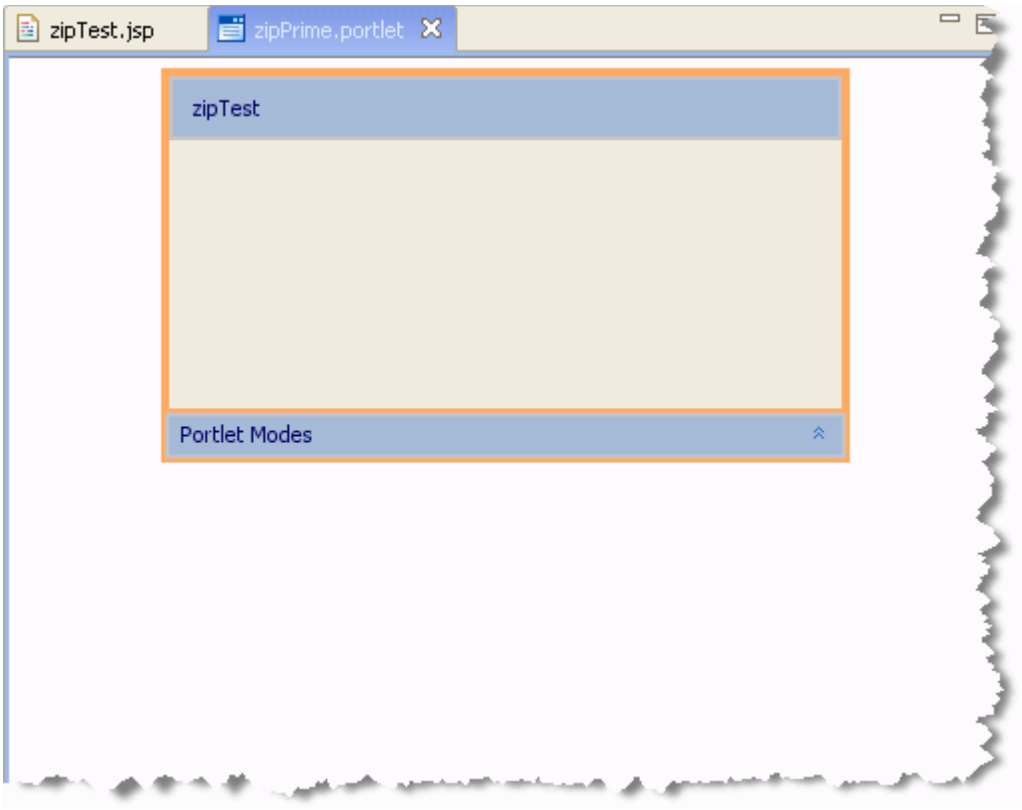
Figure 12–12 Proxy Portlet Details Dialog Box



13. Click **Create**.

    The new portlet appears in the Editor, as shown in Figure 12–13.

Figure 12–13 New Remote Portlet zipPrime.portlet in the Editor



12.3.1.5 Creating a Backing File

In this step, you will create a backing file called `CustomDataBacking.java` in the consumer application. Then, you will attach the backing file to the remote portlet you created previously, `zipPrime.portlet`.

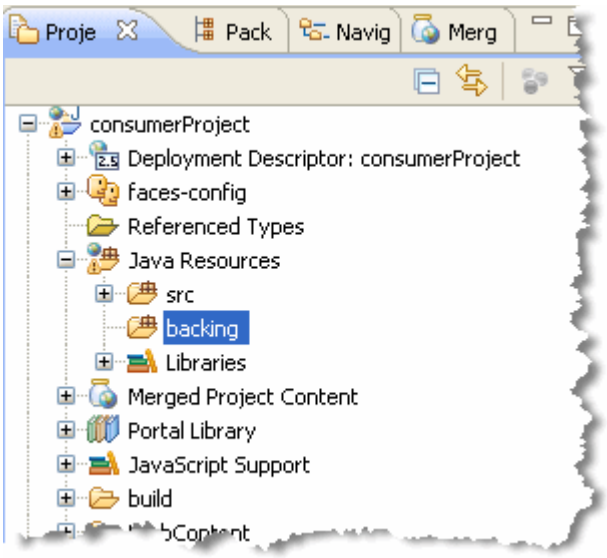
> **Tip:** A backing file is a Java class that adds functionality to a portlet. For information on backing files, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

1. In the Package Explorer tree, open the consumerProject folder, expand the Java Resources node, and right-click the src folder, and create a new Source Folder called `backing`.

   The `src/backing` folder appears in the Package Explorer, as shown in Figure 12–14.

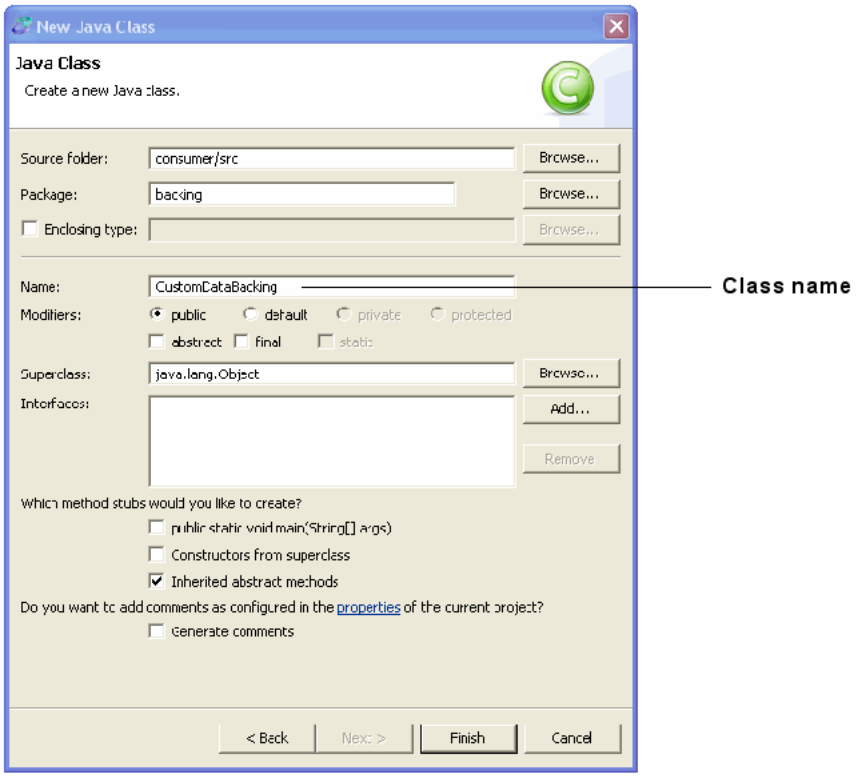
   > **Tip:** Alternatively, instead of a folder, you can create a Java package.

Figure 12–14 backing Folder



2. Right-click the **backing** folder and select **New** > **Class**. The New Java Class dialog appears, as shown in Figure 12–15.

Figure 12–15 New Java Class Dialog



3. In the Name field, enter `CustomDataBacking` and click **Finish**. The new Java source file appears in the editor.
4. Replace the entire contents of the Java source file with the code in Example 12–2.

Example 12–2 Adding an Instance of SimpleStateHolder

```
package backing;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking;
import com.bea.wsrp.ext.holders.MarkupRequestState;
import com.bea.wsrp.ext.holders.SimpleStateHolder;

public class CustomDataBacking extends AbstractJspBacking
{
   private static final long serialVersionUID = 1L;
   public boolean preRender(HttpServletRequest request,
      HttpServletResponse response)
   {
      SimpleStateHolder state = new SimpleStateHolder();
      state.addParameter("zipCode", "80501");
      request.setAttribute(MarkupRequestState.KEY, state);
      return true;
   }
}
```

5. Save the file. The completed backing file is shown in Figure 12–16.

Figure 12–16 CustomDataBacking.java in the Editor

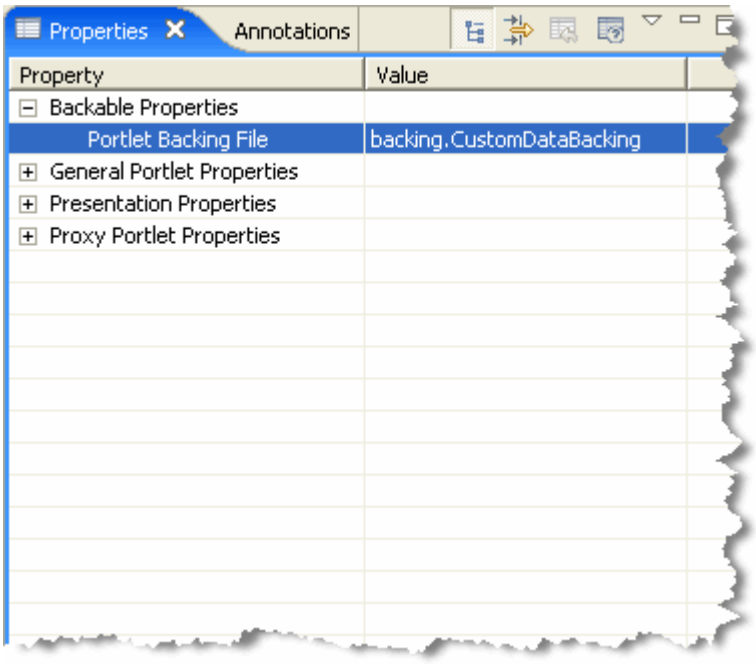
Tip: The backing file implements the AbstractJspBacking.preRender() method. This method is called *before* the request is sent to the producer. The implementation attaches a SimpleStateHolder object containing custom data to the request. This object will be retrieved on the producer where the data is extracted and displayed in the remote portlet.

6. Double-click `zipPrime.portlet` to display it in the editor.
7. Add the backing file to `zipPrime.portlet`. To do this, enter the full classname of the backing file in the Backing File field in the Properties view:

   `backing.CustomDataBacking`

Figure 12–17 shows the class name after it has been added.

Figure 12–17 Adding a Backing File



12.3.1.6 Testing the Consumer Application

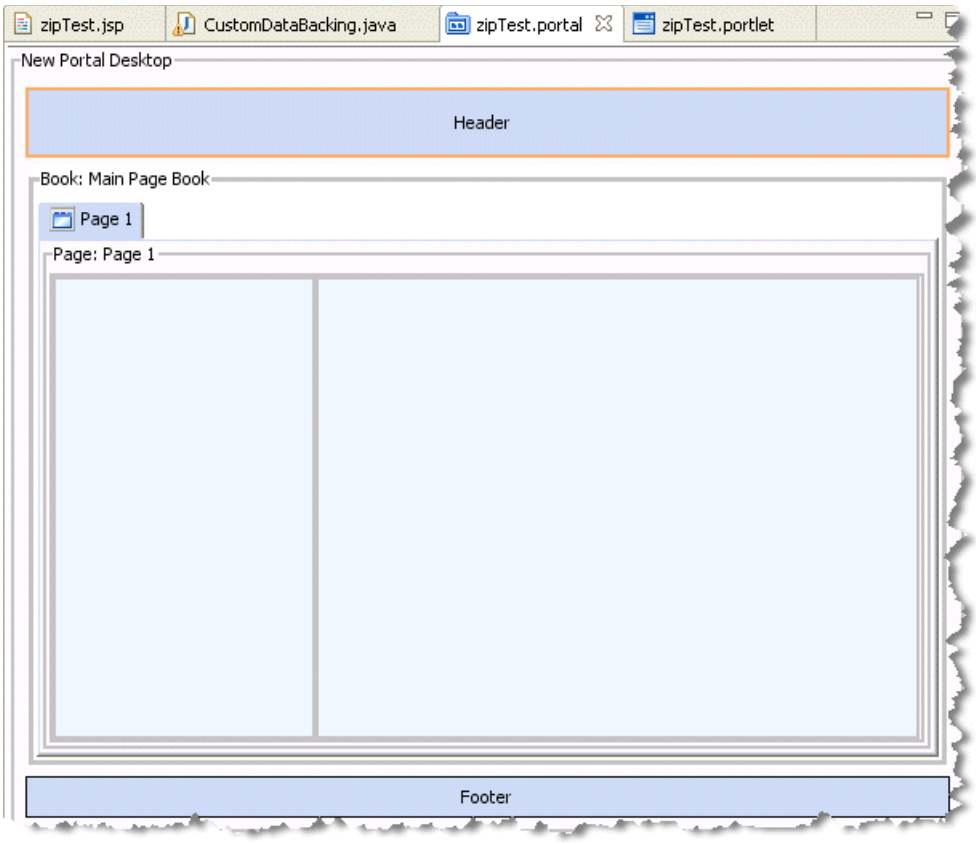
With the consumer application components in place, you can now test the configuration. If the test is successful, the zip code 80501, provided by the backing file, will appear in the remote portlet when it is rendered.

To test the application:

1. In the Package Explorer, right-click **consumerProject/WebContent** and select **New > Portal**. The New Portal dialog appears.
2. In the File name field, enter `zipTest.portal` and click **Finish**.

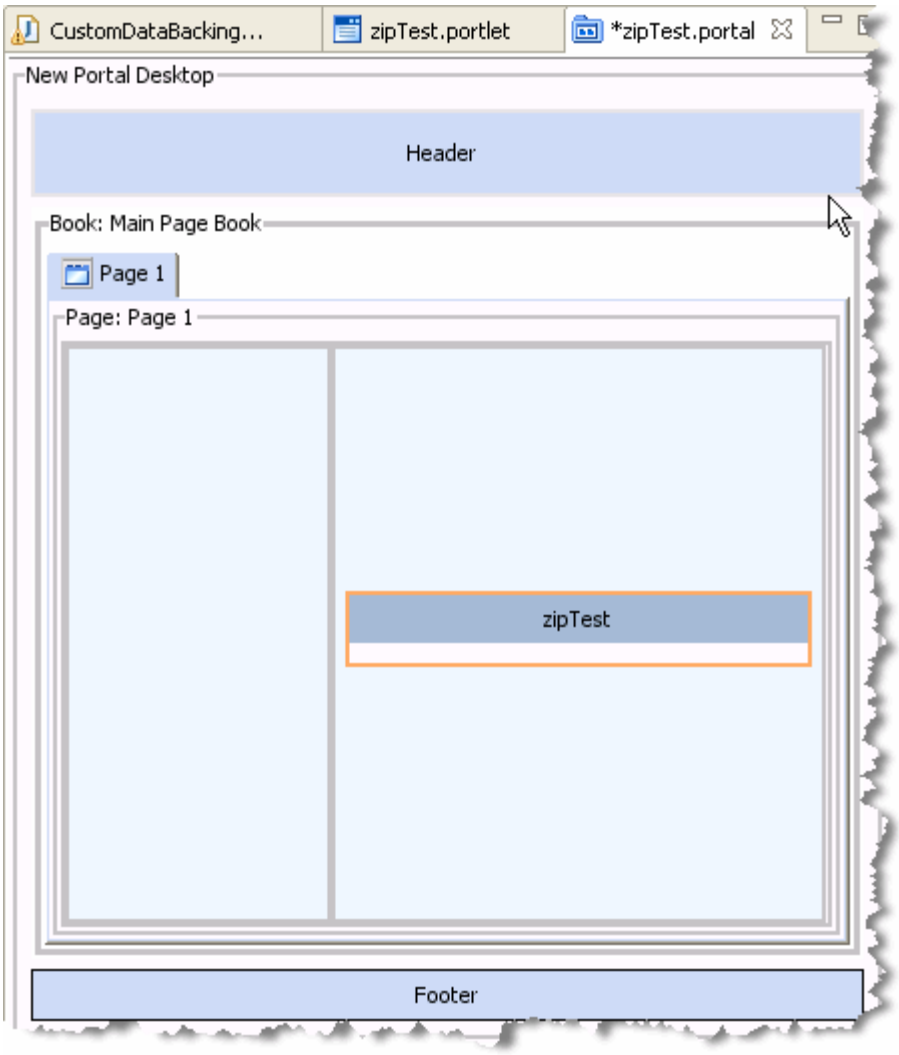
   The portal is created and appears in the editor, as shown in Figure 12–18.

Figure 12–18 zipTest.portal in the Editor



3. Drag the remote portlet **zipPrime.portlet** from Package Explorer view into the portal. (You can place it in either the left or right column; in Figure 12–19, it is in the right-hand column).

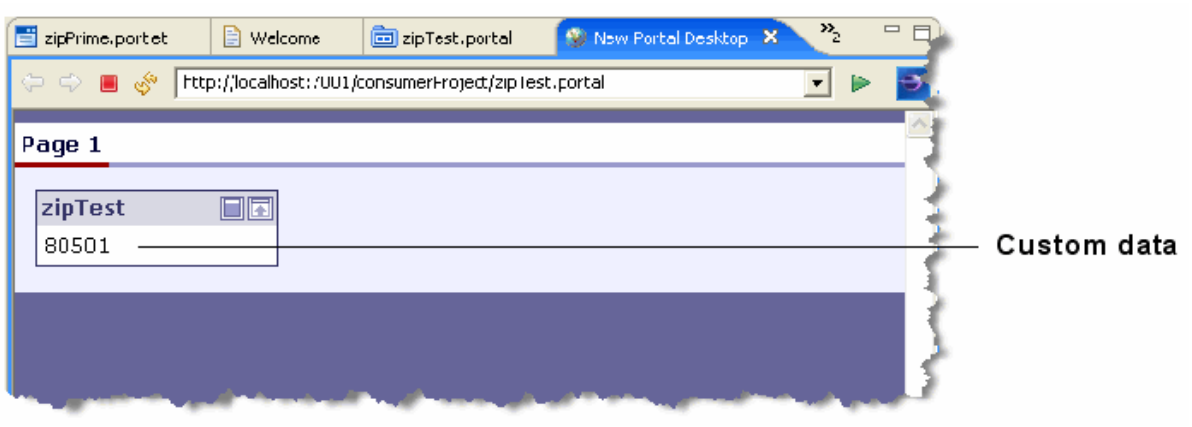
Figure 12–19 zipTest.portlet Added to zipTest.portal



4. Save the portal.
5. Run the portal. To do this, right-click **zipTest.portal** in the Package Explorer and select **Run As** > **Run on Server**.
6. In the Run On Server – Define a New Server dialog, click **Finish**.

   The portal appears in the Oracle Enterprise Pack for Eclipse browser. The custom data sent from the consumer displays in the portlet, as shown in Figure 12–20.

Figure 12–20 zipTest.portal Successfully Rendered



12.3.2 Custom Data Transfer in a Simple Producer

The previous section, Section 12.3.1, "Custom Data Transfer with a Complex Producer", explains how to transfer data between a WebLogic Portal consumer application and a complex producer running in a WebLogic Portal domain. You can also transfer data between a WebLogic Portal consumer and a simple producer running in a WebLogic Server domain.

> **Tip:** For a detailed discussion of complex and simple producers, see Section 3.4.2, "WebLogic Portal Producers".

To use custom data transfer with a simple producer:

1. Properly configure a simple producer running in a WebLogic Server domain. The procedure for doing this is explained in Chapter 8, "Configuring a WebLogic Server Producer."
2. Use the Custom Data Transfer interfaces listed in Section 12.2, "Custom Data Transfer Interfaces" to set and retrieve data in request and response objects. Follow the same basic procedure described for complex producers in Section 12.3.1, "Custom Data Transfer with a Complex Producer".

12.4 Transferring XML Data

Use an implementation of the com.bea.wsrp.ext.holders.XmlPayload interface to transfer XML data (objects of type Element) between consumers and producers. You can place an instance of this class directly in request and response objects. For more information on the XmlPayload interface, refer to its description in *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*.

Example 12–3 shows sample code that uses XmlPayload.

Example 12–3 XmlPayload Example

```
//-- Create an Element object to send.
Element xml = ...

XmlPayload payload = new XmlPayload(xml);
httpRequest.setAttribute(MarkupRequestState.KEY, payload);
```

12.5 Deploying Your Own Interface Implementations

This section discusses guidelines for implementing the custom data transfer interfaces listed in Section 12.2, "Custom Data Transfer Interfaces".

- Section 12.5.1, "General Guidelines"
- Section 12.5.2, "Implementation Rules"

12.5.1 General Guidelines

- The implementation must be serializable.
- The same class version of the implementation must be deployed on both the producer and consumer. If the versions are different, the implementations must make sure to have the same `serialVersionUID` for all versions.
- Sending large amounts of data may have performance implications.

> **Tip:** The com.bea.wsrp.ext.holders.SimpleStateHolder class provides a default implementation of the four data transfer interfaces. This class lets you exchange simple name-value pairs of data. For detailed information on the methods of this class, refer to its description in *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*.

12.5.2 Implementation Rules

Whether a consumer or producer can send custom data depends on the type of request. These rules apply:

- Consumers can always send `InteractionRequestState`. There are no exceptions.
- Producers can always return `InteractionResponseState`. There are no exceptions.
- Consumers can send `MarkupRequestState` only when there is a need to refresh the portlet. For example, if caching is enabled on the remote portlet, consumer may not always send a request to the producer to generate markup.
- Consumers cannot return `MarkupResponseState` if any the following options are enabled:
  - Returning markup as an attachment
  - Local proxy

    In both the cases, the producer invokes the portlet (typically JSPs) after creating the SOAP response, which is too late to update the SOAP response.

13

WSRP Interoperability with Oracle WebCenter Portal and Oracle Portal

This chapter discusses techniques and best practices for achieving WSRP interoperability between WebLogic Portal and Oracle WebCenter Portal and Oracle Portal.

This chapter includes the following sections:

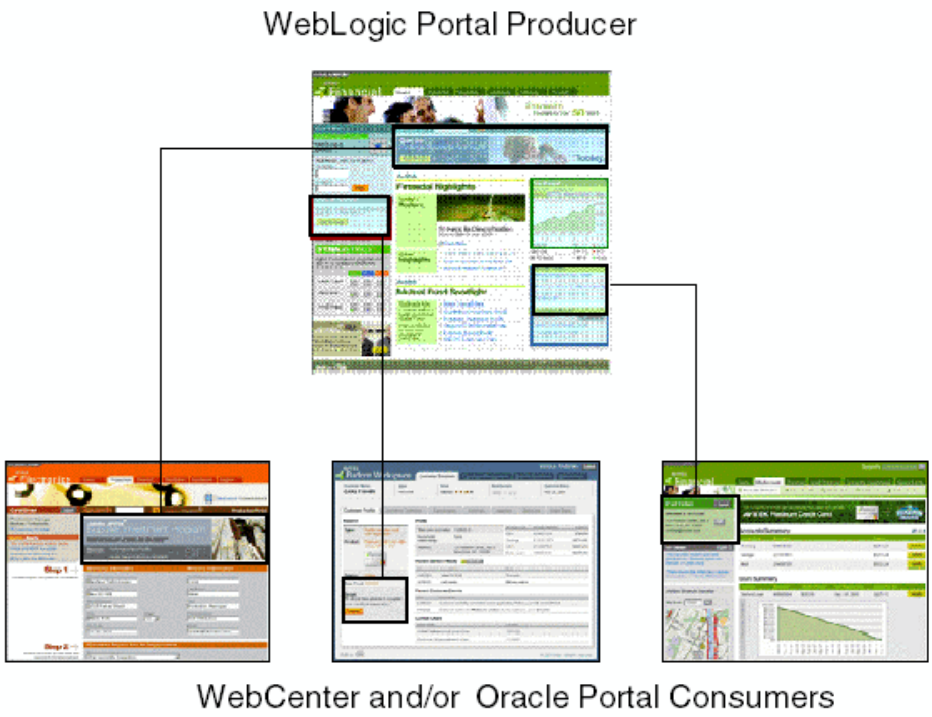
13.1 Consuming WLP Portlets in WebCenter Portal Applications and Oracle Portal Applications

This section describes the recommended technique for consuming WLP portlets in WebCenter Portal applications and Oracle Portal applications. In this scenario, WebLogic Portal is the *producer* and WebCenter Portal/Oracle Portal is the *consumer*, as illustrated in Figure 13–1.

Because of an incompatibility between the way WLP and WebCenter Portal/Oracle Portal implement certain WSRP operations, user authentication errors can occur. This section describes the nature of this incompatibility and the procedure for avoiding these errors.

This section includes these topics:

Figure 13–1 Consuming WLP Portlets in WebCenter Portal: Framework Applications



Note: When consuming JSF portlets, ensure that JSP encoding is set to UTF-8.

13.1.1 Understanding the Cause of User Authentication Errors

An incompatibility exists between the way WebLogic Portal and WebCenter Portal/Oracle Portal implement the WSRP operations clonePortlet, destroyPortlets, importPortlets, and exportPortlets. Because of this incompatibility, you might encounter user authentication errors when trying to consume portlets from a WebLogic Portal producer in Framework applications or Oracle Portal applications.

Note: For detailed information on the clonePortlet, destroyPortlets, importPortlets, and exportPortlets operations, refer to the Oasis Standard document *Web Services for Remote Portlets Specification v2.0* at `http://docs.oasis-open.org/wsrp/v2/wsrp-2.0-spec-os-01.html`.

Specifically, JDeveloper makes *unauthenticated* operation calls at various points in the development lifecycle:

- clonePortlet is called when dragging a portlet from a registered WebLogic Portal producer onto a JSF page or ADF task flow.
- destroyPortlets is called when deleting a portlet from a JSF page or ADF task flow.
- exportPortlets is called when exporting an application containing a WebLogic Portal remote portlet to an EAR, for deployment onto a production server.
- importPortlets is called when deploying a previously-exported EAR onto a production server.

The compatibility problem arises because the WebLogic Portal producer requires that a user be authenticated when the clonePortlet, destroyPortlets, importPortlets, or exportPortlets operations are invoked, while the WebCenter Portal/Oracle Portal producer does not require user authentication for these methods.

13.1.2 Preventing User Authentication Errors

To prevent user authentication errors when consuming WebLogic Portal portlets in a Framework application or Oracle Portal application, do the following:

1. In the WebLogic Portal producer application, create a new user to function as a surrogate for the WSRP operations listed in the previous section. This user should be in the Portal System Administrators group.
2. Copy the file `WEB-INF/wsrp-producer-config.xml` to your workspace and open it for editing. The procedure for copying files to your workspace is described in "Copying J2EE Library Files Into a Project" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.
3. Add the following security element attribute to the `wsrp-producer-config.xml` file. This element must be the last element in the `<wsrp-producer-config>` element (place it just above the closing `</wsrp-producer-config>` line):

   ```
   <security anonymousCloneDestroyUser="username"/>
   ```

   where *`username`* is the name of the user that you created.
4. Save your changes, and republish your web project.

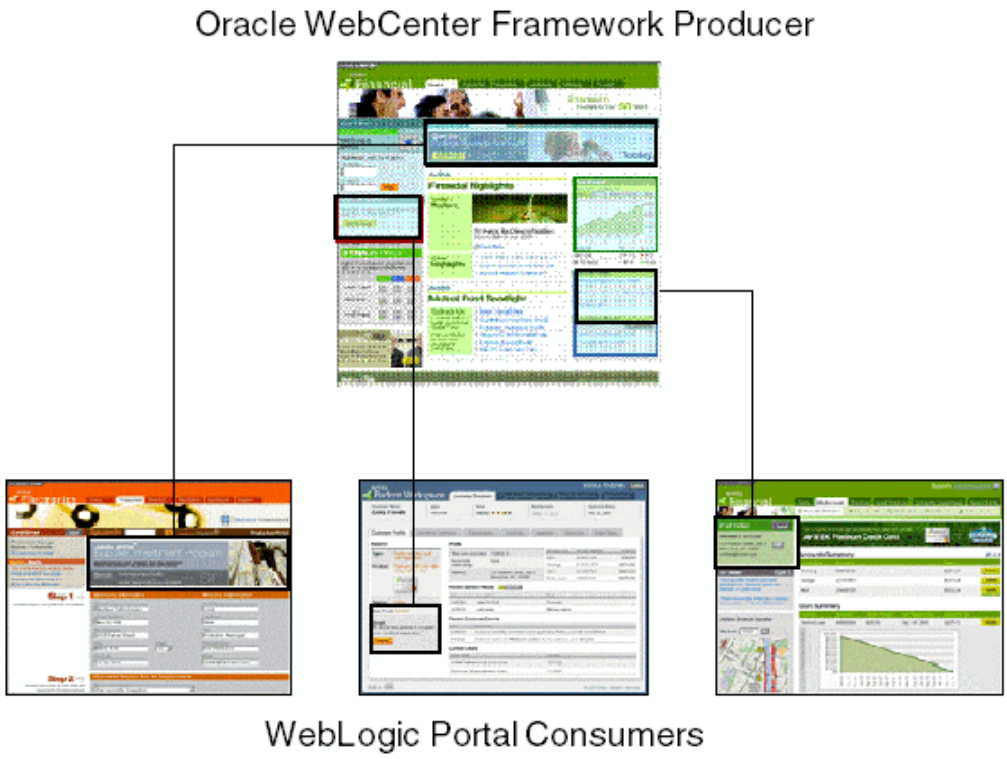
With this configuration, the WebLogic Portal producer server automatically authenticates the specified user when unauthenticated operation calls to the four WSRP operations (described in the previous section) are received. This automatic authentication only occurs for these four operations. Also, this automatic authentication works only for complex producers.

13.2 Consuming WebCenter Portal Portlets in WebLogic Portal

This section discusses techniques and best practices for consuming WebCenter Portal portlets in a WebLogic Portal application. In this scenario, WebLogic Portal is the *consumer* and WebCenter Portal/Oracle Portal is the *producer*, as illustrated in Figure 13–2. Topics in this section include:

- Section 13.2.1, "Avoiding Cookie Collisions"
- Section 13.2.2, "Configuring for Partial Page Refresh Over WSRP"
- Section 13.2.3, "Configuring Portlets That Use ADF Faces Rich Client Components"
- Section 13.2.4, "Consuming WebCenter Portal Service Portlets"

Figure 13–2 Consuming WLP Portlets in WebCenter Portal Applications



13.2.1 Avoiding Cookie Collisions

A WebCenter Portal producer and a WebLogic Portal consumer have the same cookie-name and path information configured by default. Some resource requests from the consumer to the producer will cause the producer's cookie name and value to be set on the client (the browser). When this occurs, it has the effect of overriding the value set by the consumer. This situation puts the client-consumer-producer interactions into this cycle:

1. Consumer server sets the cookie value on the client (browser).
2. Client (browser) sends the consumer's cookie value to the producer.
3. The producer doesn't recognize the value, and sends the client a new cookie value.
4. The client sends the producer's cookie value to the consumer.
5. The consumer doesn't recognize the value, and sends the client a new cookie value and so on.

Because session state is usually persisted based on the cookie value, this constant change in cookie values causes session state to be lost on both the producer and consumer.

Fortunately, there are several easy ways to correct this situation. These techniques include using different cookie names, using a system property, and blocking cookies. For details, see Section 14.15, "Configuring Session Cookies" in Chapter 14, "Other Topics and Best Practices."

13.2.2 Configuring for Partial Page Refresh Over WSRP

Certain ADF Faces Rich Client components make use of "Partial Page Refresh" JavaScript to update the page without reloading it completely. In some circumstances this does not work properly over WSRP due to improperly encoded ampersand characters. To fix this, add the following line to the consumer's `WEB-INF/wlp-framework-common-config.xml` file, just before the `</wlp-framework-common-config>` tag:

```
<ignore-wsrp-ampersand-encoding-extension default="true"/>
```

> **Tip:** The easiest way to edit this file is to copy it to your Eclipse workspace. To do this, locate the file in the Merged Projects view. Right-click the file and select **Copy to Workspace**.

13.2.3 Configuring Portlets That Use ADF Faces Rich Client Components

To consume portlets that use Oracle ADF Faces Rich Client Components in a WLP consumer, you must perform the following configuration tasks on the WLP consumer. The steps include:

- Section 13.2.3.1, "Using iframe_unwrapped"
- Section 13.2.3.2, "Disabling html-amp-entity in WEB-INF/wlp-framework-common-config.xml"
- Section 13.2.3.3, "Using CSS Styling (Optional)"
- Section 13.2.3.4, "Setting a Fixed Height on the Portlet's Contents (Optional)"

For information on Oracle ADF Faces Rich Client Components, see `http://www.oracle.com/technology/products/adf/adffaces/index.html`.

13.2.3.1 Using iframe_unwrapped

ADF Faces Rich Client portlets contain base-level HTML tags (like HTML, HEAD, BODY, and BASE) that WebLogic Portal already provides in a rendered portal page. Having such tags appear multiple times in HTML can result in the misbehavior of JavaScript on the page, among other things. To avoid these problems, render the contents of these portlets inside an IFRAME element.

To render an ADF Faces Rich Client portlet in an IFRAME, the portlet property Async Content Rendering must be set to **iframe_unwrapped**. The **iframe_unwrapped** setting renders the body of the remote portlet, "unwrapped" from any HTML artifacts that the WebLogic Portal framework would provide (for instance, skeleton artifacts such as HTML, HEAD, BODY, and other tags). For more information on the Async Content Rendering property, see "Asynchronous Portlet Rendering" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*. For information on skeletons, see "What is a Skeleton" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

13.2.3.2 Disabling html-amp-entity in WEB-INF/wlp-framework-common-config.xml

Certain ADF Faces Rich Client components make use of JavaScript to handle the display of images, such as changing a button image when it is rolled-over. An example of such a case might look like this, as served by the producer:

```
<script type="text/javascript">
...
```

```
    var hoverIcon = 'wsrp_
rewrite?wsrp-urlType=resource&amp;wsrp-url=http%3A%2F%2Fhost.com%3A8899%2Fproducer
%2Fadf%2Fwebcenter%2Fattributegroup_
ovr.png&amp;wsrp-requiresRewrite=false&amp;wsrp-extensions=oracle%3Astateless-reso
urce%3Dtrue/wsrp_rewrite';
...
</script>
```

The WebLogic Portal consumer is responsible for converting the information between the `wsrp_rewrite?` and `/wsrp_rewrite` tokens into a valid URL for that resource. When a URL like this one is a value for markup attributes, any ampersand characters within it should be encoded as `&`, according to markup validation rules. However, when the URL is to be used inside of non-markup content, such as JavaScript and CSS, it should explicitly not be encoded as `&`. Doing so causes the request to the server to send parameter names like `amp;_nfpb` instead of `_nfpb`, which typically results in an unintended or unexpected response from the server. For information on markup validation rules, see `http://www.htmlhelp.com/tools/validator/problems.html#amp`.

The WebLogic Portal URL framework performs ampersand entity encoding on all URLs in the text/* markup type by default. For interoperability with ADF Faces Rich Client portlets produced by the WebCenter Portal, this default setting should be changed. To do so:

1. Copy `WEB-INF/wlp-framework-common-config.xml` into your project, and open it for editing.
2. Change the value of the `html-amp-entity` element to false.
3. Save your changes, and republish your web project.

13.2.3.3 Using CSS Styling (Optional)

The Oracle WSRP Consumer sends a custom WSRP extension to the WebCenter Portal producer, which tells the producer what ADF Faces skin is currently applied on the consumer. The WebCenter Portal producer uses this information to return a matching CSS stylesheet for the portlet being requested. The WebLogic Portal consumer does not currently support this custom WebCenter Portal WSRP extension, and so every portlet returned from a WebCenter Portal producer will be styled with the "portlet" version of the default skin (for example, `blafplus-rich.portlet`). If you would like to style a portlet consumed in WebLogic Portal based on a specific ADF Faces skin, follow these steps:

1. On the WebCenter Portal producer, open the `WEB-INF/portlet.xml` file for editing.
2. Find the `<portlet>` element for the portlet to which to apply the skin.
3. Add the following `<init-param>` element, underneath the `<portlet-class>` element:

```
<init-param>
    <name>org.apache.myfaces.trinidad.skin.id</name>
    <value>blafplus-rich.desktop</value>
</init-param>
```

where `blafplus-rich.desktop` is the ID of the specific skin to be applied to this portlet, when a consumer does not use the Oracle-specific WSRP extension to specify a skin.

4. Redeploy your portlet producer application.

13.2.3.4 Setting a Fixed Height on the Portlet's Contents (Optional)

The WebLogic Portal framework attempts to resize a portlet's IFRAME to match the height of the rendered contents within it. However, certain ADF Faces Rich Client components also attempt to resize themselves according to the size of the surrounding window or frame. This leads to a scenario where the initial size of the contents of the IFRAME are based on the initial size of the IFRAME itself, before any client-side dynamic resizing has occurred.

In these cases, you may prefer to set a fixed height for the contents of your portlet. To do so:

1. In WebLogic Workshop, right-click your `.portlet` file and select **Open With > XML Editor**.
2. On the `<netuix:proxyPortletContent>` element, add the `presentationStyle` attribute, with a value similar to `min-height: nnnpx;`, where `nnn` is the desired pixel height of your portlet's body. For example:

   ```
   <netuix:proxyPortletContent presentationStyle="min-height: 600px;"/>
   ```

3. Save your changes, and republish your web project.

13.2.4 Consuming WebCenter Portal Service Portlets

For service portlets, the WebCenter Portal producer sends a WSRP URL extension parameter, which is used for URL rewriting, to the WSRP consumer. If you encounter the problem that popups are not appearing in service portlets, configure `wlp-framework-common-config.xml` to ignore the WSRP URL extension parameter, `oracle:escape-xml`. To do so:

1. Open `WEB-INF/wlp-framework-common-config.xml`.
2. Add the following entry:

   ```
   <ignore-wsrp-ampersand-encoding-extension default="true"/>
   ```

   This entry must be added as the last entry before the closing `</wlp-framework-common-config>` element in the configuration file.
3. Save your changes, and republish your web project.

13.3 Configuring Security

For information on configuring security between WebLogic Portal and WebCenter Portal, see Chapter 17, "Configuring WSRP Security Between WLP and a WebCenter Portal: Framework Application." That chapter describes one technique for establishing a secure communications channel for WSRP transactions between WLP and WebCenter Portal.

13.4 Interoperation of Navigational Parameters

This section shows by example how to configure interoperability of navigational parameters through WSRP.

The sample tasks described in this section are:

- Section 13.4.1, "Sharing Portlet Parameters Between WLP Portlets and WebCenter Portal Portlets"

- Section 13.4.2, "Consuming a WebCenter Portal Portlet that requires Shared Navigational Parameters With an Initial Value"

13.4.1 Sharing Portlet Parameters Between WLP Portlets and WebCenter Portal Portlets

WLP and WebCenter Portal Java portlets are able to share parameters over WSRP. WebCenter Portal calls these shared parameters "public render parameters", while WLP calls them "shared parameters", and the JSR286 specification calls them "public parameters". As long as the configured QNames for these parameters match, they will automatically be shared between portlets consumed over WSRP.

For detailed information, see "Using Shared Parameters" in *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*, and "How to Use Public Render Parameters" in *Oracle Fusion Middleware Developer's Guide for Oracle WebCenter*.

13.4.2 Consuming a WebCenter Portal Portlet that requires Shared Navigational Parameters With an Initial Value

1. In Workshop for WebLogic, register a WebCenter Portal producer application.
2. In Workshop, create a remote portlet consuming a portlet in the WebCenter Portal producer.
3. Create a backing file for the portlet with an onInit method. In this method, set `param` to the shared parameter name and `default_param_value` to the initial value you would like to set.
4. Open the `.portlet` file with the XML editor and add the following as the last attribute of netuix:proxyPortlet.

   ```
   backingFile="com.bea.wsrp.qa.SetSharedParamsBacking"
   ```

5. Add this element after the netuix:proxyPortlet element, where `shared_param_name` is the name of the shared navigational parameter that the WebCenter Portal portlet requires.

Example 13–1 Backing File onInit() Method

```
public void onInit(HttpServletRequest request, HttpServletResponse response, Event event) {

    PortletBackingContext portletBackingContext =
    PortletBackingContext.getPortletBackingContext(request);
    String foo = portletBackingContext.getSharedParameterValue("param");
    if (foo == null || foo.length() == 0) {
        portletBackingContext.setSharedParameterValue("param", "default_param_value");
    }
}
```

Example 13–2 Element Added to .portlet File

```
<netuix:handlePortalEvent event="onInit" eventLabel="handlePortalEvent1"
          fromSelfInstanceOnly="false" onlyIfDisplayed="true">
          <netuix:invokeBackingFileMethod method="onInit"/>
      </netuix:handlePortalEvent>
      <netuix:sharedParameter paramId="shared_param_name" qname="shared_param_name"/>
```

13.5 Special Considerations

This section describes additional issues to consider with respect to WLP and WebCenter Portal interoperability.

13.5.1 Interportlet Communication Considerations

Portlets using WebLogic Portal's IFRAME feature are unable to respond to public NavigationalContext changes. Therefore, if you want to share data between portlets across WebCenter Portal and WebLogic Portal containers, consider setting attributes in the HttpSession. For further information on this topic, see the WebLogic Server Session topic "Using Sessions and Session Persistence" in *Oracle Fusion Middleware Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*. For information on event distribution and the NavigationalContext type, refer to the Oasis Standard document *Web Services for Remote Portlets Specification v2.0* at `http://docs.oasis-open.org/wsrp/v2/wsrp-2.0-spec-os-01.html`.

> **Note:** Be aware that when you use HttpSession to share data, you create a dependency between portlets, which is not generally considered to be a best practice. See Section 14.2, "Avoid Interportlet Dependencies."

Even when using HttpSession attributes to share data across portlets, those inside of IFRAMEs will be isolated from the notification of the change. A reload of the entire portal page is required, so that all of the portlets residing in IFRAMEs can update their contents with the new HttpSession attribute information. To facilitate this programatically, consider using a WSRP interceptor. In the interceptor shown in Example 13–3, the responses from POST requests to a WebCenter Portal producer are being changed to force the browser's outermost window to reload itself. See also Chapter 9, "The Interceptor Framework."

Example 13–3 Interceptor to Force Reload of a Parent Frame

```
package com.oracle.wlp.wsrp;

import java.io.ByteArrayInputStream;
import java.io.UnsupportedEncodingException;

import com.bea.wsrp.consumer.interceptor.IGetMarkupInterceptor;
import com.bea.wsrp.consumer.interceptor.Status;
import com.bea.wsrp.consumer.interceptor.Status.OnFault;
import com.bea.wsrp.consumer.interceptor.Status.OnIOFailure;
import com.bea.wsrp.consumer.interceptor.Status.PostInvoke;
import com.bea.wsrp.consumer.interceptor.Status.PreInvoke;
import com.bea.wsrp.model.markup.IGetMarkupRequestContext;
import com.bea.wsrp.model.markup.IGetMarkupResponseContext;

public class ReloadInterceptor implements IGetMarkupInterceptor {
    // A very simple piece of HTML markup that, via JavaScript, will cause
    // the parent frame to reload. Tested in IE7 and FF3.
    private static final String RELOAD_MARKUP =
        "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\"
\"http://www.w3.org/TR/html4/loose.dtd\"><html><head><script
type=\"text/javascript\">window.parent.location.reload(true);</script></head><body></body></html>";

    @Override
```

```
    public PostInvoke postInvoke(IGetMarkupRequestContext request, IGetMarkupResponseContext
response) {
        // Do something here to determine if you'd like to reload the parent frame.
        // In this example, we're forcing a reload of the parent frame if this request
        // was of type "POST".
        if (request.getRequestVerb().equalsIgnoreCase("POST")) {
            try {
                // Completely replace the response markup with our simple "reload" HTML markup.
                response.setMarkupData(new ByteArrayInputStream(RELOAD_MARKUP.getBytes("UTF-8")));

                // Abort the interceptor chain, as we don't want any other interceptors
                // to undo what we've just done. This is ok, as a subsequent GET request
                // will soon follow, as the parent frame is reloaded.
                return Status.PostInvoke.ABORT_CHAIN;
            } catch (UnsupportedEncodingException e) {
                // Ignore, as in this case the encoding will always be valid.
            }
        }
        return Status.PostInvoke.CONTINUE_CHAIN;
    }

    @Override
    public PreInvoke preInvoke(IGetMarkupRequestContext request) {
        return Status.PreInvoke.CONTINUE_CHAIN;
    }

    @Override
    public OnFault onFault(IGetMarkupRequestContext request, IGetMarkupResponseContext response,
Throwable t) {
        return Status.OnFault.CONTINUE_CHAIN;
    }

    @Override
    public OnIOFailure onIOFailure(IGetMarkupRequestContext request, IGetMarkupResponseContext
response, Throwable t) {
        return Status.OnIOFailure.CONTINUE_CHAIN;
    }
}
```

13.5.2 Consuming Oracle ADF Faces Rich Client Component Portlets

If you want to consume Oracle ADF Faces Rich Client Component Portlets in WLP, you must use WebLogic Workshop. The WebLogic Portal Administration Console does not currently support the portlet configuration required for consuming Oracle ADF Faces Rich Client Component portlets.

For information on Oracle ADF Faces Rich Client Components, see `http://www.oracle.com/technology/products/adf/adffaces/index.htm l`. For information on using the WLP Administration Console, see the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

14

Other Topics and Best Practices

This chapter focuses on best practices for developing portlets in a producer. By following the practices described in this chapter, you will help to ensure that remote portlets created in consumers function properly. We recommend that you review Chapter 3, "Federated Portal Architecture" before using this chapter.

This chapter the following sections:

14.1 Decouple Rendering from Interaction

As explained in Section 3.5, "Life Cycle of a Remote Portlet", the rendering and interaction phases of a remote portlet's life cycle are decoupled. As a result, you cannot expect a portlet to receive the same HTTP response or request for the render phase as it receives for an interaction.

A portlet that is being rendered must not expect to receive form data in the request object. This is because the request may have been submitted some time ago and is being rendered now, and you may not have the same data.

If you want to maintain data between requests, you need to store that data locally, typically in the session. For instance, if you are processing and order ID, you can store that ID locally.

If you are using page flows, data is automatically passed forward. However, if you are using backing files with a remote portlet, you need to make sure that data is stored in the session, because you won't get back the same request object.

Note: Page flows are a feature of Apache Beehive, which is an optional framework that you can integrate with WLP. See "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

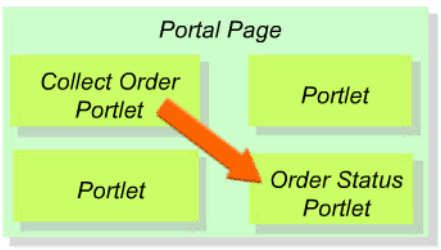
To avoid problems, keep the following points in mind:

- Portlets will not get the same servlet request for the interaction and render phases, even after the first rendering after an interaction.
- Decouple rendering from interaction processing. A portlet should be able to render itself as many times as necessary without depending on the user's interaction directly. Store interaction changes for future rendering. Note that WebLogic Portal stores state automatically for page flows.
- Use JSP tags with render parameters.
- Use HTTP session for backing files.

14.2 Avoid Interportlet Dependencies

Rather than create explicit dependencies between portlets, use events to communicate between portlets. For example, suppose that on a portal page, there is a portlet for collecting orders and a portlet for displaying the status of all orders. When an order is taken, data is stored in the database, and the data is then displayed in the order status portlet, as shown in Figure 14–1.

Figure 14–1 Interportlet Dependencies



In this scenario, a strong dependency is created between the collect order and the order status portlets. The Collect Order portlet needs to somehow communicate some information (the order ID) to the Order Status portlet. Storing the ID in the session or other common state between the portlets creates a strong dependency between the Collect Order and Order Status portlets. Depending on the implementation of the portlets, if one of them is changed or replaced, the changes will necessarily affect the other portlet.

To avoid this dependency, use events to communicate between portlets. In this example, if an event is used to communicate order information to the order status portlet, the order status portlet does not have to care about where the order came from. The order status portlet just handles an event, retrieving, for example, an order ID from the event's payload.

For more information on how event handling occurs in WebLogic federated portals, see Section 3.5.4, "Interportlet Communication with Events".

- Use events to communicate between remote portlets.
- Use event names for dependencies.
- Avoid using sourceDefinitionLabels on events.

14.3 Avoid Portal Layout Dependencies

Some portals are built with inherent portal layout dependencies. For example, a login portlet might be designed to function differently if it is on a human resources page versus a finance page. In other words, when an interaction takes place, the portlet tries to find out what page it is on before taking action. This practice closely couples the portlet to the Portal Framework elements, such as pages, books, or desktops on the consumer.

This scenario does not work in a federated portal, because the producer does not know what page layouts exist on the consumer. Avoid this scenario when possible. If it is required, deploy those portlets locally on the consumer, or use shared components where possible and create alternative layouts that are offered through separate portlets.

14.4 Avoid Coupling by URL

If you embed URLs in your portlets, such as in links, you may find that your portlets work as expected when they are running locally. However, when you move those portals to a federated environment, the links no longer work. For example, in the following code fragment, a developer is invoking the action of a page flow portlet on the same portal using string manipulation.

> **Note:** Page flows are a feature of Apache Beehive, which is an optional framework that you can integrate with WLP. See "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

In a federated portal, this sort of construction will not work. Typically, this sort of programming arises because of reverse engineering, where a developer looks at and copies how links are created.

```
String url = "http://mydomain.com/portal/portal.portal?";
url = url + "myportlet_actionOverride=login";
```

```
url = url + "...";
```

Likewise, the following resource URL will not work in a federated portal because it includes an explicitly specified link to a document. Because the document doesn't exist on the consumer, the consumer doesn't know what to do with it:

```
<img src="images/logo.jpg"/>
<a href="/docServlet?docId=9999">Download</a>
```

Common URL problems found in federated portals include the following. These problems stem from the fact that remote portlets do not follow the same URL structure as portlets in a local environment.

- Creating links to page flow actions, images or files through string manipulation.

  Note: Page flows are a feature of Apache Beehive, which is an optional framework that you can integrate with WLP. See "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

- Directly adding parameters to URL strings.
- Getting a page flow action name from the outer request.

It is important that you let the WebLogic Portal Framework create URLs for you using the proper set of JSP tag libraries and utility classes. Use the following tags and classes:

- netui tags
- page flow tags
- struts tags
- render tags
- GenericURL class

All of these tags go through the WebLogic Portal URL rewriters and will work properly in a federated environment.

It is important to realize that there are inherent differences between remote portlets and local portlets. Developers must not expect that all correctly functioning local portlets will function properly as remote portlets, although in many cases they do.

14.5 Avoid Accessing Request Parameters in Rendering Code

When you deploy a local portlet, the portlet can access the request parameters from the portal's request and the request attributes set by other portlets on the same page. If you implement a portlet to depend on such request parameters and attributes, the portlet will not function correctly in a WSRP environment. In a WSRP environment, remote portlets are running on remote systems; the HTTP request received by a remote portlet on a producer is not the same as the one that is received by the consumer portal.

14.6 Avoid Moving Producers

When you add producers and create remote portlets, the producer registry (`WEB-INF/wsrp-producer-registry.xml`) and the portal framework database tables contain specific information about the producer, such as its WSDL address and

the addresses of ports described in the WSDL. If you move the producer from one environment to another, this data becomes invalid. In this case, consumers whose proxy portlets reference the producer's portlets will no longer be able to find them.

If you must move a producer from one environment to another (such as a staging to a production environment) WebLogic Portal supports two mechanisms for achieving this.

The first mechanism, shared registration, is only recommended for WebLogic Portal producers older than version 10.0. With shared registration, the staging and production environments share the same producer registration handle. This model has a number of serious drawbacks. Only use this model when the producer is a version of WebLogic Portal prior to 10.0. For more information on shared registration and propagating WSRP producers in this case, see the *Oracle Fusion Middleware Production Operations Guide for Oracle WebLogic Portal*.

The second mechanism is recommended for producers, such as WebLogic Portal versions 10.0 and higher, that support WSRP 2.0 exportPortlet and importPortlet operations. When producers are propagated using these operations, producer registration handles do not need to be shared. The propagation tool in WebLogic Portal versions 10.0 and higher handles these operations automatically. See the *Oracle Fusion Middleware Production Operations Guide for Oracle WebLogic Portal* for details.

You can also update the database entries for a producer programmatically. The following class provides methods to get and update producer information:

`com.bea.wsrp.consumer.management.producer.ProducerManager`

Refer to the *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal* for information on this class.

14.7 WebLogic Server Producers

In some cases, you may want to expose portlets with WSRP from a producer environment that does not include any WebLogic Portal components. For example, you may be running a Struts Web application in a Basic WebLogic Server Domain, or a Java Page Flow application in a Basic Oracle Enterprise Pack for Eclipse Domain. In either case, WebLogic Portal is not part of the server configuration. For detailed information on using a non-portal server domain to host remote portlets, see Chapter 8, "Configuring a WebLogic Server Producer."

> **Note:** Page flows are a feature of Apache Beehive, which is an optional framework that you can integrate with WLP. Apache Struts is also an optional framework that you can integrate with WLP. See "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

If you are using a Portal Web application as your producer, all the portal artifacts are available in the web application; however, for any WSRP producer that is not a Portal Web application, you cannot use portal features such as property sets. If you need to access portal features in your producer, use a Portal Web application.

14.8 Security for Remote Portlets

Securing WSRP messages ensures their confidentiality between just the interested parties. When a portlet's messaging is secure, only parties authorized to handle the contents of that portlet's messages can see those messages. To secure WSRP messages:

- Use SSL on any port through which the Producer will be offered.
- Configure the Producer to offer secure portlets by specifying `true` for all secure attributes in the `<service-config>` element of the Producer project's `WEB-INF/wsrp-producer-config.xml` file, as shown in Example 14–1.

Example 14–1 <service-config> Element Configured for Security

```
<service-config>
   <registration required="true" secure="true"/>
   <service-description secure="true"/>
   <markup secure="true" rewrite-urls="true" transport="string"/>
   <portlet-management required="true" secure="true"/>
</service-config>
```

If you make any changes to `wsrp-producer-config.xml`, you will need to redeploy or restart the server before the changes become active.

For detailed information on configuring single sign-on security for federated portals, see:

- Chapter 15, "Establishing WSRP Security with SAML" – Discusses how to configure SAML security between WebLogic Portal domains. This chapter also covers cross version compatibility: security between WebLogic Portal 9.2 and 8.1x domains.
- Chapter 16, "Configuring User Name Token Security" – User Name Token, or UNT, is an alternative to SAML and provides the same basic single sign-on capability as SAML provides.

14.9 Error Handling

This section gives an overview of error handling techniques for federated portals.

14.9.1 On the Producer

To prevent stack traces from appearing, handle errors on the producer side and provide a suitable business message.

14.9.2 On the Consumer

In Oracle Enterprise Pack for Eclipse, with a remote portlet open:

1. Click the portlet in the editor to display the Properties view.
2. Enter a path for the error page (JSP or HTML page).

14.9.3 Interceptors

You can use interceptors to handle errors returned from a producer. For instance, if a specific producer is not registered, you can trap the registration error and handle it as you wish. For detailed information on using interceptors, see Chapter 9, "The Interceptor Framework."

14.10 Portlet Programming Guidelines and Best Practices

This section discusses guidelines and best practices for developing remote portlets.

- Requests and Sessions

  If two or more remote portlets share session data, host them on the same producer. You cannot assume that session information will be shared by portlets hosted on different systems.

- Look and Feel
  - Let portlets use standard style attributes and specify those attributes on the portal skins. For information on look and feel, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

- Backing Files
  - You can use backing files on the consumer side (remote-portlet) to take some action based on session / request objects or property sets. For information on backing files, see the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

- Caching WSRP Portlets
  - Producer – Use `<wl:cache>` or `p13nCache` wherever possible.
  - Consumer (remote-portlet) – Use the `RenderCacheable` attribute if you want to cache the remote portlet's rendered HTML. However, this is a session scoped cache and is not configurable.

  For more information on caching, see the "Portlet Caching" section of the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

14.11 Designing for Performance

To ensure optimal performance of your producers and consumers, we recommend the following performance tuning guidelines on the producer and the consumer.

14.11.1 Performance Guidelines for Producers

This section lists several ways to improve the performance of producer applications.

14.11.1.1 Reorder Authentication Providers

One way to improve performance on the producer is to make sure the SAML Authentication Provider is deployed in front of other authentication providers. To reorder the providers:

1. Log in to the WebLogic Server Administration Console.
2. Select **Security Realms** in the Domain Structure tree.
3. In the Realms table, select the active security realm used by the producer application.
4. In the Settings page, select the Providers tab.
5. In the Change center, click **Lock & Edit**.
6. Below the Authentication Providers table, click **Reorder**.
7. In the list of providers, use the arrow buttons to move SAMLAuthenticator to the top of the list, and click **OK**.

8. In the Change center, click **Activate Changes**.

14.11.1.2 Enable Attachment Support

Enable attachment support by adding `<markup transport="attachment"/>` to `WEB-INF/wsrp-producer-config.xml`, as shown in Example 14–2.

Example 14–2 Enabling Attachment Support

```
<?xml version="1.0" encoding="UTF-8"?>
wsrp-producer-config
    xmlns="http://www.bea.com/servers/weblogic/wsrp-producer-config/8.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bea.com/servers/weblogic/wsrp-producer-config/8.0
wsrp-producer-cnfig.xsd">
    <service-config>
        <registration required="false" secure="true"/>
        <service-description secure="true"/>
        <markup secure="true" rewrite-urls="true" transport="attachment"/>
        <portlet-management required="false" secure="true"/>
    </service-config>
```

14.11.1.3 Other Techniques

- Let the producer create correct URLs by using consumer-supplied URL templates. This is the default practice.
- Use caching. For more information on caching, see the section "Portlet Caching" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.
- Enable multi-threaded (forked) rendering. For more information, see the section "Portlet Forking" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

14.11.2 Performance Guidelines for Consumers

- Accept the default behavior to enable caching for remote portlets.
- Enable forked rendering for remote portlets.
- Set connection timeout. See Section 5.6, "Setting a Timeout Value on a Remote Portlet" for detailed information on setting timeouts.
- Disable logging by undeploying MessageMonitor servlet from `WEB-INF/web.xml`.

14.12 Using Local Proxy Mode

Local proxy support allows co-located producer and consumer web applications to short-circuit network I/O and "SOAP over HTTP" overhead. When you enable this feature, the consumer tries to determine if the producer is deployed on the same server and, if it discovers that the producer is so deployed, it uses a local proxy to send requests to the producer. If the producer is not deployed on the same server, the consumer uses the default remote proxy. Remote producers can still be invoked as usual even when the local proxy support is enabled.

This section describes how to implement local proxy support. It includes information on the following subjects:

- Section 14.12.1, "Why Use Local Proxy Mode?"

14.12.1 Why Use Local Proxy Mode?

Local proxy mode provides a number of advantages over the default remote proxy when you are working with co-located consumers and producers. Among the most significant advantages of local proxy mode are:

- Avoids local network I/O.
- Avoids serialization and deserialization of SOAP.
- Invokes remote portlets using the same execute thread.
- Writes portlet markup directly to the response without intermediate buffers.
- Enables large file uploads without causing `OutOfMemoryErrors`.

Additionally, by enabling local proxies, customers can take advantage of the decoupling benefits of WSRP without incurring its performance overhead.

14.12.2 Deployment Configuration

To take advantage of local proxy support:

1. Deploy the producer and consumer web applications on the same server. These applications could be in the same enterprise application or across different enterprise applications.
2. Enable local proxy support by setting `<enable-local-proxy>` to `true` in `WEB-INF/wsrp-producer-registry.xml` in the consumer web application, as shown in Example 14–3:

Example 14–3 Setting <enable-local-proxy> to "true"

```
<wsrp-producer-registry
   xmlns="http://www.bea.com/servers/weblogic/wsrp-producer-registry/8.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/servers/weblogic/wsrp-producer-
      registry/8.0 wsrp-producer-registry.xsd">
   <!-- Upload limit (in bytes) -->
   <upload-read-limit>1048576</upload-read-limit>
   <!-- Timeout (in milli seconds) -->
   <connection-timeout-secs>120000</connection-timeout-secs>
   <!-- Enable local proxy -->
   <enable-local-proxy>true</enable-local-proxy>
...
</wsrp-producer-registry>
```

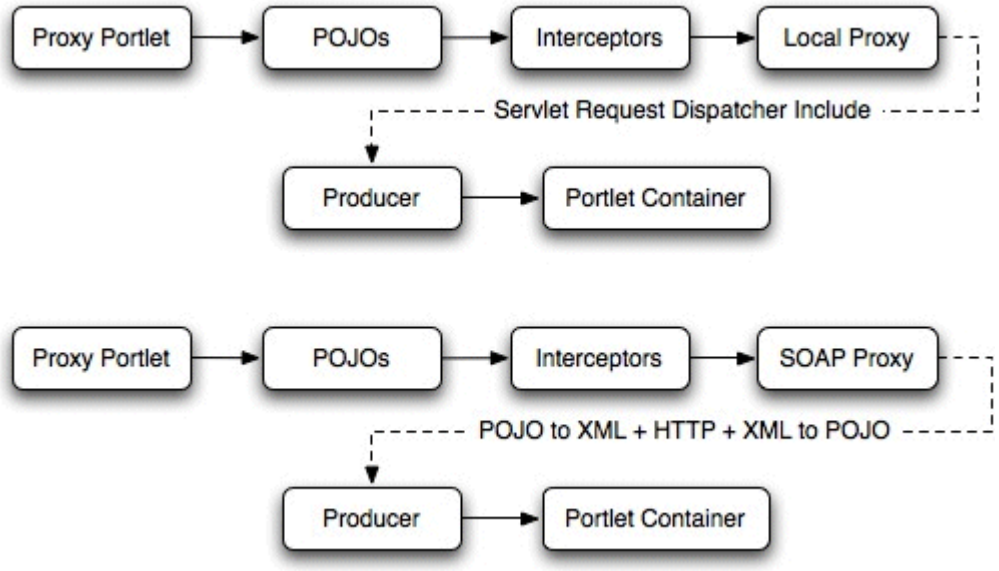
You can also enable local proxy support by setting a system property `com.bea.wsrp.proxy.LocalProxy.enabled = true`. If this system property is set to true, it will override the `<enable-local-proxy>` setting in `WEB-INF/wsrp-producer-registry.xml`.

Local proxy support is disabled by default in web application templates.

14.12.3 How Local Proxy Mode Works

Figure 14–2 compares the layers of operations involved when local proxy support is enabled (top flow diagram) and when it is not (bottom flow diagram). In the local proxy case, there is no network or SOAP related overhead and the servlet API is used for communication.

Figure 14–2 Local Versus Remote Proxy Flow Diagrams



Note: It is a recommended practice to enable local proxy mode when you deploy JSR-168 portlets using the JSR-168 Import Utility. As far as performance and complexity are concerned, there is no difference between the way JSR-168 portlets and WSRP local proxy interoperate in WebLogic Portal versus other vendors. For more information on the import utility, see the section "Deploying JSR-168 Portlets in a WAR File" in the *Oracle Fusion Middleware Production Operations Guide for Oracle WebLogic Portal*.

Table 14–1 summarizes the evolution of the WebLogic Portal local proxy architecture.

Table 14–1 Evolution of Local Proxy Architecture for WebLogic Portal

Version	Local Proxy Architecture
WLP 8.1x	Exchange XmlBeans between consumer and producer.
WLP 9.2	Convert POJOs into XmlBeans while sending data from the consumer to the producer.
WLP 10.0	Exchange POJOs between the consumer and the producer.

14.12.4 When to Use and Not Use

As powerful a tool as local proxy support is, you should only use it when it will benefit your application. The most common reasons for using local proxy support are:

- When portlets are deployed in self-contained web applications on the same server. The local proxy support provides isolated portlet deployment. In this mode, each

portlet web application can be deployed as a WSRP producer. Portlets can therefore be loaded by separate class loaders and have their own servlet context and session. Portlet web applications can be deployed/undeployed without affecting the portal web application.

- When you don't need advanced monitoring software between the producer and consumer.

On the other hand, you should not use local proxy support when interoperating with non-Oracle producers and consumers.

14.13 Monitoring and Logging

You can monitor activity between producers and consumers by using the message monitor servlet installed with Oracle Enterprise Pack for Eclipse. You can also create custom logs to display specific information about WSRP sessions. These features can help you debug problems with remote portlets.

This section contains information on these subjects:

- Section 14.13.1, "Using the Monitor Servlet"
- Section 14.13.2, "Creating Custom Logs"

14.13.1 Using the Monitor Servlet

To monitor the response and request headers, as well as the action SOAP messages that are passed between producers and consumers:

1. Ensure that the producer and consumer applications whose communication you want to monitor are running.
2. Open a new browser and enter the following URL:

   host:port/webProject_name/monitor

   Where:

   - *host*:*port* is the host and port you want to monitor. This can be the host and port of either the producer or consumer server.
   - *webProject_name* is the web project you want to monitor.

   For example:

   ```
   http://localhost:7001/wsrpMonitorTest/monitor
   ```

   The monitor appears in the browser. Click **Enable** to start monitoring. Click **Refresh** to see the latest transactions. Click **Clear** to remove all messages from the browser window.

Figure 14–3 Message Monitor Functions

Tip: The monitor does not display new transactions until you click **Refresh**.

Each time the remote portlet communicates with the producer, a request and response message headers appear on the monitor screen, as shown in Figure 14–4.

Figure 14–4 Monitor Appearing in a Browser

By clicking **Show**, you can display the content of the request or the response, as shown in Figure 14–5. Click **Hide** to close the message content.

Figure 14–5 Message Content

14.13.2 Creating Custom Logs

To create custom logs, we recommend that you use the Interceptor Framework described in Chapter 9, "The Interceptor Framework."

You can also create custom logs that display particular information about a WSRP session by using Logger and Handler objects instantiated by WebLogic Server. You can use these objects to create your own message handlers and subscribe them to loggers. For example, if you want the remote portlet to listen for the messages that the producer generates, you can create a handler and subscribe it to a logger in the producer. For detailed information on using Logger and Handler objects, see "Filtering

WebLogic Server Log Messages" in *Oracle Fusion Middleware Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*.

14.14 Managing Delivery of Headers and Cookies to the Browser

The WSRP 2.0 `clientAttributes` feature permits the producer to send cookies and headers to the client (browser) application. Before WSRP 2.0, cookies were retained and managed by the consumer and returned to the producer. The browser never received cookies directly. This limitation made it difficult to share cookies between different producers in the same domain. In addition, before WSRP 2.0, the consumer removed header information from the response before forwarding the response to the browser. This limitation restricted the use of certain browser operations, such as using the `content-disposition` header to allow the browser to display the "Save As" dialog box before opening a file when the resource was clicked inside the producer portlet.

14.14.1 Best Practice for Setting Cookies and Headers

The best practice for setting cookies and headers that you want to send to the client (browser) is to set them in the portlet's pre-render code. This practice is the best way to ensure that the cookie or header will reach the browser. For example, if you are using a portlet backing file, set the cookies or headers in the preRender() method. For information on backing files, see "Backing Files" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

Although not a best practice, if you want to set cookies or headers in the portlet's render code, you can set an option on the consumer to prevent the consumer from flushing responses until the buffer fills. While this can allow cookies and headers to reach the browser, more server resources are required to cache the entire response during rendering. If you want to use this feature, set the `<avoid-response-commit>` element to `true` in `WEB-INF/wlp-framework-common-config.xml`. See also "Avoiding Comitting Responses" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

This setting avoids committing the response until all the portlets have rendered or until the buffer fills. This behavior differs from the WLP default behavior, which is to stream responses without caching them.

> **Tip:** If the buffer fills and thereby begins to render before all the portlets' render() methods are called, the last portlet headers and cookies will be lost. That is why it is best practice to set headers and cookies in the preRender() method.

WLP always sends cookies and response headers from the consumer to the browser. However, as mandated by the WSRP 2.0 specification, anything represented elsewhere in the specification, such as `content-type`, `content-length`, `transfer-encoding`, and `user-agent`, are not sent. In addition, some headers that are not useful to transmit are not sent, like `accept-ranges`, `age`, and `proxy-authenticate`. The same is true for request headers from the client. To override this default behavior, the best option is to write an interceptor. For information on interceptors, see Chapter 9, "The Interceptor Framework."

14.14.2 Configuring Client Attribute Preferences on the Producer

WLP provides two producer-side settings that you can use to control the way client attributes (headers and cookies) are handled. Both of these settings can be configured in the `WEB-INF/wsrp-producer-config.xml` file on the producer.

- `isPreferClientCookies` – Sets whether or not the producer will use the cookie sent from the client (browser) or from the consumer in the event of a name collision. True = prefer from client (browser). False = prefer from consumer. (Default: false)
- `headerMode` – Sets where headers and cookies are sent. Possible settings are `Consumer`, `Client`, or `Both`. The default is `Both`. The `Client` setting specifies that headers and cookies set by the portlet will be directed to go to the client. (browser). The `Both` setting specifies that headers and cookies set by the portlet will be directed to go to the client and the consumer. If you do not want headers to be sent to the client (browser), set this attribute to `Consumer`.

  Tip: You can also set the header mode in Java portlets using a Container Runtime option: `com.oracle.portlet.wsrpHeaderMode`. For information on container runtime options, see "Using Container Runtime Options" in the *Oracle Fusion Middleware Portlet Development Guide for Oracle WebLogic Portal*.

14.14.3 Handling Cookies that Contain the Producer's Domain

If the browser receives a cookie with a domain attribute value that is different than the domain it connected to, the browser will reject the cookie. By default, WLP forwards all cookies from the producer to the browser without interfering in any way, except for the handling of the JSESSIONID cookie for security reasons (see Section 14.14.6, "Managing Security Between Consumer and Producer"). If you want to manipulate a cookie before it is sent from the consumer to the browser, you can do so by writing an interceptor. For details, see Chapter 9, "The Interceptor Framework."

14.14.4 URL/Path Rewriting of the Cookie Path

WSRP 2.0 does not define, and WLP does not support, the notion of rewriting URLs inside of cookies or headers.

14.14.5 Using Secure Cookies

If you use secure cookies, be sure that you are using a secure transport protocol (for example, HTTPS) between the consumer and the client (browser).

14.14.6 Managing Security Between Consumer and Producer

In most cases, the connection between browsers and consumers lies outside the network's DMZ. Likewise, the consumer to producer connections generally lie behind the network's DMZ. If you want to avoid the network overhead of using HTTPS behind the DMZ, you can use an insecure protocol (like HTTP) for the consumer to producer connection.

Caution: If you use an insecure connection between consumer and producer, the SOAP messages between the consumer and producer can contain cookies that are destined for the browser. If you intend these cookies to be secure, then it is recommended that you use a secure channel between consumer and producer. For more information, see Section 14.8, "Security for Remote Portlets."

For more information, see Section 14.8, "Security for Remote Portlets."

For security reasons, the JSESSIONID cookie from the producer is not sent to the browser. In addition, if a cookie is set as `isSecure`, the cookie is sent to the browser only if the communication between client and consumer is secure. Note that it is possible to receive a secure cookie on the browser even though the producer and consumer communication may be on an insecure protocol. This scenario allows a consumer and producer that are both behind a company firewall in the same domain to communicate and still issue secure cookies to the client (browser).

14.15 Configuring Session Cookies

This section describes techniques for preventing the loss of the consumer session when resource requests are made to a remote portlet. These techniques include:

- Section 14.15.1, "Using Different Cookie Names"
- Section 14.15.2, "Using a System Property"
- Section 14.15.3, "Blocking Cookies"

14.15.1 Using Different Cookie Names

If you have a remote portlet that contains images, WebLogic Portal sends cookies and other headers from the producer to the browser when an image resource is requested. Note that when resource requests are made to a portlet in a producer, it is possible for the user's browser to drop or lose the consumer session. This situation occurs when the producer and consumer are configured to include only the default path ("/") in the session cookies, which causes the browser to replace the `Set-Cookie` header set by the consumer with the `Set-Cookie` header set by the producer.

To prevent this potential loss of the consumer session, open `weblogic.xml`, and configure your web applications to include the domain name and web application path for session cookies. This technique prevents the cookie names from overlapping. See "session-descriptor" in *Oracle Fusion Middleware Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server* for details on how to set the domain name and path.

14.15.2 Using a System Property

In most cases, using different cookie names solves the problem of lost consumer sessions following resource requests. In some cases, however, this solution does not work. One such use case is when single sign-on is used with the producer and consumer running in the same domain. In this case, identical cookie names are required. For cases where using different cookie names does not work, set the following system property:

```
wlp.resource.proxy.servlet.block.response.headers=true
```

By enabling this system property, WebLogic Portal prevents a `Set-Cookie` header from being sent back to the user's browser. This property prevents the consumer's cookie from being overwritten by the producer's cookie on the browser when a resource is returned. Using this technique, you can keep the cookie names the same for both the producer and consumer applications, which is required for single sign-on.

14.15.3 Blocking Cookies

To block cookies to the browser, set `<resource-cookies>` to `block-all` in `WEB-INF/wsrp-producer-registry.xml` in the consumer web application, as shown in Example 14–4. When this element is set to `block-all`, the resource proxy servlet does not transfer any cookies from the producer resource to the browser. Cookies are not blocked by default. The default setting is `block-none`.

Example 14–4 Blocking Cookies to the Browser

```
<wsrp-producer-registry
   xmlns="http://www.bea.com/servers/weblogic/wsrp-producer-registry/8.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/servers/weblogic/wsrp-producer-
      registry/8.0 wsrp-producer-registry.xsd">
   <!-- Upload limit (in bytes) -->
   <upload-read-limit>1048576</upload-read-limit>
   <!-- Timeout (in milli seconds) -->
   <connection-timeout-secs>120000</connection-timeout-secs>
   <!-- Enable local proxy -->
   <enable-local-proxy>true</enable-local-proxy>
   <!-- Block cookies to the browser -->
   <resource-cookies>block-all</resource-cookies>
...
</wsrp-producer-registry>
```

14.16 User Sessions on CWEB Applications

User sessions on CWEB applications might be lost if session cookies between producers and consumers overlap. To prevent this, open `weblogic.xml`, configure your web applications to include the domain name and web application path for session cookies. See "session-descriptor" in *Oracle Fusion Middleware Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server* for details on how to set the domain name and path.

14.17 Using Multiple Views with Remote Portlets

Whenever multiple views of a remote portlet are created, links in the portlets can be inconsistent and not work properly. Typically, multiple views occur when a remote portlet uses the popup mechanism in a page flow, or when a user floats a remote portlet using the portlet Float button.

Note: Page flows are a feature of Apache Beehive, which is an optional framework that you can integrate with WLP. See "Apache Beehive and Apache Struts Supported Configurations" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

If a WebLogic Portal producer is set up to use consumer-supplied URL templates, the producer caches those templates in a session created on the producer. However, when multiple views of a portlet are created either through the page flow popup mechanism, or through a Float button, the cached templates may not be valid for the current view.

You can correct the inconsistent links using one of these methods:

- Disabling the caching of templates for your remote portlets. To do this, in the `.portlet` file for each remote portlet that is affected, change the value of the `templatesStoredInSession` element to `false`.
- Configure the producer to require consumer rewriting. To do this, set the `rewrite-urls` element to FALSE in the `wsrp-producer-config.xml` file.

14.18 Handling User Identity Changes

If the user's identity changes while a request generated from the portal is in progress, remote portlets can behave inconsistently. Typically, this occurs when the portal desktop includes a portlet or other mechanism for logging in and logging out a user. If the user identity changes, any user-specific data loaded by the portal can become invalid. In the case of remote portlets, such data includes their persistent state. When user identity changes, the consumer portal can send incorrect persistent state data to producers.

To avoid this problem, be sure to always use a browser redirect call immediately after a login or logout. The browser redirect ensures that data loaded by the portal is valid for the request.

14.19 Storing Registration Properties

This section discusses the Store Registration Properties feature and why enabling it is generally recommended.

When you register a producer either using Oracle Enterprise Pack for Eclipse or the WebLogic Portal Administration Console, you have the option of storing registration property sets on the consumer. Registration properties are values that are passed from the consumer to the producer when the producer is registered. These values can be used to allow producers to control which portlets are offered to specific consumers.

> **Tip:** For detailed information on consumer entitlement and creating registration property sets, see Chapter 11, "Consumer Entitlement."

This section explains why storing registration properties is the recommended procedure for storing registration properties using both Oracle Enterprise Pack for Eclipse and the Administration Console. This section includes these topics:

- Section 14.19.1, "Why Store Registration Properties?"
- Section 14.19.2, "Using the Administration Console"
- Section 14.19.3, "Using Oracle Enterprise Pack for Eclipse"

14.19.1 Why Store Registration Properties?

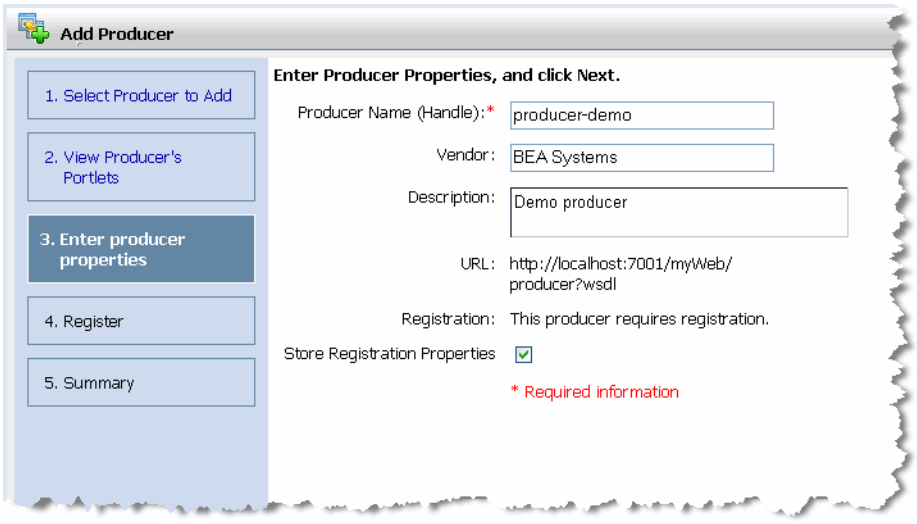
It is recommended that you choose to store registration properties when you register a producer. This option provides these advantages:

- If the producer is unable to provide registration properties, they are still available to your portal application. The producer may be unable to provide registration properties if:
  - The producer is not a WebLogic Portal 10.2 or later version. The WSRP extension for providing registration properties from the producer does not work with WebLogic Portal 10.0 or older versions or with third-party producers.
  - The producer is configured not to offer support for the WSRP registration property extension. Typically, the a producer is configured in this way to prevent sending registration property values back from the producer as a security measure. In this case, you might consider it to be acceptable to store the registration property values on the consumer and retain the benefits of registration property storage.
  - The producer is temporarily unavailable.
- If you choose to modify the registration properties for a producer, the Modify Registration Properties dialog box will always be filled in with the currently used registration properties. The procedure for modifying registration properties is described in Section 20.2, "Modifying Producer Registration Properties".

14.19.2 Using the Administration Console

The Store Registration Properties check box is provided in the Enter Producer Properties dialog of the Add Producer Wizard, as shown in Figure 14–6. (The complete procedure for registering producers is discussed in detail in Chapter 18, "Adding Remote Resources to the Library.")

Figure 14–6 Store Registration Properties Option



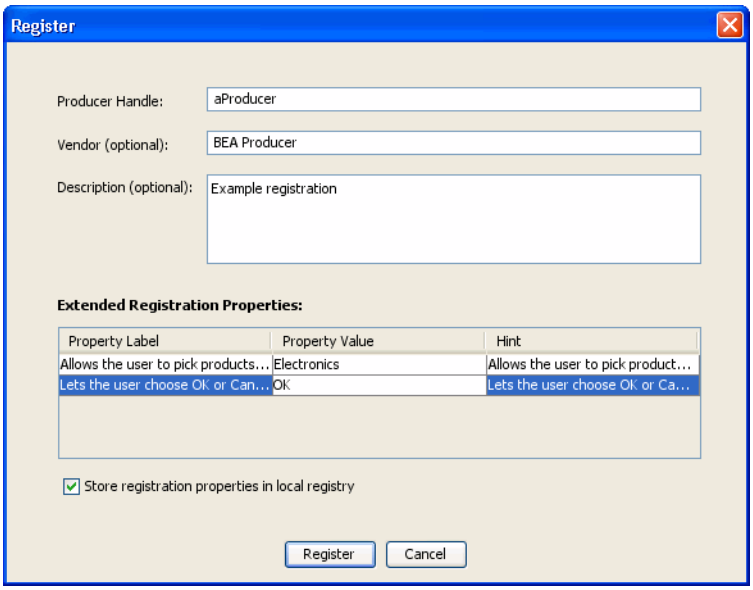
You can change the value of the Store Registration Properties check box in the WebLogic Portal Administration Console. Go to the Summary tab of the producer and click **Producer Properties** to bring up the Update Producer Url dialog. This dialog lets you change the setting.

14.19.3 Using Oracle Enterprise Pack for Eclipse

The **Store registration properties in local registry** check box is provided in the Register dialog of the Remote Portlet Wizard, as shown in Figure 14–7.

> **Tip:** You can also store registration properties for remote books and pages. See Chapter 4, "Creating Remote Portlets, Pages, and Books" for detailed information in the wizard used for creating remote portlets, pages, and books.

Figure 14–7 Storing Registration Properties



When **Store registration properties in local registry** is checked, the `wsrp-producer-registry.xml` file is updated with the stored registration information. A sample is shown in Example 14–5. To provide flexibility, this option is always selected by default even if there are no properties defined. If the checkbox is selected during registration you can add registration properties later.

You can only edit the registration properties using the IDE dialog when initially registering or re-registering a producer. To change the properties after the producer is registered, you need to edit the XML file directly using an XML editor.

Example 14–5 Registration Property Information

```
...
<registration-properties>
      <stringProperty name="{urn:bea:wlp:prop:reg:propset-1}Selection">
        <value>OK</value>
      </stringProperty>
      <stringProperty name="{urn:bea:wlp:prop:reg:propset-1}Choices">
        <value>Electronics</value>
      </stringProperty>
    </registration-properties>
    <store-registration-properties>true</store-registration-properties>
...
```

14.20 Editing the WSRP WSDL Template File

You can customize the producer-generated WSDL. For example, you might want the WSDL to point to a proxy server other than the default one. To customize the default WSDL, you can edit the `WEB-INF/beehive-url-template-config.xml` file. The easiest way to edit this file is to copy it to your workspace in Oracle Enterprise Pack for Eclipse. To do this, locate the file in the Merged Projects view in Oracle Enterprise Pack for Eclipse. Right-click the file and select **Copy to Workspace**. The template file uses URL templates. See Javadoc for the GenericURL class for information on configuring URL templates.

14.21 Configuring a Custom JAX-RPC Handler

This section explains how to configure custom JAX-RPC handlers on the WSRP consumer or producer. Custom handlers can be used to intercept and process the outbound SOAP requests and inbound SOAP responses. For example, handlers can inspect the incoming and outgoing messages, change the messages before they make it to the end point, log information, and so on. This section only explains how to configure and register a handler, not how to write a handler class.

> **Tip:** The handler class must implement the javax.xml.rpc.handler.Handler interface or extend javax.xml.rpc.handler.GenericHandler.

This section includes these topics:

- Section 14.21.1, "Configuring a Handler on the Consumer"
- Section 14.21.2, "Configuring a Handler on the Producer"

14.21.1 Configuring a Handler on the Consumer

Edit the file `WEB-INF/wsrp-consumer-handler-config.xml` to add a custom handler the consumer. The easiest way to edit this file is to copy it to your workspace in Oracle Enterprise Pack for Eclipse. To do this, locate the file in the Merged Projects view in Oracle Enterprise Pack for Eclipse. Right-click the file and select **Copy to Workspace**.

Example 14–6 shows an example configuration file:

Example 14–6 Event Handler Configuration

```
<wsrp-consumer-handler-config
xmlns="http://www.bea.com/ns/portal/90/wsrp-consumer-handler-config">
    <description>Description goes here</description>

    <soap-handler>
        <name>ProducerHandlesFilter</name>
        <description>Producer Handles Filter test handler</description>

        <!-- List of producer handles to deploy to, if none are specified ALL
             producers will have this handler -->
        <producer-handle>NoOpProducer1</producer-handle>

<handler-class>com.bea.wsrp.qa.consumer.handlers.ProducerHandlesFilter
</handler-class>

        <!-- If true, the handler will run before the SAML token is added. -->
```

```
        <pre-security>true</pre-security>

        <!-- init parameters if needed -->
         <init-parameter>
            <name>param1</name>
            <value>value1</value>
        </init-parameter>
        <init-parameter>
            <name>param2</name>
            <value>value2</value>
        </init-parameter>


        <!-- Specify which ports to add the handler to. -->
        <port-name
xmlns:wsrp="urn:oasis:names:tc:wsrp:v1:wsdl">wsrp:WSRPServiceDescriptionService
</port-name>
        <port-name
xmlns:wsrp="urn:oasis:names:tc:wsrp:v1:wsdl">wsrp:WSRPBaseService</port-name>
        <port-name
xmlns:wsrp="urn:oasis:names:tc:wsrp:v1:wsdl">wsrp:WSRPRegistrationService
</port-name>
        <port-name
xmlns:wsrp="urn:oasis:names:tc:wsrp:v1:wsdl">wsrp:WSRPPortletManagementService
</port-name>
        <port-name
xmlns:wsrp="urn:oasis:names:tc:wsrp:v1:wsdl">wsrp:WLP_WSRP_Ext_Service
</port-name>

        <soap-role>someRoleHere</soap-role>
    </soap-handler>
```

14.21.2 Configuring a Handler on the Producer

On the producer edit the file `WEB-INF/webservices.xml` as defined by the JAX-RPC specification.

Part III

Staging

In the staging phase of the portal life cycle, you use the WebLogic Portal Administration Console to create portal desktops, manage users and groups, and perform other administration tasks. You can add producers and consume portlets, books, and pages that are deployed in producers. In a staging environment, you build and test all of your portal's components before moving the portal to production.

If you are developing federated portals, you perform most of the security configuration in the staging environment using the WebLogic Portal Administration Console and WebLogic Server Administration Console.

For a detailed description of the staging phase of the portal life cycle, see the *Oracle Fusion Middleware Overview for Oracle WebLogic Portal*.

Part III contains the following chapters:

15

Establishing WSRP Security with SAML

This chapter discusses how to configure the security realms of WebLogic Portal producers and consumers running in different domains. In the first part of this chapter, we explain the configuration that is required when both the producer and consumer are running in WebLogic Portal version 9.2 or later domains. In the second part of this chapter, the case of mixed domains is discussed, where the producer and consumer can be running in either WebLogic Portal 8.1x or 9.2 or later domains.

This chapter includes the following sections:

15.1 SAML Security Between WebLogic Portal Domains

This section explains the procedure for configuring WSRP security using custom SAML tokens when the producer and consumer are running in different WebLogic Portal version 9.2 or later domains. Use the procedure described in this section to configure security on production systems where custom SAML tokens are required.

Note: By default, with no previous configuration, WebLogic Portal 9.2 and later domains share a common key. This allows you to quickly create, for demonstration or testing purposes, federated portals that require user authentication without undergoing the configuration procedure described in this section.

Caution: It is recommended that you do not use the default key described in the previous note in a production environment. Using this default setting allows any consumer to connect to your producer.

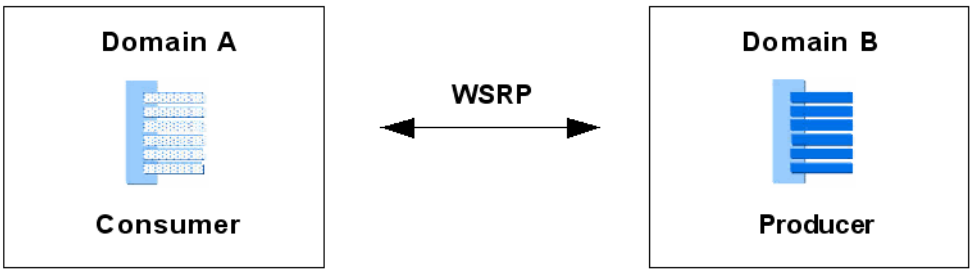
This section includes these topics:

- Section 15.1.4, "Configuring the Producer"
- Section 15.1.5, "Testing the Configuration"

15.1.1 Overview

In a typical scenario, consumer and producer applications are running in separate WebLogic Portal 9.2 or later domains, as shown in Figure 15–1.

Figure 15–1 Basic Use Case



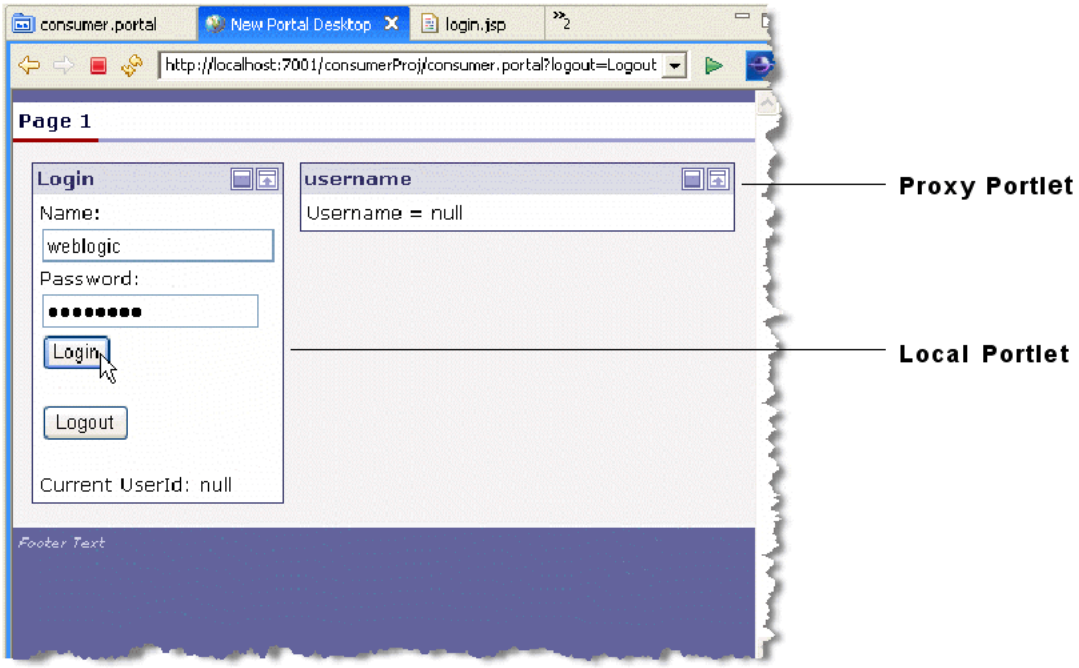
By default, WebLogic Portal specifies SAML as the default security token type for WSRP. If you are running in a demonstration or development environment, no further configuration is required. However, for a production environment it is recommended that you perform the SAML configuration described in this section.

15.1.2 Setting Up the SAML Configuration Example

This section describes an example federated portal application where the producer and consumer are running in different WebLogic Portal 9.2 or later domains. This example provides a basis for discussing how to configure SAML security between these domains.

The portal shown in Figure 15–2 includes a local portlet on the left and a remote (proxy) portlet on the right. The local portlet is a login portlet. When a user logs in successfully, the producer portlet displays the user's name. As you will see, however, unless SAML security is properly configured, an error results when a user logs in to the portal.

Figure 15–2 Consumer Portal Before User Login



As you can see in Figure 15–2, the proxy portlet renders without error before a user logs in. It isn't until a user attempts to log in that a SAML message is sent, resulting in an error.

Checkpoint: This section described an example federated portal where the consumer and producer are running in separate WebLogic Portal domains. In the following sections, we explain how to configure the consumer and producer so that the SAML token sent from the consumer is accepted by the producer.

15.1.3 Configuring the Consumer

To correct the error shown in the previous section, you need to configure both the consumer and the producer. This section discusses the consumer configuration.

15.1.3.1 Generate a Key

This section explains how to generate a key on the consumer using the keytool utility, a Java utility distributed by Sun Microsystems that manages private keys and certificates. For detailed information on keytool, refer to the Sun Microsystems website.

> **Tip:** Anytime you generate a new key, you must copy the keystore to the entire cluster.

> **Note:** By default, the consumer has a keystore that the server uses for its SSL key. The default keystore is called DemoIdentity.jks. If you are using a different keystore, then modify the one that you are currently using.

1. On the consumer, open a command window and CD to the *<WEBLOGIC_HOME>*`/server/lib` directory.

2. Run the keytool command to generate a new key, as shown in Figure 15–3. For example, the following command generates a key called `testalias`.

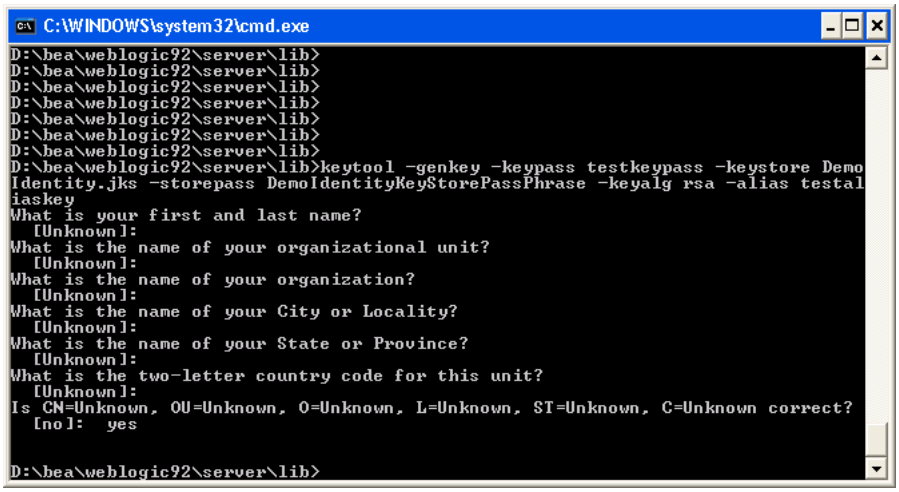
   ```
   keytool -genkey -keypass testkeypass -keystore
   DemoIdentity.jks -storepass DemoIdentityKeyStorePassPhrase
   -keyalg rsa -alias testalias
   ```

   The options used in the example keytool command include the following:

Table 15–1 Keytool Options

Command Parameter	Description
keytool	A Java tool used to generate a key.
-genkey	Instructs the keytool to generate a key.
-keypass	Specifies the password to be used with the new key.
-keystore	Specifies the name of the keystore. A keystore stores keys and certificates. The default keystore, `DemoIdentity.jks`, is implemented as a file that protects private keys with a password.
-storepass	Specifies the password for the keystore.
-keyalg	Specifies the encryption algorithm for the keystore. You must use `rsa` for the key algorithm. If you use another algorithm, you will receive an error when the consumer sends a SAML message.
-alias	Specifies a name for the generated key.

Figure 15–3 Generating a Key

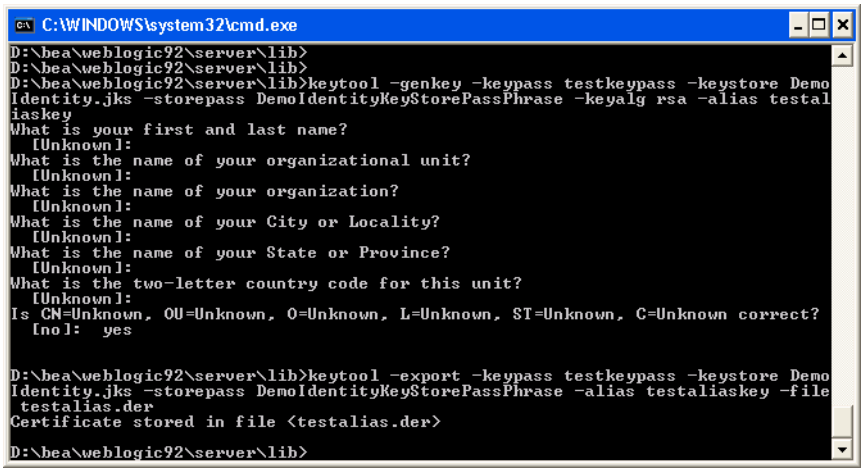


15.1.3.2 Export the Key

Export the key from the consumer server. In the same command window that you used to generate the key, in the same directory, run the keytool command with the `-export` option, as shown in Figure 15–4. For example, the following command generates a key file called `testalias.der`.

```
keytool -export -keypass testkeypass -keystore DemoIdentity.jks -storepass
DemoIdentityKeyStorePassPhrase -alias testalias -file testalias.der
```

Figure 15–4 Exporting the Certificate



15.1.3.3 Modify the Consumer's Security Realm

This section explains the procedure for configuring the consumer to use the key that you generated.

> **Tip:** A security realm is a container for the mechanisms–including users, groups, security roles, security policies, and security providers–that are used to protect WebLogic resources. You can have multiple security realms in a WebLogic Server domain, but only one can be set as the default (active) realm. The default is called myrealm.

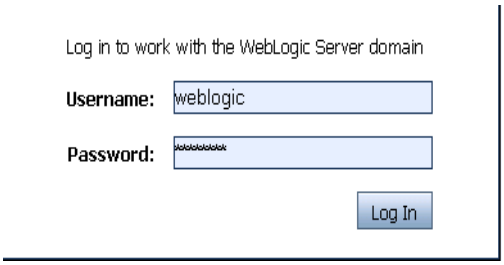
1. Log in to the WebLogic Server Administration Console on the consumer. To do this, open a browser and enter the following URL:

   `http://`*`serverIP`*`:`*`port`*`/console`

   where *`serverIP`* is the IP address of the server on which the consumer application is running, and *`port`* is the server's port number. For example:

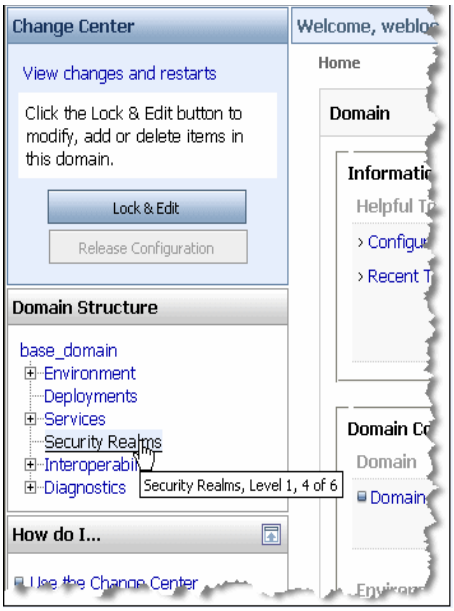
   `http://localhost:7001/console`

Figure 15–5 WebLogic Server Administration Console Login Dialog



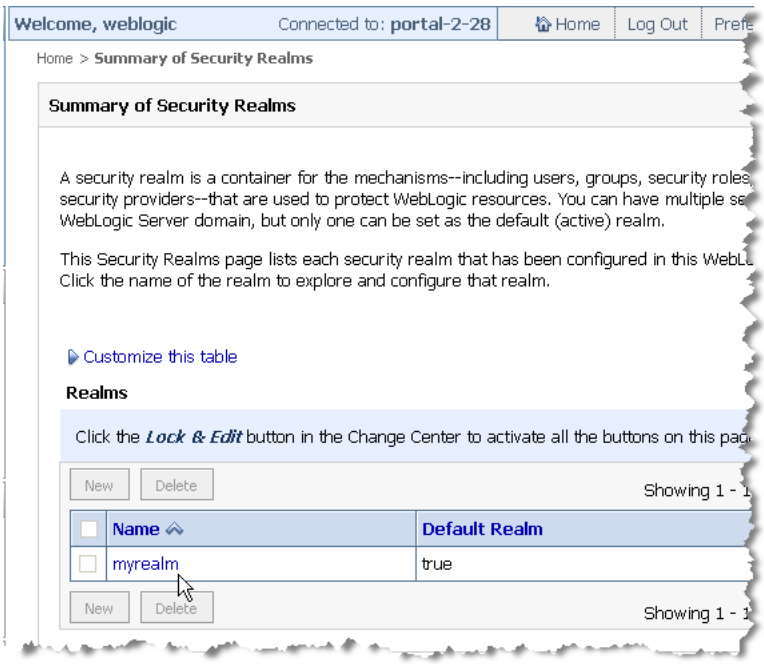
2. In the Administration Console, select **Security Realms** in the Domain Structure tree, as shown in Figure 15–6.

Figure 15–6 Selecting Security Realms



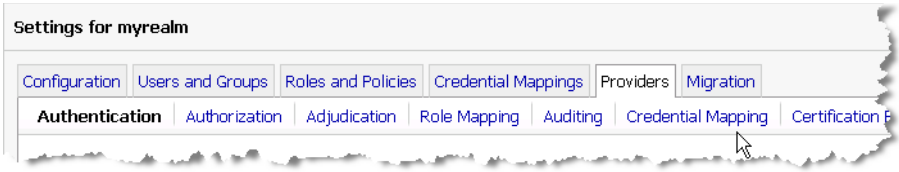
3. Select a security realm. The default security realm is called `myrealm`, as shown in Figure 15–7.

Figure 15–7 Selecting a Security Realm



4. Select the **Providers** tab and then the **Credential Mapping** tab, as shown in Figure 15–8.

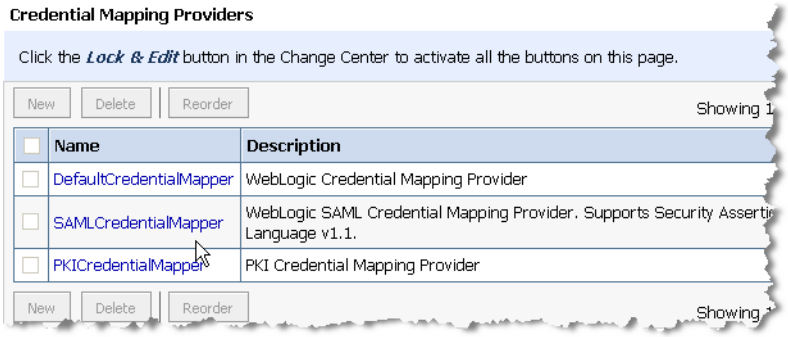
Figure 15–8 Selecting the Credential Mapping Tab



5. Select **SAMLCredentialMapper**, as shown in Figure 15–9.

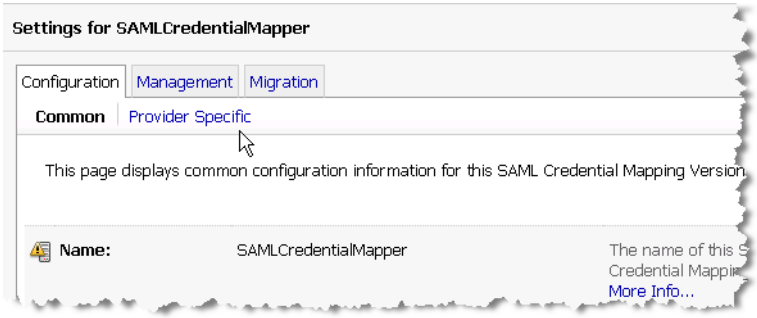
   Tip: The SAML Credential Mapper provider acts as a producer of SAML security assertions, allowing WebLogic Server to act as a source site for using SAML for single sign-on.

Figure 15–9 Selecting the SAMLCredentialMapper



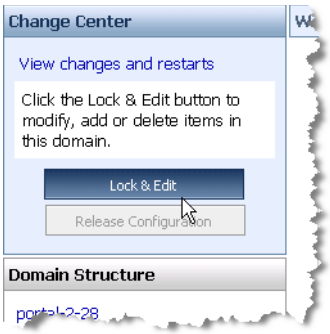
6. Select **Provider Specific**, as shown in Figure 15–10.

Figure 15–10 Selecting the Provider Specific Tab



7. In the Change Center window, select **Lock and Edit**, as shown in Figure 15–11. This function blocks other users from making Administration Console changes and enables you to edit the SAMLCredentialMapper settings.

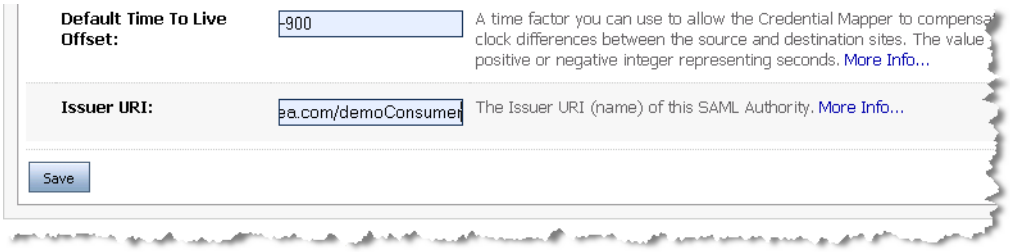
Figure 15–11 Locking the Console



8. Edit the Issuer URI, as shown in Figure 15–12. This unique URI tells the producer the origin of the SAML message and allows the producer to match the consumer with the key. Typically, the consumer's URL is used in this string to guarantee that it is unique. For example:

   ```
   http://www.bea.com/demoConsumer
   ```

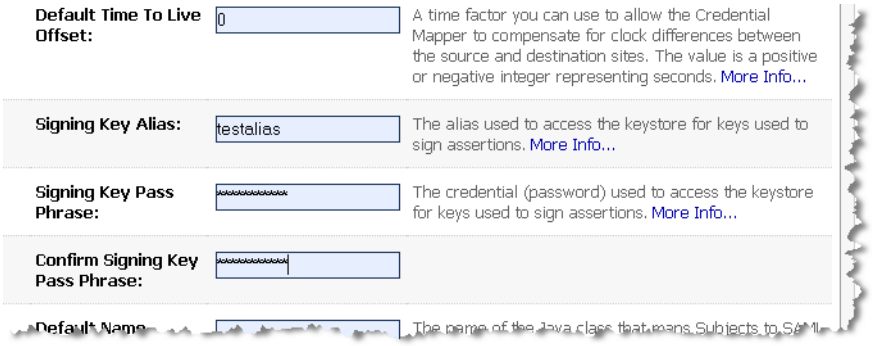
Figure 15–12 Issuer URI



9. Enter the Signing Key Alias and Signing Key Pass Phrase, as shown in Figure 15–13. These are the values you used when you generated the keystore. In this example they were:

Provider Field	Value
Signing Key Alias	`testalias`
Signing Key Pass Phrase	`testkeypass`

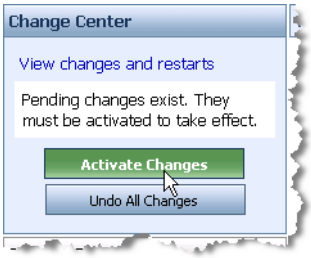
Figure 15–13 Additional Provider Fields



Tip: It is recommended that you set the Default Time To Live to 120 seconds, the Cred Cache Min Viable TTL to 10 seconds, and the Default Time To Live Offset to 0. Then, select the **Management** tab and in the relying party configuration, set the Assertion Time To Live Offset to the difference between the clock times of the consumer and producer (consumer time minus producer time).

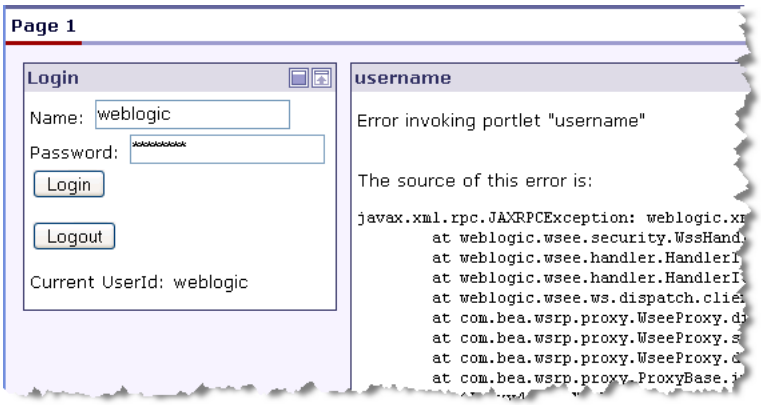
10. Click **Save**.
11. In the Change Center window, click **Activate Changes**, as shown in Figure 15–14.

Figure 15–14 Activating Changes



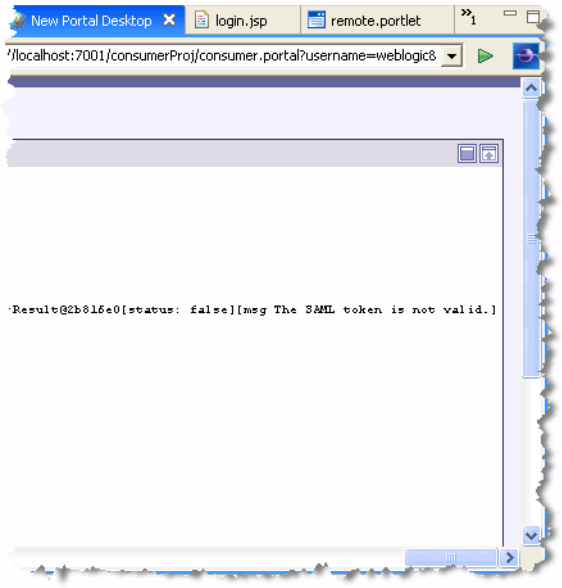
Checkpoint: At this point, the SAML credential mapper provider on the consumer is configured to use the keystore you generated. If you were to try to log in to the login portlet, you would receive an error, as shown in Figure 15–15. This is because the producer does not yet recognize the new key sent from the consumer. In the next steps, you will configure the producer to accept the key sent from the consumer.

Figure 15–15 Login Results in an Error in the Producer Portlet



Tip: If you scroll the portal to the right, you will see that the error message says "The SAML token is not valid," as shown in Figure 15–16.

Figure 15–16 Error Message



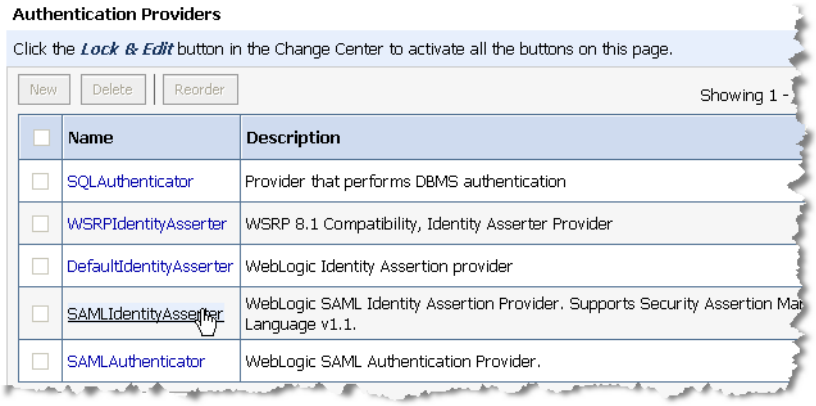
15.1.4 Configuring the Producer

This section explains how to configure the producer. To do this, you import the certificate into the SAML asserter, and configure the asserting party properties.

15.1.4.1 Import the Certificate

1. Copy the key file (`testailias.der`) that you generated on the consumer to the producer using any method you want, such as FTP or SMB. You can place the file in any directory on the destination machine.
2. Open the WebLogic Server Administration Console on the producer machine and log in.
3. Select **Security Realms**.
4. Select a security realm, such as **myrealm**.
5. Select the **Providers** tab.
6. Select the **Authentication** tab.
7. Select **SAMLIdentityAsserter**, as shown in Figure 15–17. An identity asserter allows WebLogic Server to establish trust by validating a user.

Figure 15–17 Selecting the Identity Asserter

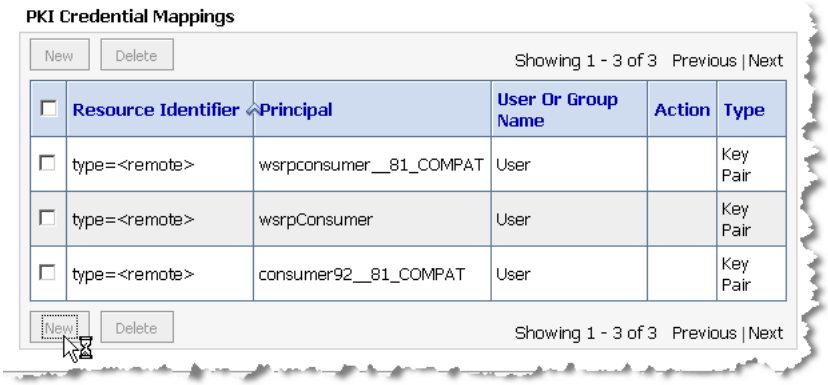


8. Click the **Management** tab, and click **Certificates**, as shown in Figure 15–18.

Figure 15–18 Selecting the Certificates Tab

9. In the Certificates dialog, click **New**, as shown in Figure 15–19.

Figure 15–19 Creating a New Certificate



10. In the Alias field, enter a name for the certificate, as shown in Figure 15–20. It is a good practice to use the same name you used when you created the certificate. In this example, the name of the alias is `testalias`.
11. In the Certificate File Name field, enter the path to the certificate file, as shown in Figure 15–20.

Figure 15–20 Entering Certificate Properties

12. Click **Finish**. If there are no problems, the following message is displayed:

    The certificate has been successfully registered.

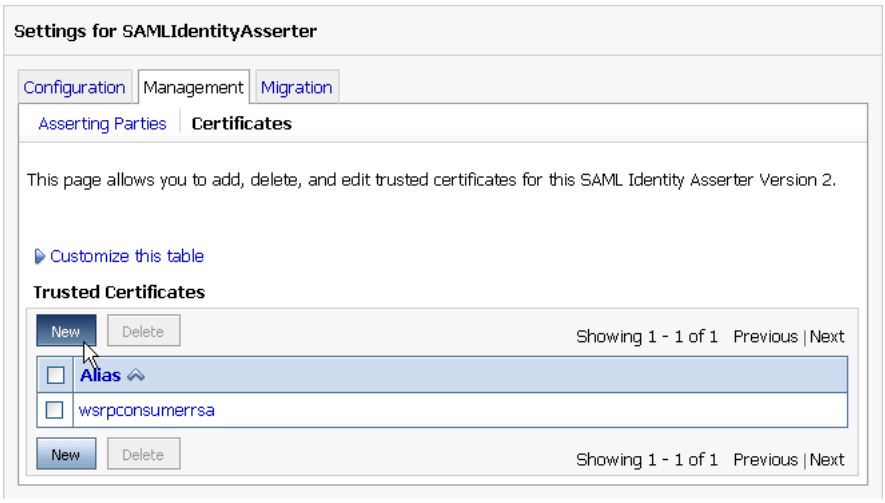
15.1.4.2 Configure the Asserting Party Properties

1. On the Management tab, click **Asserting Parties**.

   Tip: The WsrpDefault asserting party is set up for the producer's default WSRP key. If the consumer and producer applications were running on the same server, the WSRP key of the consumer would be accepted by the producer. It is a good practice to delete the WsrpDefault party for an application that is in production.

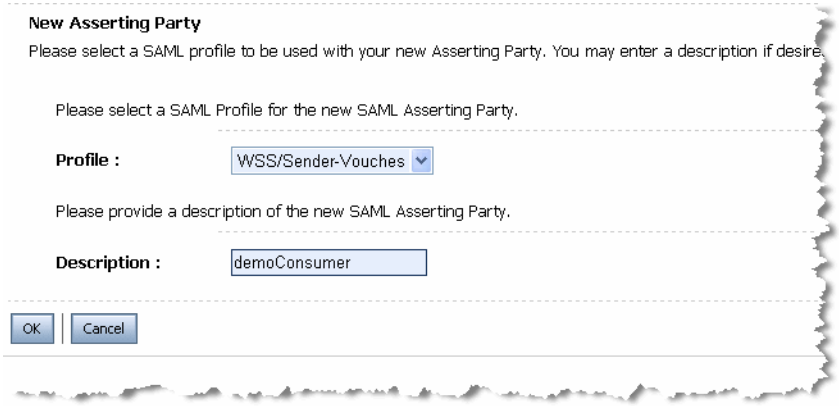
2. In the Asserting Parties table, click **New**, as shown in Figure 15–21.

Figure 15–21 Creating a New Asserting Party



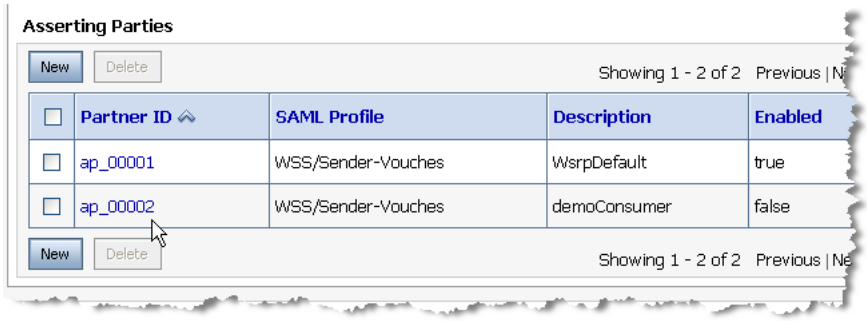
3. In the Profile dropdown menu, select **WSS/Sender Vouches**, as shown in Figure 15–22.
4. In the Description field, enter a name to identify the asserting party, as shown in Figure 15–22. For example: `demoConsumer`.

Figure 15–22 Asserting Party Properties



5. Enable the new asserting party. To do this, click the **Partner ID** link for the asserting party. In this example, the link is ap_0002 for the asserting party called demoConsumer, as shown in Figure 15–23.

Figure 15–23 Selecting the New Asserting Party

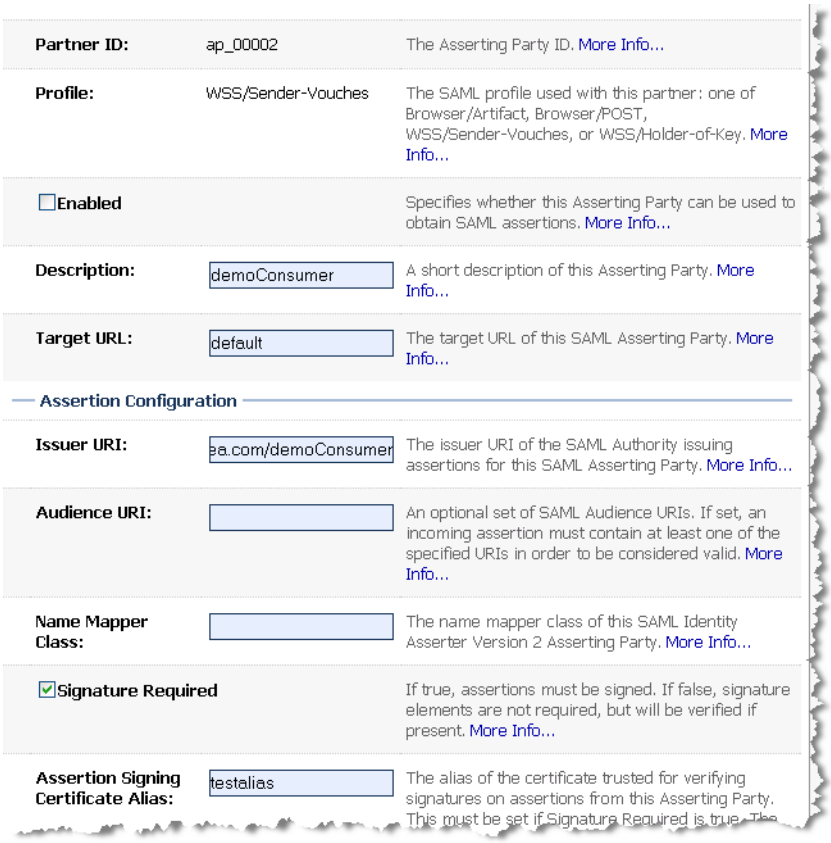


6. Set the asserting party values, as listed in Table 15–1 and shown in Figure 15–24.

Table 15–2 Asserting Party Values

Parameter	Value
Enabled	Select the checkbox (true).
Target URL	`default`
Issurer URI	Use the same one that you configured on the consumer. In this example, it is `http://bea.com/demoConsumer`
Signature Required	Select the checkbox (true).
Assertion Signing Certificate Alias	Use the same one that you configured on the consumer. In this example it is `testalias`.

Figure 15–24 Asserting Party Values



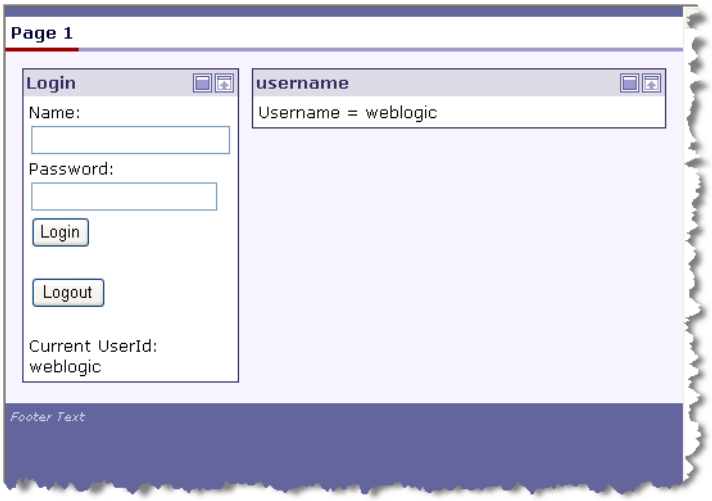
7. Click **Save**.

   If there were no problems, the message, Settings updated successfully, appears.

15.1.5 Testing the Configuration

On the consumer, log into the portal application with a valid user name and password. You will see the user name appear in the proxy portlet. This indicates that the SAML message was passed from the consumer to the producer, and that the producer recognized and accepted it.

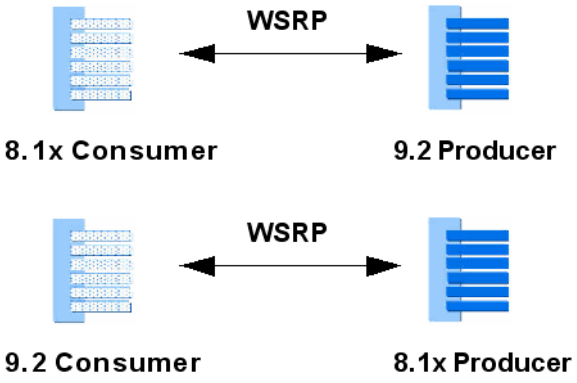
Figure 15–25 Successful Test



15.2 SAML Security Between WebLogic Portal 8.1x and 9.2 or Later Versions

Producer and consumer applications developed with WebLogic Portal 9.2 or later versions are compatible with producers and consumers developed with WebLogic Portal 8.1x. That is, a portal developed with WebLogic Portal 9.2 or later versions can consume portlets deployed in a WebLogic Portal 8.1x domain. Similarly, portlets exposed in a WebLogic Portal 9.2 or later producer can be consumed by an 8.1x consumer. These two use cases are summarized in Figure 15–26.

Figure 15–26 Compatibility Use Cases



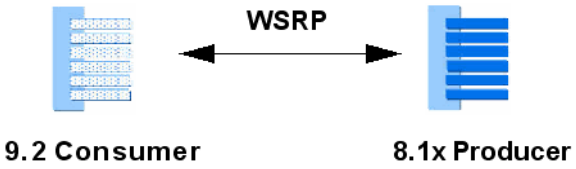
This section discusses addresses both of these use cases. The following topics are discussed:

- Section 15.2.1, "SAML Security Between 9.2 or Later Version Consumers and 8.1x Producers"
- Section 15.2.2, "SAML Security Between 8.1x Consumers and 9.2 or Later Version Producers"

15.2.1 SAML Security Between 9.2 or Later Version Consumers and 8.1x Producers

This section explains how to achieve SAML-based security compatibility between a WebLogic Portal 9.2 or later version consumer and an 8.1x producer, as summarized in Figure 15–27.

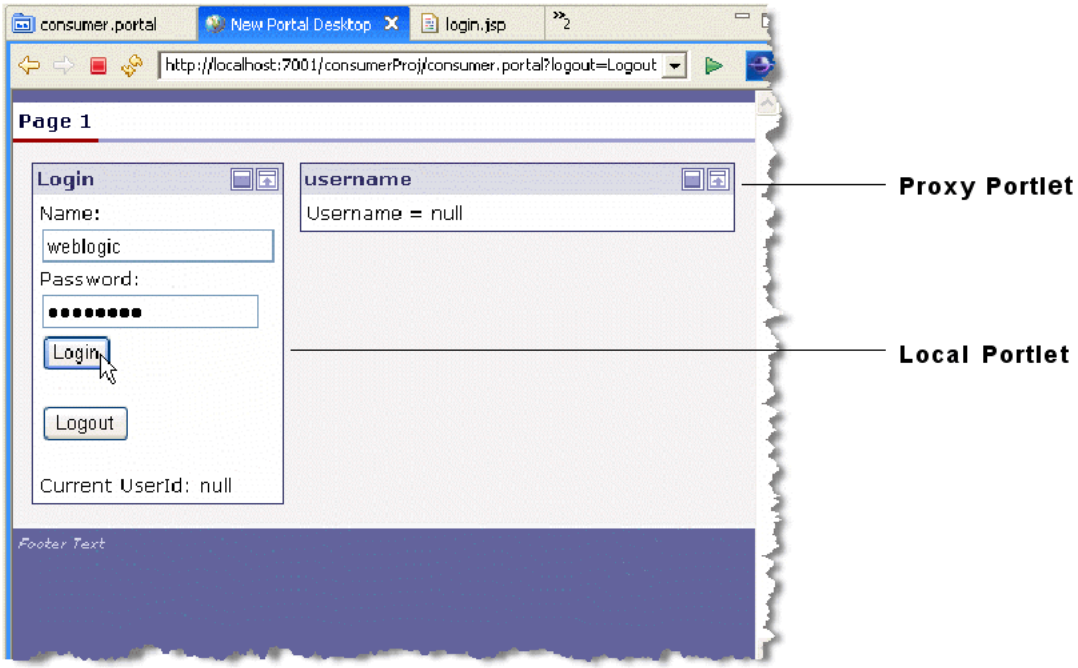
Figure 15–27 Compatibility Use Case



Tip: By default, with no configuration changes made to either side, WSRP between a 9.2 or later version consumer and 8.1x producer works. That is, a 9.2 or later version consumer can consume a portlet from an 8.1x producer with no configuration changes. However, if you want to use your own key for the 9.2 or later version consumer, you need to follow the procedure outlined in this section

The portal shown in Figure 15–28 includes a local portlet on the left and a remote (proxy) portlet on the right. The remote portlet is deployed in an 8.1x producer. The local portlet is a login portlet. Before SAML security is properly configured, when a user logs in, the name that is returned is null.

Figure 15–28 Consumer Portal Before User Login



15.2.1.1 Configuring the Consumer

The following sections explain how to configure the consumer with a key that can sign the SAML assertion sent to the producer. The basic tasks include:

- Generating a key

- Changing the consumer's name
- Modifying the consumer's security realm

15.2.1.1.1 Generate a Key This section explains how to generate a key on the consumer using the keytool utility, a Java utility distributed by Sun Microsystems that manages private keys and certificates. For detailed information on keytool, refer to the Sun Microsystems web site.

> **Note:** WebLogic Portal is configured with a default identity keystore (`wsrpKeystore.jks`). This default keystore configuration is appropriate for testing and development purposes. However, it should not be used in a production environment. For more information, see the WebLogic Server security documentation.

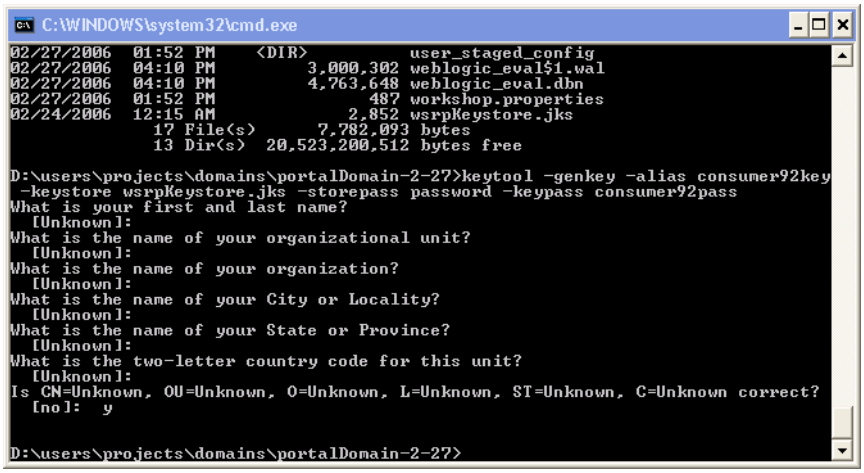
1. On the consumer, open a command window and CD to the *<WEBLOGIC_HOME>*`/server/lib` directory.
2. Run the keytool command to generate a new key, as shown in Figure 15–29. For example, the following command generates a key called `consumer92key`.

   ```
   keytool -genkey -alias consumer92key -keystore
   wsrpKeystore.jks -storepass password -keypass consumer92pass
   ```

   The options used in the example keytool command include the following:

Command parameter	Description
keytool	A Java tool used to generate a key.
-genkey	Instructs the keytool to generate a key.
-alias	Specifies a name for the generated key.
-keystore	Specifies the name of the keystore. A keystore stores keys and certificates. The default keystore, `wsrpKeystore.jks`, is implemented as a file that protects private keys with a password.
-storepass	Specifies the password for the keystore.
-keypass	Specifies the password to be used with the new key.

Figure 15–29 Generating a Key



15.2.1.2 Change the Consumer's Name

1. Copy the `wsrp-consumer-security-config.xml` from the J2EE Shared Library to your project. To do this in Oracle Enterprise Pack for Eclipse, open the Merged Projects view, find the file in the WEB-INF directory of your consumer web application. Right-click the file and select **Copy to Project**. For more information on copying files from J2EE Shared Libraries, see the *Oracle Fusion Middleware Production Operations Guide for Oracle WebLogic Portal* and the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

2. Edit the file `wsrp-consumer-security-config.xml` in the `WEB-INF` directory of your consumer web application. Change the `<consumer-name>` element from `wsrpConsumer` to another arbitrary name. For example, change:

   ```
   <consumer-name>wsrpConsumer<consumer-name>
   ```

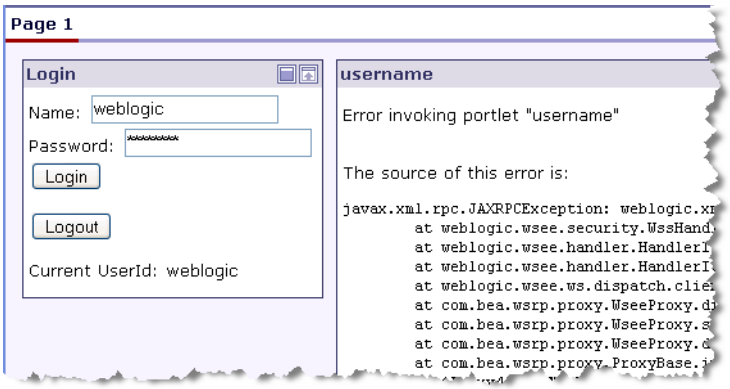
   to

   ```
   <consumer-name>consumer9x<consumer-name>
   ```

3. Restart the consumer application's server so that the change to the configuration file takes effect.

   Checkpoint: If you try to log in to the remote portlet again, you will receive an error, as shown in Figure 15–30. This error is caused by the fact that the producer cannot find the key that was sent from the consumer. The next step is to configure the security realm for the consumer domain.

Figure 15–30 Login Error



15.2.1.3 Modify the Consumer's Security Realm

1. Log in to the WebLogic Server Administration Console on the consumer. The URL for the console is:

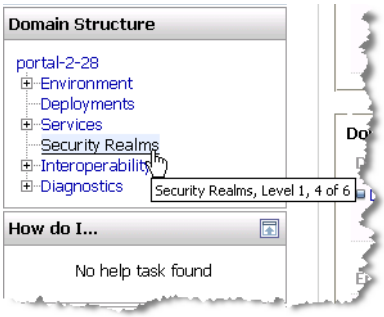
   `http://`*`servername`*`:`*`portnumber`*`/console`

   where *`servername`* is your server's IP name, and *`portnumber`* is the server's port. For example:

   ```
   http://localhost:7001/console
   ```

2. Click the **Security Realms** link in the Domain Structure window, as shown in Figure 15–31.

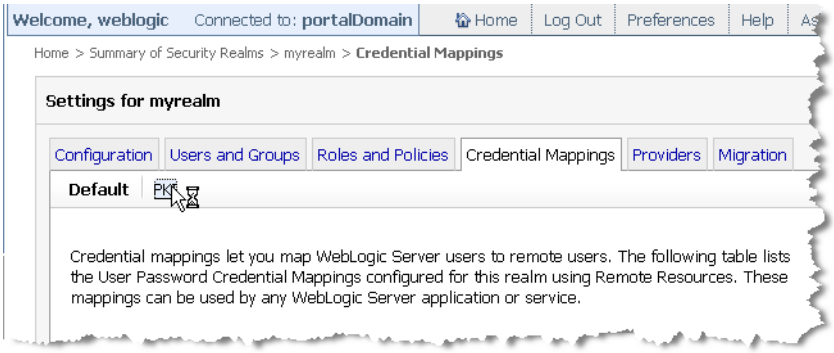
Figure 15–31 Selecting Security Realms



3. Select **myrealm** (or the name of the security realm you are using) and then select the **Credential Mappings** tab.
4. In the Credential Mappings tab, select **PKI**, as shown in Figure 15–32.

> **Tip:** PKI, or public key infrastructure, allows the exchange of data through the use of a public and a private cryptographic key pair that is obtained and shared through a trusted authority. For more information, see "Configure Credential Mapping Providers" in *Oracle Fusion Middleware Oracle WebLogic Server Administration Console Online Help*.

Figure 15–32 Select PKI



5. In the PKI Credential Mappings table, click **New** to create a new credential.
6. In the Create New Security Credential dialog, click **Next** without entering any remote resource attribute information.

> **Tip:** By leaving the remote resource attributes blank, the credential will be accepted by all producers. If you want to specify a producer, enter the appropriate information in this dialog.

7. In the Create a New Security Credential Map Entry dialog, enter the following in the Local User field:

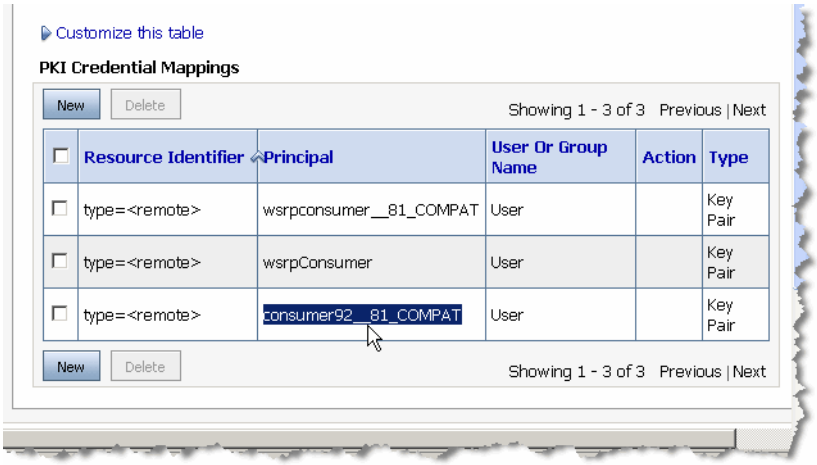
   *consumerName*__81_COMPAT

   where *consumerName* is the consumer name you entered previously in the `wsrp-consumer-security-config.xml` file. (Note that the name is followed by a double underscore.)

   For this example, the correct value for Local User is: `consumer9x__81_COMPAT`.

8. Select the **User** radio button.
9. In the Keystore Alias field, enter the alias you used for the key that you generated previously. In this example, the alias is `consumer92key`.
10. In the Password field, enter the key password you used when you generated the key. In this example, the password is `consumer92pass`.
11. Click Finish. Figure 15–33 shows the new principal name added to the PKI Credential Mappings: `consumer92__81_COMPAT`.

Figure 15–33 List of PKI Credential Mappings



12. Export the key from the consumer's keystore. Use the `keytool` utility to export the key that you created previously. You will use this key in the next set of steps to configure the WebLogic Portal 8.1x producer. For example:

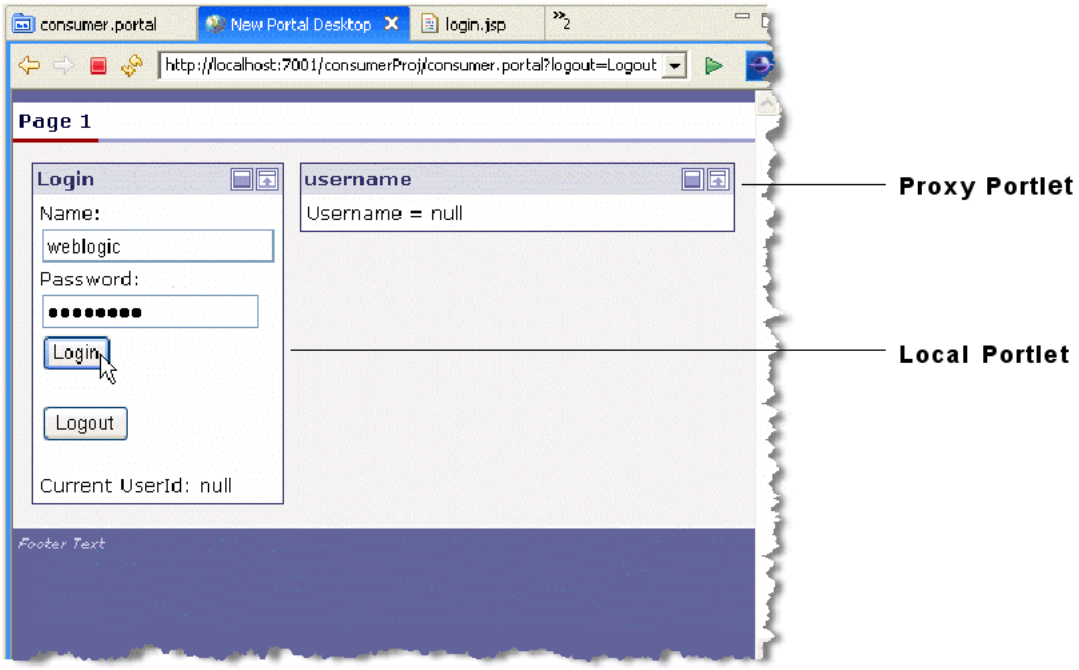
```
keytool -export -alias consumer92key -keystore
wsrpKeystore.jks -storepass password -keypass consumer92pass
-file consumer92.der
```

> **Note:** WebLogic Portal is configured with a default identity keystore (`wsrpKeystore.jks`). This default keystore configuration is appropriate for testing and development purposes. However, it should not be used in a production environment. For more information, see the WebLogic Server security documentation.

Checkpoint: In the previous steps, you associated this consumer, `consumer92`, to a key to sign the SAML assertion. If you now try to log in to the remote portlet, the previously seen error does not appear. This means that the consumer is now properly associated with a key. However, now after logging in, the user name is `null`, as shown in Figure 15–34. This is because this consumer is not yet known to the producer. The next set of steps demonstrate how to configure the WebLogic Portal 8.1x producer to accept the WebLogic Portal 9.2 or later version consumer's key.

Tip: It is interesting to note an important difference between the behavior of a WebLogic Portal 9.2 or later version producer and a WebLogic Portal 8.1x producer. If a WebLogic Portal 9.2 or later version producer cannot verify what the consumer is sending, you will receive an error. If a WebLogic Portal 8.1x producer cannot verify what the consumer is sending, the producer ignores this condition and continues with an anonymous user. In addition, if an 8.1x consumer sends an unverifiable message to a 9.2 or later version producer, the producer likewise ignores the condition and continues with an anonymous user.

Figure 15–34 User Name is Null



15.2.1.4 Configure the WebLogic Portal 8.1x Producer

This section explains how to configure the WebLogic Portal 8.1x producer. To do this, you import the key into the producer's keystore.

15.2.1.4.1 Import the Certificate To import the certificate, follow this procedure.

1. Copy the previously exported certificate to the system on which the producer application is deployed, using whichever method is appropriate for your system, such as FTP or SMB. You can put this file anywhere on the destination machine.
2. In a command window, CD to the root directory of the producer's domain. For example:

   ```
   <MW_HOME>/weblogic81/user_projects/domains/portalDomain
   ```

3. Import the key using the keytool utility. For example:

   ```
   keytool -import -keystore wsrpKeystore.jks -file
   c:\consumer92.der -storepass password -alias consumer9x
   -keypass consumer92pass
   ```

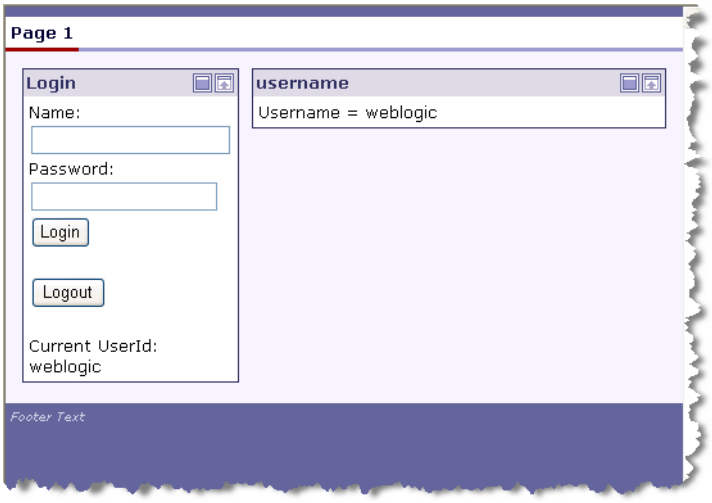
Note: WebLogic Portal is configured with a default identity keystore (`wsrpKeystore.jks`). This default keystore configuration is appropriate for testing and development purposes. However, it should not be used in a production environment. For more information, see the WebLogic Server security documentation.

Note: The alias argument must match the consumer name you used when you created the key on the consumer. In this example, that name is consumer9x.

4. Restart the server in which the producer application is deployed.

15.2.1.4.2 Test the Configuration After the producer server is restarted, you can once again test the remote portlet in the consumer application. When you log into the portal, you will see that the remote portlet now recognizes the user as logged in, as shown in Figure 15–35.

Figure 15–35 Successful Configuration



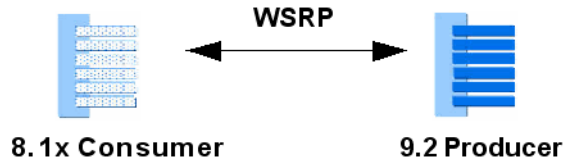
15.2.1.5 Summary

The preceding example demonstrated how to configure SAML security between a WebLogic Portal 9.2 or later version consumer and a WebLogic Portal 8.1x producer. In the next example, you will see the reverse: configuring SAML security between a WebLogic Portal 8.1x consumer and a WebLogic Portal 9.2 or later version producer.

15.2.2 SAML Security Between 8.1x Consumers and 9.2 or Later Version Producers

This section explains how to achieve security compatibility between a WebLogic Portal 8.1x consumer and an 9.2 or later version producer, as summarized in Figure 15–36.

Figure 15–36 Compatibility Use Case



The basic steps include:

- Section 15.2.2.1, "Configure the 8.1x Consumer"
- Section 15.2.2.2, "Configure the 9.2 or Later Version Producer"

15.2.2.1 Configure the 8.1x Consumer

This section explains how to configure the 8.1x consumer. The basic steps include generating a key and configuring the WSRP Consumer Security Service in the WebLogic Administration Portal.

15.2.2.1.1 Generate a Key This section explains how to use the keytool utility to generate a key on the consumer. Keytool, a Java utility distributed by Sun Microsystems, manages private keys and certificates. For detailed information on keytool, refer to the Sun Microsystems web site.

> **Note:** WebLogic Portal is configured with a default identity keystore (`wsrpKeystore.jks`). This default keystore configuration is appropriate for testing and development purposes. However, it should not be used in a production environment. For more information, see the WebLogic Server security documentation.

1. If you have not already done this, generate a key. To do this, CD to the root directory of the WebLogic Portal 8.1x consumer application's domain and use the keytool utility to generate the key. For example:

   ```
   <MW_HOME>/weblogic81/user_projects/domains/portal
   ```

   ```
   keytool -genkey -keystore wsrpKeystore.jks -alias
   consumer8xkey -storepass password -keypass consumer8xpass
   ```

   The options used in the example keytool command include the following:

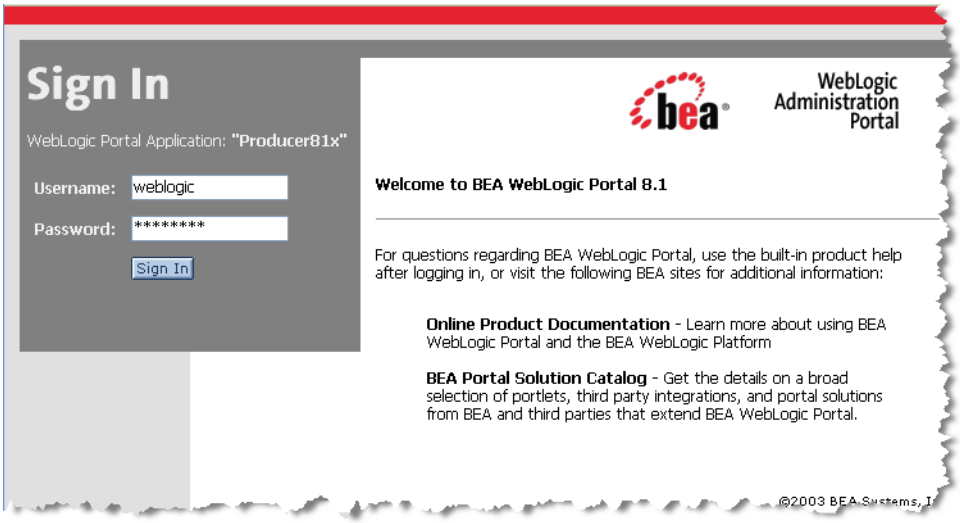
Command Parameter	Description
keytool	A Java tool used to generate a key.
-genkey	Instructs the keytool to generate a key.
-keystore	Specifies the name of the keystore. A keystore stores keys and certificates. The default keystore, `wsrpKeystore.jks`, is implemented as a file that protects private keys with a password.
-alias	Specifies a name for the generated key.
-storepass	Specifies the password for the keystore.
-keypass	Specifies the password to be used with the new key.

2. Log in to the version 8.1x WebLogic Administration Portal on the consumer application's server. To start the Administration Portal from Oracle Enterprise Pack for Eclipse, select **Portal** > **Portal Administration**. Or, enter the following URL in a browser:

   `http://localhost:7001/`*`applicationName`*`/login.jsp`

   where *`applicationName`* is the name of the WebLogic Portal consumer application.

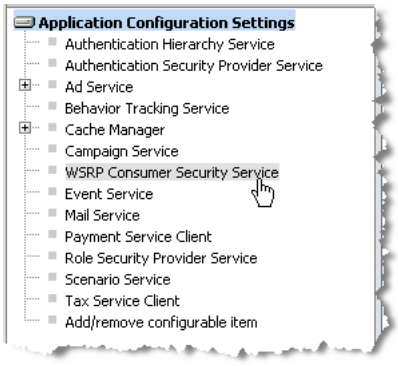
Figure 15–37 WebLogic Administration Portal Sign In Page



3. In the Administration Portal, select **Service Administration**.

4. In the Application Configuration Settings tree, select **WSRP Consumer Security Service**, as shown in Figure 15–38.

Figure 15–38 Selecting WSRP Consumer Security Service



5. In the Configuration Settings dialog, enter a name for the consumer, the Certificate Alias that you used when you generated the consumer key, and the Certificate Private Key Password that you used when you generated the key, as shown in Figure 15–39 and click **Update**.

Figure 15–39 Entering Security Service Parameters

6. Export the key using the keytool utility. To do this, CD to the consumer's domain root directory, and enter the appropriate keytool command. For example:

```
keytool -export -alias consumer8xkey -keystore
wsrpKeystore.jks -file consumer81.der
```

15.2.2.2 Configure the 9.2 or Later Version Producer

This section explains how to configure the producer. To do this, you must configure the producer's PKI credential mappings to include the consumer's certificate.

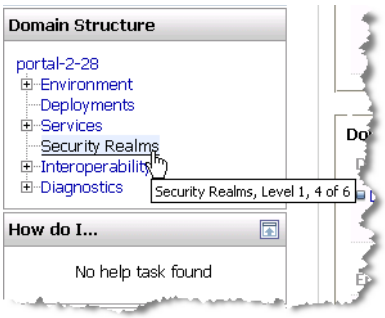
> **Note:** WebLogic Portal is configured with a default identity keystore (`wsrpKeystore.jks`). This default keystore configuration is appropriate for testing and development purposes. However, it should not be used in a production environment. For more information, see the WebLogic Server security documentation.

1. Copy the exported key to the WebLogic Portal 9.2 or later version producer's root domain directory using an appropriate method, such as FTP or SMB. You can put this file anywhere on the destination machine.
2. Use the keytool utility to import the key into the 9.2 or later version producer's keystore. For example:

```
keytool -import -keystore wsrpKeystore.jks -file
consumer81.der -alias consumer8xkey -keypass consumer8xpass
```

3. Log in to the WebLogic Server Administration Console on the producer server.
4. Click the **Security Realms** link in the Domain Structure window, as shown in Figure 15–31.

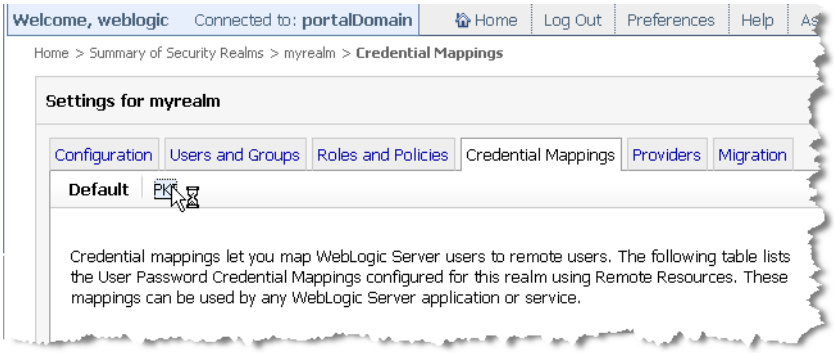
Figure 15–40 Selecting Security Realms



5. Select **myrealm** (or the name of the security realm you are using) and then select the **Credential Mappings** tab.
6. In the Credential Mappings tab, select **PKI**, as shown in Figure 15–32.

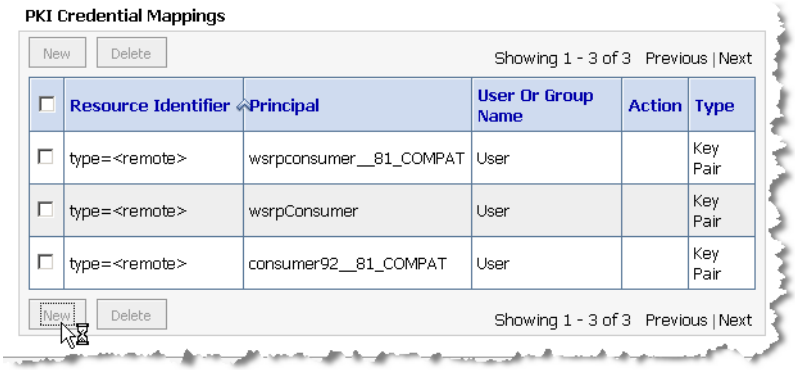
> **Tip:** PKI, or public key infrastructure, allows the exchange of data through the use of a public and a private cryptographic key pair that is obtained and shared through a trusted authority. For more information, see "Configure Credential Mapping Providers" in *Oracle Fusion Middleware Oracle WebLogic Server Administration Console Online Help*.

Figure 15–41 Select PKI



7. In the PKI Credential Mappings dialog, click **New**, as shown in Figure 15–42.

Figure 15–42 Creating a New PKI Credential Mapping



8. In the Creating the Remote Resource for the Security Credential Mapping dialog, leave all fields blank and click **Next**.

   Tip: By leaving the fields blank, this indicates that the credential is recognized for all consumers. If you want to restrict the credential to a specific consumer, you can fill in the required information.

9. In the Create a New Security Credential Map Entry dialog, enter the following information:

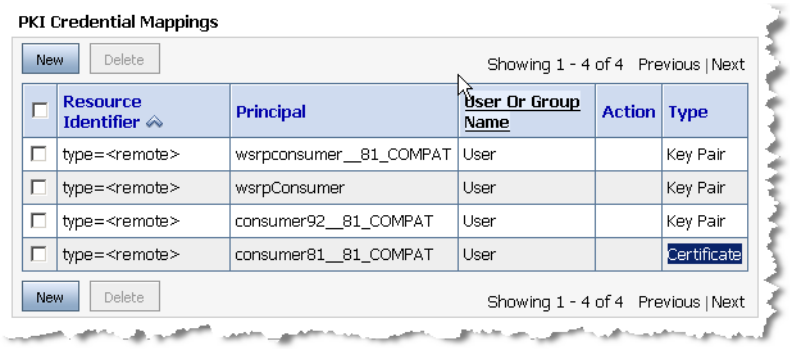
   - Select the **Certificate** radio button (true).
   - In the Principal Name field, enter *consumerName*`__81_COMPAT`, where *consumerName* is the name of the consumer. In this example, the name is `consumer8x`.
   - Select the **User** radio button.
   - In the Keystore Alias field, enter the alias you used when you imported the keystore. In this example, it is `consumer8xkey`.
   - In the Password field, enter the key password you used when you imported the keystore. In this example, it is `consumer81pass`.

   Figure 15–43 shows the completed dialog.

Figure 15–43 Entering PKI Credential Mappings Parameters

10. Click **Finish**. The PKI Credential Mappings table reappears and shows that the new certificate has been added, as shown in Figure 15–44.

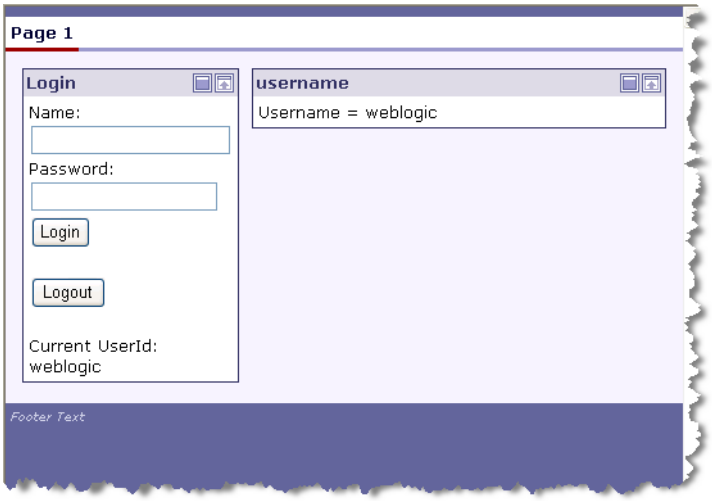
Figure 15–44 New Certificate Added to the Producer



15.2.2.3 Testing the Configuration

To test the configuration, log in to the consumer portal. As shown in Figure 15–45, the user name `weblogic` appears in the proxy portlet. This indicates success: the user was logged in successfully on the producer.

Figure 15–45 Successful Test



15.3 Using SAML Security with a Name Mapper

A name mapper is a class that maps one user name to another. Use a name mapper when the producer and consumer have different names for the same user. This section explains how to write and configure a name mapper class on both the consumer and the producer.

If you want to use a name mapping class on the producer or the consumer, the basic steps include:

- Section 15.3.1, "Writing a Name Mapper Class"
- Section 15.3.2, "Deploying the Mapper Classes"
- Section 15.3.3, "Configuring the Mapper Classes"

15.3.1 Writing a Name Mapper Class

WebLogic Portal provides two user name mapping interfaces:

- weblogic.security.providers.saml.SAMLCredentialNameMapper

  Implement this interface on the consumer to map a user name on the consumer to a new name. See Section 15.3.1.1, "Implementing SAMLCredentialNameMapper on the Consumer" for an example.

- weblogic.security.providers.saml.SAMLIdentityAssertionNameMapper

  Implement this interface on the producer to map a user name sent from the consumer to a user name on the producer. See Section 15.3.1.2, "Implementing SAMLIdentityAssertionNameMapper on the Producer" for an example.

15.3.1.1 Implementing SAMLCredentialNameMapper on the Consumer

Implement SAMLCredentialNameMapper on the consumer to provide name mapping on the consumer. Example 15–1 shows an example implementation of SAMLCredentialNameMapper.

The mapSubject() method gets a Subject (user) and returns a SAMLNameMapperInfo object. The method provides logic to test the user name and replace it with a new user name. This new user name is then returned in a SAMLNameMapperInfo object, which is then passed to the producer.

For detailed information on this interface, see the *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*.

Example 15–1 Example SAMLCredentialNameMapper Implementation

```
package com.bea.wsrp.qa.security;

import java.util.Collection;
import java.util.Set;
import javax.security.auth.Subject;

import weblogic.security.SubjectUtils;
import weblogic.security.providers.saml.SAMLCredentialNameMapper;
import weblogic.security.providers.saml.SAMLNameMapperInfo;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.WLSGroup;

public class CustomSAMLNameMapperImpl implements SAMLCredentialNameMapper {

private String nameQualifier = null;

public CustomSAMLNameMapperImpl (){ }

/************ SAMLCredentialNameMapper implementation**************/

public synchronized void setNameQualifier(String nameQualifier)
{
this.nameQualifier = nameQualifier;
}

public SAMLNameMapperInfo mapName (String name, ContextHandler handler)
{
return new SAMLNameMapperInfo(nameQualifier, name, null);
}

public SAMLNameMapperInfo mapSubject (Subject subject, ContextHandler handler)
{
// Provider checks for null Subject...
```

```
Set groups = subject.getPrincipals(WLSGroup.class);
String userName = null;

userName = SubjectUtils.getUsername(subject);
if (userName == null || userName.equals("")) {
System.out.println("mapSubject: Username string is null or
                      empty, returning null");
return null;
}

if (userName.equals("testUser"))
{
userName = "testUser_Mapped";
}

// Return mapping information...
return new SAMLNameMapperInfo(nameQualifier, userName, groups);
}
}
```

15.3.1.2 Implementing SAMLIdentityAssertionNameMapper on the Producer

Implement SAMLIdentityAssertionNameMapper on the producer to provide name mapping. Example 15–2 shows an example implementation of SAMLIdentityAssertionNameMapper. In this example, if you log in on the consumer as `testUser_Mapped`, the name mapper class retrieves that user name on the producer and logs you in as `testUser_Producer`.

The mapNameInfo() method gets a SAMLNameMapperInfo object from the consumer. This object contains the name with which the user logged in on the consumer. The method provides logic to test the user name from the consumer and replace it with a user name on the producer.

For detailed information on this interface, see the *Oracle Fusion Middleware Java API Reference for Oracle WebLogic Portal*.

Example 15–2 Example SAMLIdentityAssertionNameMapper Implementation

```
package com.bea.wsrp.qa.security;

import java.util.Collection;
import java.util.Set;
import javax.security.auth.Subject;

import weblogic.security.SubjectUtils;
import weblogic.security.providers.saml.SAMLIdentityAssertionNameMapper;
import weblogic.security.providers.saml.SAMLNameMapperInfo;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.WLSGroup;

public class CustomSAMLNameMapperImpl implements SAMLIdentityAssertionNameMapper
{
private String nameQualifier = null;

public CustomSAMLNameMapperImpl (){ }


/************ SAMLIdentityAssertionNameMapper implementation**************/

public String getGroupAttrName ()
```

```
{
return SAMLNameMapperInfo.BEA_GROUP_ATTR_NAME;
}

public String getGroupAttrNamespace ()
{
return SAMLNameMapperInfo.BEA_GROUP_ATTR_NAMESPACE;
}

public Collection mapGroupInfo(SAMLNameMapperInfo info, ContextHandler handlr)
{
return info.getGroups();
}

public String mapNameInfo(SAMLNameMapperInfo info, ContextHandler handler)
{
String userName = info.getName();

if (userName == null || userName.equals("")) {
System.out.println("mapNameInfo: Username string is null or
                      empty, returning null");
return null;
}

if (userName.equals("testUser_Mapped"))
{
userName = "testUser_Producer";
}

return userName;
}
}
```

15.3.2 Deploying the Mapper Classes

Whether you are implementing a mapper class on the producer or the consumer, the class must be in the server's class path. For information on adding classes to the server class path, refer to the WebLogic Server topic "Adding Startup and Shutdown Classes to the Classpath" at in the Oracle WebLogic Server Administration Console Help.

15.3.3 Configuring the Mapper Classes

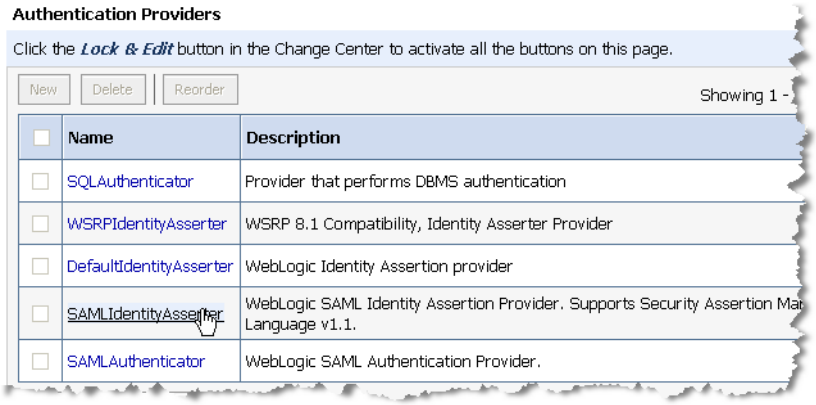
You need to use the WebLogic Server Administration Console add the mapper classes to the security realm of the producer and/or consumer.

15.3.3.1 Adding a Mapper Class to the Producer

To add a mapper class to the producer:

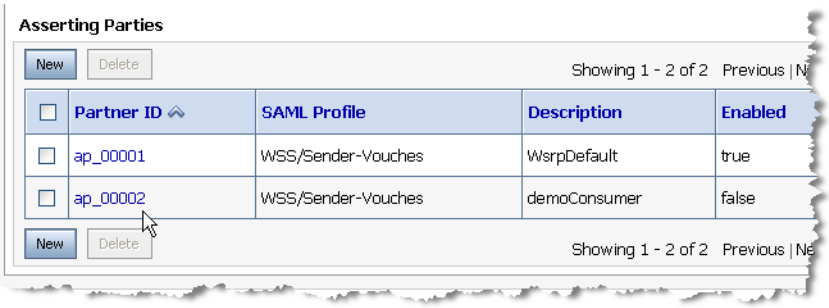
1. Open the WebLogic Server Administration Console on the producer machine and log in.
2. Select **Security Realms** from the Domain Structure tree.
3. Select a security realm, such as **myrealm**.
4. Select **Providers**.
5. Select **SAMLIdentityAsserter**, as shown in Figure 15–46. An identity asserter allows WebLogic Server to establish trust by validating a user.

Figure 15–46 Selecting the Identity Asserter



6. Click the **Management** tab.
7. In the Asserting Parties table, click the **Partner ID** link for the asserting party you want to use. In this example, the link is ap_0002 for the asserting party called demoConsumer, as shown in Figure 15–47.

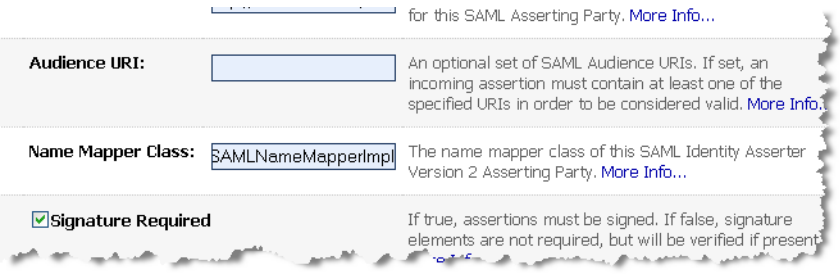
Figure 15–47 Selecting the New Asserting Party



8. In the Configuration tab, enter the full class name of the mapper class in the Name Mapper Class field, as shown in Figure 15–48. For example:

   ```
   com.bea.wsrp.qa.security.CustomSAMLNameMapperImpl
   ```

Figure 15–48 Entering the Name Mapper Class



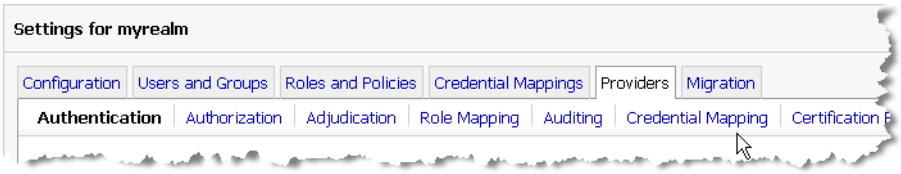
9. Click **Save**.

15.3.3.2 Adding a Mapper Class to the Consumer

To add a mapper class to the producer:

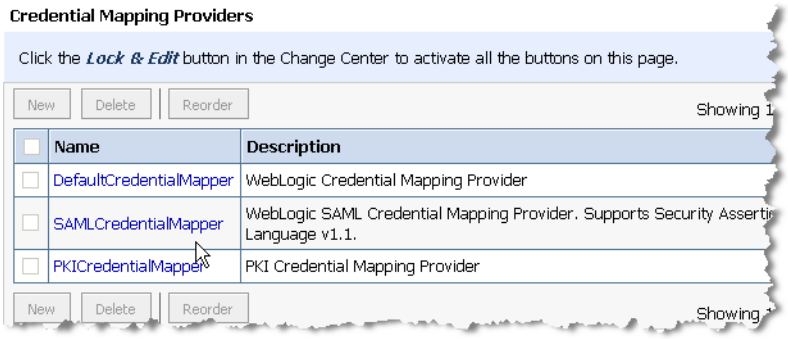
1. Open the WebLogic Server Administration Console on the consumer machine and log in.
2. Select **Security Realms** from the Domain Structure tree.
3. Select a security realm, such as **myrealm**.
4. Select the **Providers** and then the **Credential Mapping** tab.

Figure 15–49 Selecting the Credential Mapping Tab



5. Select **SAMLCredentialMapper**, as shown in Figure 15–50.

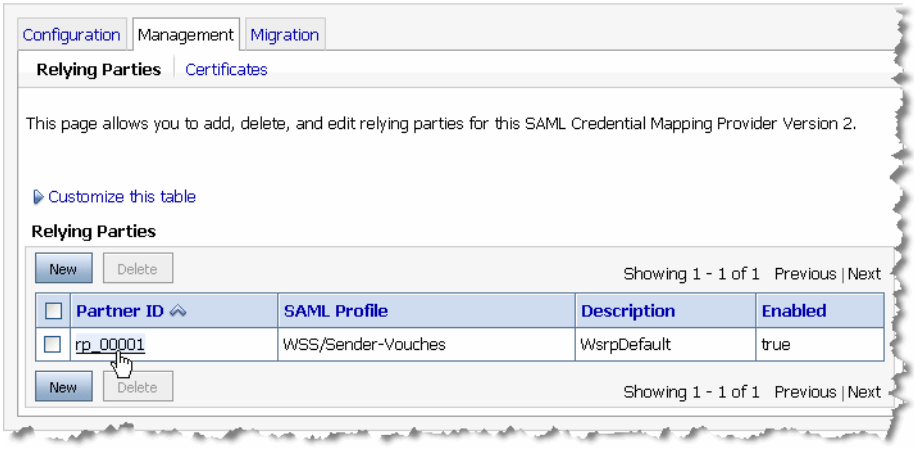
Figure 15–50 Selecting the SAMLCredentialMapper



6. Select **Management** tab.
7. Select the **Relying Parties** link for the relying party you want to use. For example, the relying party shown in Figure 15–51 is rp_00001.

> **Tip:** For more information on relying party configuration, see the WebLogic Server topic, "Configuring a Relying Party" in the Oracle WebLogic Server Administration Console Online Help.

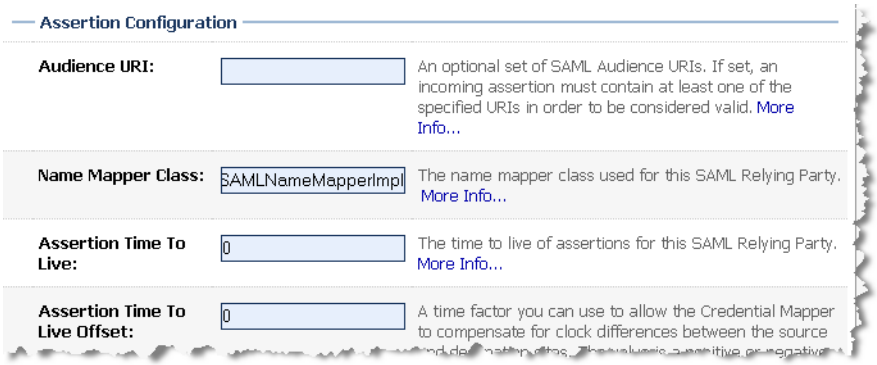
Figure 15–51 Select the Relying Party



8. In the Name Mapper Class field, enter the full class name of the mapper class, as shown in Figure 15–52. For example:

   com.bea.wsrp.qa.security.CustomSAMLNameMapperImpl

Figure 15–52 Entering the Name Mapper Class



9. Click **Save**.

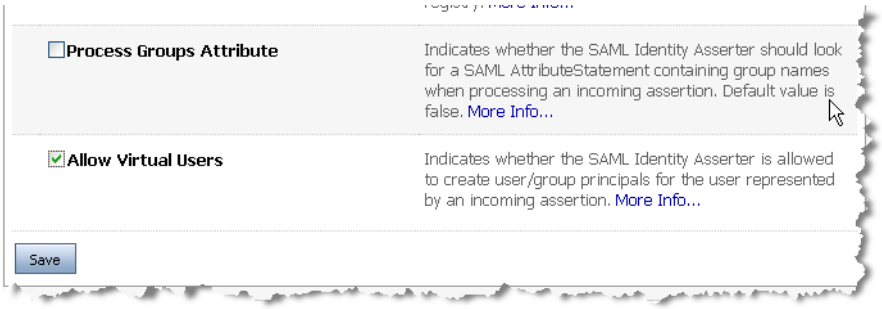
15.4 Allowing Virtual Users

You can configure the producer to automatically create a new user if it does not recognize the user name sent from the consumer. This feature is useful if you do not want to manually create a unique user name on the producer for every user who might log in from a consumer application. You can use this feature as long as the producer is configured to recognize the consumer's SAML token, as explained previously in this chapter.

To configure the producer to allow virtual users:

1. Log in to the WebLogic Server Administration Console.
2. Navigate to the SAMLIdentityAsserter **Configuration** tab. For instructions on navigating to this tab, see Section 15.3.3.1, "Adding a Mapper Class to the Producer".
3. Check the **Allow Virtual Users** checkbox, as shown in Figure 15–53.
4. Click **Save**.

Figure 15–53 All Virtual Users



16

Configuring User Name Token Security

User Name Token, or UNT, is an alternative to SAML and provides the same basic single sign-on capability as SAML provides. User Name Token lets you map the local user on the consumer to a user on the producer. This chapter explains how to configure User Name Token security for a federated portal.

This chapter includes the following sections:

16.1 Configuring the Consumer

On the consumer, you need to set up credential mappings. Credential mapping is the process whereby a legacy system's database is used to obtain an appropriate set of credentials to authenticate users to a target resource. In WebLogic Server, a Credential Mapping provider is used to provide credential mapping services and bring new types of credentials into the WebLogic Server environment. For more information on credential mapping, see the WebLogic Server topic, "Credential Mapping Providers" in *Oracle Fusion Middleware Developing Security Providers for Oracle WebLogic Server*.

1. Log in to the WebLogic Server Administration Console on the consumer. The URL for the console is:

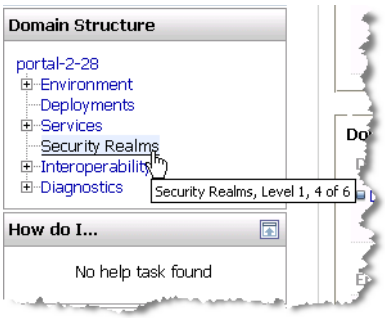
   `http://`*`servername`*`:`*`portnumber`*`/console`

   where *`servername`* is your server's IP name, and *`portnumber`* is the server's port. For example:

   `http://localhost:7001/console`

2. Click the **Security Realms** link in the Domain Structure window, as shown in Figure 16–1.

Figure 16–1 Selecting Security Realms



1. Select **myrealm** (or the name of the security realm you are using).
2. Select the **Credential Mappings** tab.
3. Select the **Default** link to open the Default Credential Mappings dialog, as shown in Figure 16–2.

Figure 16–2 Default Credential Mappings Dialog

4. Click **New**.
5. In the Create a New Security Credential Mapping dialog, shown in Figure 16–6, complete the fields listed below.
   - **Protocol** – The protocol for the remote resource, such as HTTP or HTTPS.
   - **Remote Host** – The name of the remote resource. For example: `myproducer`
   - **Remote Port** – The port number of the remote resource. For example: `7001`
   - **Remote Path** – The path of the remote resource. You need to enter the markup path for the producer. Be sure to begin the path with a "/". For example:

     ```
     /myProducerWebProject/producer/wsrp-1.0/markup
     /myProducerWebProject/producer/wsrp-1.0/portletManagement
     /myProducerWebProject/producer/wsrp-1.0/registration
     /myProducerWebProject/producer/wsrp-wlp-ext-1.0/markup
     /myProducerWebProject/producer/wsrp-1.0/serviceDescription
     ```

To obtain this path, you can enter the WSDL address of the producer in a browser. For example, if the producer web application is called myProducerWebApp, the WSDL URL is:

```
http://producerHost:producerPort/myProducerWebApp/producer?wsdl
```

where *producerHost* is the host name of the producer server and *producerPort* is the port number of the producer server.

The producer's WSDL definition appears in the browser. Locate the service description, and copy the markup path, as shown in Figure 16–3.

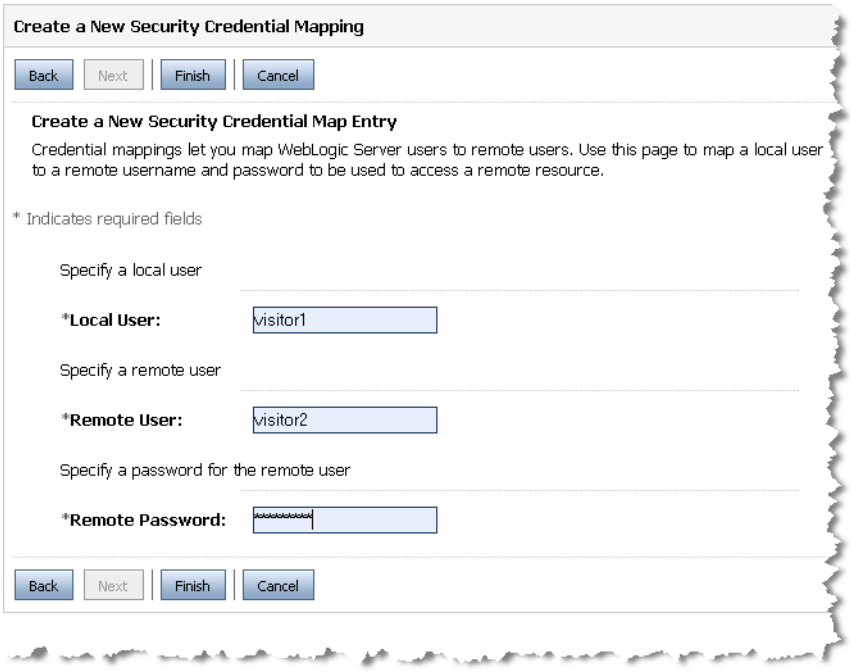
Figure 16–3 Finding the Markup Port

6. Click **Next**.
7. In the Create a New Security Credential Map Entry dialog, enter the local (consumer) user name and the user name on the producer to which you want to map that local name. Also, enter the password for the user name on the producer, as shown in Figure 16–4.

Note: The local user you enter must exist on the consumer. If the user does not exist, you need to create it using the User Management feature of the WebLogic Portal Administration Console.

Tip: The local user name and the user name on the producer can be the same name or different names.

Figure 16–4 Specify User Mapping



8. Click **Finish**. The new mapping appears in the Default Credential Mappings table, as shown in Figure 16–5.

Figure 16–5 Default Credential Mappings

Figure 16–6 Completed Dialog

Checkpoint: You have configured a credential mapping on the consumer. The next step is to configure the producer to recognize that mapping.

16.2 Configuring the Producer

This section explains how to configure the producer. On the producer, you need to set up the authentication and change the WSDL templates to include the UNT policy.

16.2.1 Set Up Authentication

You set up authentication using WebLogic Server Administration Console.

> **Tip:** The WebLogic Authentication provider allows you to manage users and groups in one place, the embedded LDAP server. Note that the Administration Console refers to the WebLogic Authentication provider as the Default Authenticator. For more information on authentication, see the WebLogic Server topic, "Configure Authentication and Identity Assertion Providers" in the Oracle WebLogic Server Administration Console Online Help.

To set up authentication:

1. Log in to the WebLogic Server Administration Console on the consumer. The URL for the console is:

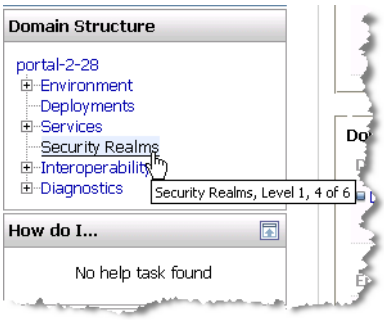
   `http://servername:portnumber/console`

where *servername* is your server's IP name, and *portnumber* is the server's port. For example:

```
http://localhost:7001/console
```

2. Click the **Security Realms** link in the Domain Structure tree, as shown in Figure 16–1.

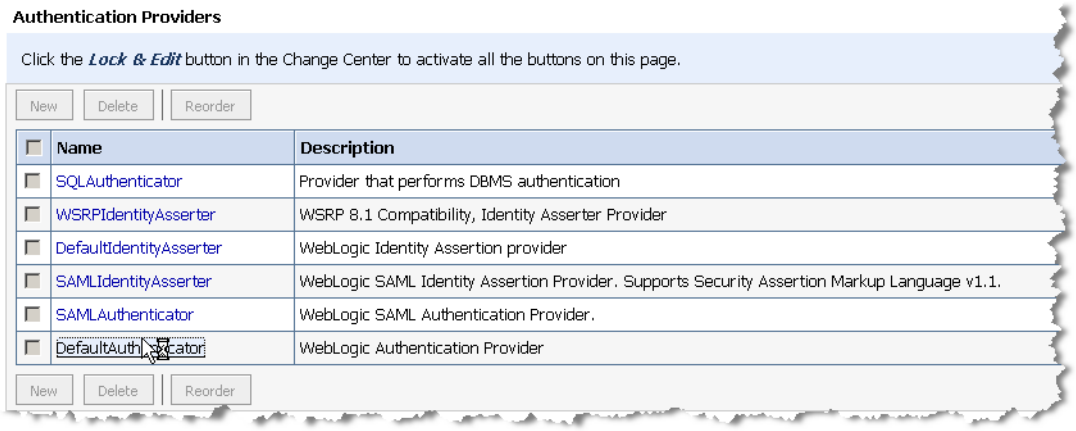
Figure 16–7 Selecting Security Realms



3. Select **myrealm** (or the name of the security realm you are using).
4. Select the **Providers** tab.
5. Select the **Authentication** tab.
6. Select **DefaultAuthenticator**, as shown in Figure 16–8.

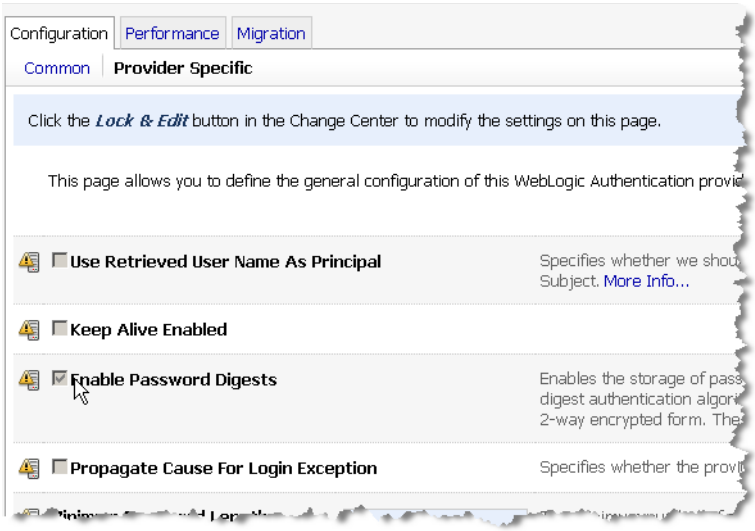
> **Tip:** If the DefaultAuthenticator selection is not present, you need to add it and restart the server.

Figure 16–8 Select the DefaultAuthenticator



7. In the Configuration tab, select **Provider Specific**.
8. Select the **Enable Password Digest** checkbox, as shown in Figure 16–9. You must select this checkbox to enable the WebLogic Authentication Provider to store the password in a two-way encrypted (reversible) form.

Figure 16–9 Enable Password Digests

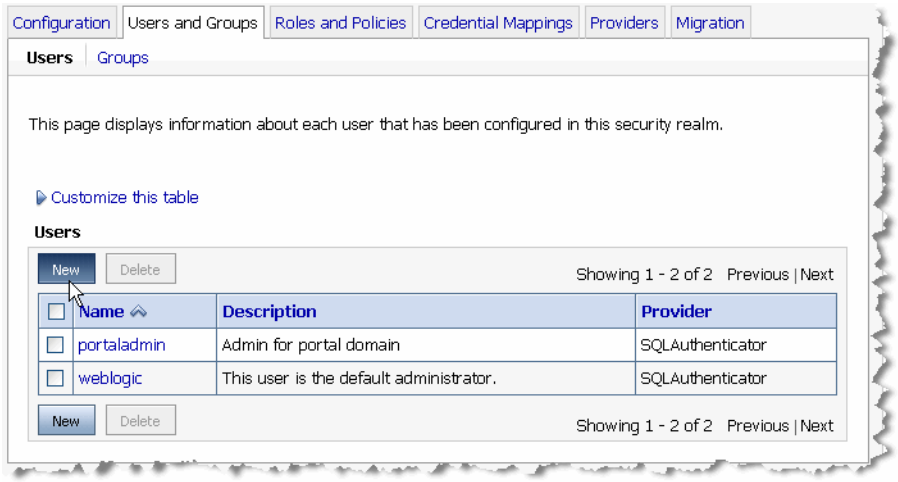


9. Select the **Users and Groups** tab.
10. Select **Users**.

> **Note:** The existing user name and password will not work.

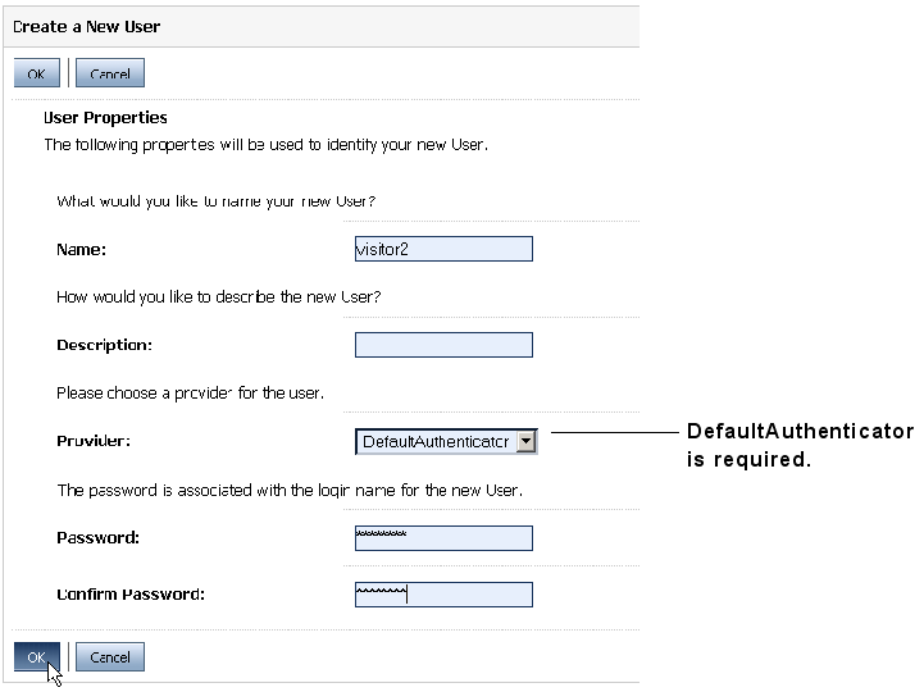
11. Click **New**, as shown in Figure 16–10. The Create a New User dialog appears.

Figure 16–10 Create a New User



12. In the Create a New User dialog, complete the Name and Password fields.
13. Select **DefaultAuthenticator** from the dropdown menu, as shown in Figure 16–11, and click **OK**. Note that you must use the DefaultAuthenticator for users on the producer. The user you create must match the user you mapped to when you configured the consumer (as explained previously).

Figure 16–11 Create a New User Dialog



16.2.2 Modify the WSDL Templates in the Producer Web-App

You must update WSDL templates of your producer web application to include the UNT policy.

To update WSDL templates to include the UNT policy:

1. Open your producer web application in Oracle Enterprise Pack for Eclipse (OEPE).
2. Open the Merged Projects view for your web application.
3. Right-click `wsrp-wsdl-template.wsdl` and select **Copy to Workspace**.
4. In the **Copy to Workspace** dialog, click **OK**.
5. Right-click `wsrp-wsdl-template-v2.wsdl`, and select **Copy to Workspace.**
6. In the **Copy to Workspace** dialog, click **OK**.
7. In your workspace, open both files for editing.
8. In both files, replace the following entries:

```
<wsp:Policy wsu:Id="ProducerDefaultPolicy">
        <wsp:All>
            <wssp:Identity>
                <wssp:SupportedTokens>
                    <wssp:SecurityToken
TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-prof
ile-1.0#SAMLAssertionID">
                        <wssp:Claims>
<wssp:ConfirmationMethod>sender-vouches</wssp:ConfirmationMethod>
                        </wssp:Claims>
                    </wssp:SecurityToken>
                </wssp:SupportedTokens>
            </wssp:Identity>
        </wsp:All>
```

```
    </wsp:Policy>
```

With:

```
<wsp:Policy wsu:Id="ProducerDefaultPolicy">
        <wsp:All>
            <wssp:Identity>
              <wssp:SupportedTokens>
                <wssp:SecurityToken
TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-tok
en-profile-1.0#UsernameToken">
                     <wssp:UsePassword
Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-pr
ofile-1.0#PasswordDigest"/>
                </wssp:SecurityToken>
              </wssp:SupportedTokens>
            </wssp:Identity>
        </wsp:All>
    </wsp:Policy>
```

9. Save your changes to the two files.

16.2.3 Summary

The User Name Token security feature lets you set up single sign-on between consumers and producers. The User Name Token method is an alternative to SAML, which is the default security for WebLogic Portal consumers and producers.

17

Configuring WSRP Security Between WLP and a WebCenter Portal: Framework Application

This chapter describes one technique for establishing a secure communications channel for WSRP transactions between WebLogic Portal and a WebCenter Portal: Framework application. It includes the following sections:

17.1 Introduction

For web-based transactions to be secure, the following four components must be addressed:

- **Authentication** – Verification of the sender's identity.
- **Integrity** – Protection against unauthorized changes.
- **Message Freshness** – Protection against replay attacks in which a message is captured and resent.
- **Confidentiality** – Protection against unauthorized viewing of the message.

The following configuration steps will enable integrity, authentication, and message freshness constraints in WSRP transactions between Framework applications and WLP applications, as follows:

- *Authentication* is handled through the SAML 1.1 protocol, with the sender-vouches assertion. This means that the user will authenticate through some unspecified mechanism on the consumer, and the consumer will propagate and "vouch" for the user's identity to the producer.
- *Integrity* is handled by digitally signing the message's body, BST, and SAML assertion with the SHA1 algorithm. This signature asserts that these components of the message have not been modified in transit from the consumer to the producer.

- *Message freshness* is handled by adding a time constraint condition to the SAML assertion. This specifies that the SAML assertion is only valid for a limited window of time. When the SAML assertion is invalidated, the entire message will be rejected. This time window is configurable on the consumer.

Note: *Message confidentiality* is not addressed in these steps. If confidentiality is a concern for your WSRP environment, please consider enabling SSL between your producer and consumer.

These security settings are but one possible configuration of Web Service security for WSRP. Many other Web Service security configuration settings can be further adjusted in both the WebLogic Portal and Framework application environments, as long as the settings are enabled and recognized in both environments. For further detailed information, see *Oracle Fusion Middleware Securing WebLogic Web Services for Oracle WebLogic Server*.

17.2 SAML Security Between a WebCenter Portal: Framework Application Consumer and a WebLogic Portal Producer

This section explains how to configure SAML security for both a Framework application consumer and a WLP producer. The tasks described in this section are:

- Section 17.2.1, "Configuring the Consumer"
- Section 17.2.2, "Configuring the Producer"

17.2.1 Configuring the Consumer

This section discusses how to generate a key pair and export the public key certificate on the consumer.

17.2.1.1 Generate a Key Pair

This section explains how to generate a key on the consumer using the keytool utility, a Java utility distributed by Sun Microsystems that manages private keys and certificates. For detailed information on keytool, refer to the Sun Microsystems website.

1. On the Framework application consumer, open a command window and change directory to the `<WEBLOGIC_HOME>/wlserver_10.3/server/bin` directory.
2. Run the `setWLSEnv.cmd/.sh` command to set up the required environment variables.
3. Run the keytool command to generate a new key pair. For example, the following command generates a key pair, wraps the public key in a certificate, and stores the certificate and the private key in a keystore named `mykeystore.jks`, identified by the alias `wckey`:

```
keytool -genkeypair -alias wckey -keypass wckeypass -keyalg rsa -keysize 1024
-keystore mykeystore.jks -storepass mykeystorepass -dname "CN=Oracle Corp,
OU=WLP, O=Oracle, L=Boulder, ST=CO, C=US"
```

4. Make a note of your new keystore's passphrase, the key pair's alias, and the key pair's passphrase. This data, as well as the keystore file itself (`mykeystore.jks`), will be used when configuring the Framework application consumer.

17.2.1.2 Export the Public Key Certificate

The producer needs the public key certificate (the public half of the "key pair" generated in the previous step) installed in its trust key store. Follow these steps to export the public key certificate to a file, which will then be imported into a trusted key store on the producer.

1. On the consumer, open a command window and change directory to the `<WEBLOGIC_HOME>/wlserver_10.3/server/bin` directory.
2. Run the `setWLSEnv.cmd/.sh` command to set up the required environment variables.
3. Run the keytool command to export the previously-created certificate to a file. For example, the following command creates a certificate file named `wckey.der` from the key pair identified by alias `wckey`:

```
keytool -exportcert -alias wckey -keypass wckeypass -keystore mykeystore.jks
-storepass mykeystorepass -file wckey.der
```

17.2.2 Configuring the Producer

This section explains how to configure the producer. To do this, you import the public key certificate into the SAML asserter, and configure the asserting party properties.

17.2.2.1 Import the Public Key Certificate Into The Producer Domain's Trust Key Store

1. Copy the certificate file created in the previous step to the WebLogic Portal producer's domain directory (for example, `<MW_HOME>/user_projects/domains/base_domain`).
2. On the producer, open a command window and change directory to the `<WEBLOGIC_HOME>/server/bin` directory.
3. Run the `setWLSEnv.cmd/.sh` command to set up the required environment variables.
4. Change directory to the root directory for your producer's domain (for example, `<MW_HOME>/user_projects/domains/base_domain`)
5. Run the keytool command to import the previously-created certificate file to the domain's trust keystore. For example, the following command imports the certificate identified by alias wckey from the certificate file named wckey.der to the DemoTrust.jks keystore:

```
keytool -importcert -keystore DemoTrust.jks -storepass
DemoTrustKeyStorePassPhrase -file wckey.der -alias wckey -keypass wckeypass
```

6. If prompted to "Trust this certificate? [no]: ", type yes and press Enter to add the certificate to the keystore.
7. If your server is currently running, restart it.

Note: WebLogic Portal is configured with a default identity keystore (`DemoIdentity.jks`) and a default trust keystore (`DemoTrust.jks`). In addition, WebLogic Portal trusts the CA certificates in the JDK cacerts file. This default keystore configuration is appropriate for testing and development purposes. However, these keystores should not be used in a production environment. For more information, see the WebLogic Server security documentation.

17.2.2.2 Modify the WSDL Templates in the Producer Web-App

1. Copy the files `wsrp-wsdl-template.wsdl` and `wsrp-wsdl-template-v2.wsdl` to your workspace and open them for editing. The procedure for copying files to your workspace is described in "Copying J2EE Library Files Into a Project" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.
2. In both files, replace the existing `<wsp:Policy>` element with the following XML:

Example 17–1 Replacement wsp:Policy Element

```
<wsp:Policy wsu:Id="ProducerDefaultPolicy"/>
<wsp:Policy wsu:Id="WebCenterPolicy" xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
    <sp:AsymmetricBinding>
        <wsp:Policy>
            <sp:InitiatorToken>
                <wsp:Policy>
                    <sp:X509Token
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRec
ipient">
                        <wsp:Policy>
                            <sp:WssX509V3Token10/>
                        </wsp:Policy>
                    </sp:X509Token>
                </wsp:Policy>
            </sp:InitiatorToken>
            <sp:RecipientToken>
                <wsp:Policy>
                    <sp:X509Token
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never">
                        <wsp:Policy>
                            <sp:WssX509V3Token10/>
                        </wsp:Policy>
                    </sp:X509Token>
                </wsp:Policy>
            </sp:RecipientToken>
            <sp:AlgorithmSuite>
                <wsp:Policy>
                    <sp:Basic128/>
                </wsp:Policy>
            </sp:AlgorithmSuite>
            <sp:Layout>
                <wsp:Policy>
                    <sp:Lax/>
                </wsp:Policy>
            </sp:Layout>
            <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
    </sp:AsymmetricBinding>
    <sp:SignedSupportingTokens>
        <wsp:Policy>
            <sp:SamlToken
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRec
ipient">
                <wsp:Policy>
                    <sp:WssSamlV11Token10/>
                </wsp:Policy>
            </sp:SamlToken>
```

```
        </wsp:Policy>
    </sp:SignedSupportingTokens>
    <sp:Wss10>
        <wsp:Policy>
            <sp:MustSupportRefKeyIdentifier/>
            <sp:MustSupportRefIssuerSerial/>
        </wsp:Policy>
    </sp:Wss10>
</wsp:Policy>
```

3. Save your changes to these two files.

17.2.2.3 Modify the Web Services Policy Configuration in the Producer Web-App

1. Copy the file `WEB-INF/weblogic-webservices-policy.xml` to your workspace and open it for editing. The procedure for copying files to your workspace is described in "Copying J2EE Library Files Into a Project" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.
2. Replace the entire contents of the file with the following XML:

Example 17–2 Replacement weblogic-webservices-policy.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<webservice-policy-ref xmlns="http://www.bea.com/ns/weblogic/90"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <!-- Use WebLogic Server Admin Console to add new policies -->
    <ref-name>WebCenter Policies for the WSRP Producer</ref-name>

    <port-policy>
        <port-name>WSRP_v2_Markup_Service</port-name>
        <ws-policy>
            <uri>#WebCenterPolicy</uri>
            <direction>inbound</direction>
        </ws-policy>
    </port-policy>
    <port-policy>
        <port-name>WSRPBaseService</port-name>
        <ws-policy>
            <uri>#WebCenterPolicy</uri>
            <direction>inbound</direction>
        </ws-policy>
    </port-policy>
    <port-policy>
        <port-name>WLP_WSRP_Ext_Service</port-name>
        <ws-policy>
            <uri>#WebCenterPolicy</uri>
            <direction>inbound</direction>
        </ws-policy>
    </port-policy>
</webservice-policy-ref>
```

3. Save your changes, and republish your web project.

17.2.2.4 Add a New Asserting Party to the SAML Identity Asserter

This section describes the final step in the producer configuration.

> **Tip:** For more information on asserting party and other topics in this section, see "SAML Framework Concepts" in *Oracle Fusion Middleware Understanding Security for Oracle WebLogic Server*.

1. Open the WebLogic Server Administration Console on the producer server and log in.
2. Select **Security Realms**.
3. Select a security realm, such as **myrealm**.
4. Select the **Providers** tab.
5. Select the **Authentication** tab.
6. Select **SAMLIdentityAsserter**. An identity asserter allows WebLogic Server to establish trust by validating a user.
7. Select the **Management** tab.
8. Select the **Asserting Parties** tab
9. In the Asserting Parties table, click **New**.
10. In the Profile pulldown menu, select **WSS/Sender Vouches**.
11. In the Description field, enter a name to identify the asserting party, and select **OK**. For example: WebCenter SAML token.
12. Enable the new asserting party. To do this, click the **Partner ID** link for the new asserting party (for example, ap_0002).
13. Set the asserting party values as follows:

Parameter	Value
Enabled	true (Select the checkbox)
Target URL	default
Issuer URI	Set on the consumer (for example, `www.oracle.com`)

14. Click **Save**. If there were no problems, the message "Settings updated successfully" appears.
15. Perform the WSRP interoperability steps described in Section 13.1, "Consuming WLP Portlets in WebCenter Portal Applications and Oracle Portal Applications."

The WebLogic Portal producer is now configured for SAML interoperability with a basic Framework application SAML configuration. The next step is to associate the Framework application consumer with the key pair created earlier (see Section 17.2.1.1, "Generate a Key Pair").

> **Note:** For more detailed information on the following steps, see "Securing a WSRP Producer with WS-Security" in *Oracle Fusion Middleware Administrator's Guide for Oracle WebCenter*.

17.2.2.5 Register the WebLogic Portal Producer with the WebCenter Portal: Framework Application Consumer

1. Copy the keystore created earlier (see Section 17.2.1.1, "Generate a Key Pair") to your consumer server's filesystem, and note the path.

2. From Oracle JDeveloper, follow these standard steps for registering a producer using the Register WSRP Portlet Producer wizard, with the following exceptions:

   a. On the Configure Security Attributes page, set the following values:

Parameter	Value
Token Profile	WSS 1.0 SAML Token with Message Integrity
Configuration	Custom
Default User	A default username to send when unauthenticated (for example, fmwadmin)
Issuer Name	This needs to match the Issuer URI on the producer (for example, www.oracle.com). See Section 17.2.2.4, "Add a New Asserting Party to the SAML Identity Asserter."

   b. On the Specify Key Store page, set the following values

Parameter	Value
Store Path	Path on the consumer server to the JKS file. See Section 17.2.2.5, "Register the WebLogic Portal Producer with the WebCenter Portal: Framework Application Consumer."
Store Password	The keystore password. See Section 17.2.1.2, "Export the Public Key Certificate."
Store Type	JKS
Signature Key Alias	The key alias. See Section 17.2.1.2, "Export the Public Key Certificate."
Signature Key Password	The key passphrase. See Section 17.2.1.2, "Export the Public Key Certificate."
Encryption Key Alias	Leave the field blank.
Encryption Key Password	Leave the field blank.

17.2.2.6 Test the Configuration

The easiest way to test the configuration involves three steps:

1. Create a simple JSP portlet on the producer with the following content:

```
<%@ page language="java" contentType="text/html;charset=UTF-8" %>
<p>Principal: <%=request.getUserPrincipal() %></p>
<p>Remote User: <%=request.getRemoteUser() %></p>
```

   This will show the username sent by the consumer when rendered, if the SAML configuration is working properly.

2. Specify a default authenticated user when you establish your consumer's connection to the producer. (See Section 17.2.2.5, "Register the WebLogic Portal Producer with the WebCenter Portal: Framework Application Consumer.") By doing this, the Framework application consumer will automatically send that username to the WebLogic Portal producer, without requiring the creation of a login mechanism on the consumer-side.

3. Render the remote portlet on the consumer, and verify that the default username that was specified is rendered in the portlet's body.

17.3 SAML Security Between a WebLogic Portal Consumer and a WebCenter Portal: Framework Application Producer

This section discusses the producer-side and consumer-side configuration required to set up SAML security between a WLP consumer and a Framework application producer.

The configuration steps include:

- Section 17.3.1, "Register the SSL-Enabled Producer"
- Section 17.3.2, "Update Runtime Keystore"
- Section 17.3.3, "Register the WebCenter Portal: Framework Application Producer with the WebLogic Portal Consumer"
- Section 17.3.4, "Add an Authentication Mechanism To Your Portal"
- Section 17.3.5, "Testing the Configuration"

17.3.1 Register the SSL-Enabled Producer

To register the SSL-enabled producer:

1. Download the certificate of the HTTPS producer URL and save it in the `.PEM` format.

   Use Firefox 3.0 or later to download the certificate directly to `.PEM` format, or for other browsers use the WebLogic Server `der2pem` tool to convert to PEM format. Note that WebLogic does not recognize any other format other than `.PEM` format.

2. Import the certificate into the `cacerts` file in *`JDK_HOME`*`/jre/lib/security` using the `importcert` keytool command.

   Note that *`JDK_HOME`* is the path of Java used by Eclipse, and this should be the JDK used by the WebLogic Portal consumer.

   ```
   keytool -importcert -alias portlet_cert -file HOME/portlet_pem -keystore
   ./cacerts -storepass password
   ```

   Where:

   - *`portlet_cert`* is the portlet certificate alias
   - *`portlet_pem`* is the portlet certificate file (for example, `portlet_cert.pem`)
   - *`password`* is the keystore password

3. Restart Oracle Enterprise Pack for Eclipse.
4. Use Oracle Enterprise Pack for Eclipse to register the producer.

17.3.2 Update Runtime Keystore

The runtime SSL keystore used by the consumer server could be different than the one used by Oracle Enterprise Pack for Eclipse. For registration, the certificate (since it's a self-signed one) should be trusted by the consumer runtime keystore. This means that you will need to locate the keystore and then update the keystore with the SSL certificate of the producer.

1. Download the certificate of the HTTPS producer URL and save it in `.PEM` format.

2. Locate keystore by looking up the server startup logs or by looking up setDomainEnv.sh. To do this, start the server and search for "`-Djavax.net.ssl.trustStore`" in the logs.
3. Update the keystore with the SSL certificate of the producer using the following command:

```
-importcert -trustcacerts -alias <valid_alias_name> -file <SSL_Cert_Producer>
-keystore <SSL_Trust_Store_Consumer  -storepass <Pass_Phrase>
```

   For example:

```
<JDK6>/bin/keytool -importcert -trustcacerts -alias seshan_prod_vm4 -file
customidentity.crt -keystore <JDEV_MW_HOME>/wlserver_
10.3/server/lib/DemoTrust.jks -storepass DemoTrustKeyStorePassPhrase
```

4. Restart the consumer server.
5. Turn off the Host Name Verification on the consumer:
   a. Access the WebLogic Portal Administration Console.
   b. Navigate to **Servers**, select the required consumer server, and select the SSL Tab.
   c. Expand Advanced Section.
   d. Set **Hostname Verification** to **None**.
   e. Click **Save**.
   f. Restart the individual managed servers.

17.3.3 Register the WebCenter Portal: Framework Application Producer with the WebLogic Portal Consumer

Follow the steps in Section 4.3.3, "Locating and Consuming a Portlet" to register your Framework application producer with the WebLogic Portal consumer, if not already registered. Make a note of the Producer Handle that you specify (for example, `my_wc_producer`), as this will be used later.

17.3.4 Add an Authentication Mechanism To Your Portal

For information on how to add a programmatic authentication mechanism to your portal, see "Implementing Authentication Programatically" in *Oracle Fusion Middleware Security Guide for Oracle WebLogic Portal*.

17.3.5 Testing the Configuration

To test the configuration, do the following:

1. Create a portal.
2. Add the remote portlet from Section 17.3.3, "Register the WebCenter Portal: Framework Application Producer with the WebLogic Portal Consumer"to the portal.
3. Add the authentication (login) mechanism to the portal, as explained in Section 17.3.4, "Add an Authentication Mechanism To Your Portal."
4. Run the portal in a browser.
5. Log in to the portal.

6. View the portlet.

If the portlet renders correctly, the configuration is working properly.

17.4 Consumer Security for an Unsigned SAML Token Configuation

This section discusses the producer-side and consumer-side configuration required to set up security if the producer has a wss10_saml_token_policy token configured. This token is also called an unsigned SAML or simple SAML token.

The configuration steps include:

- Section 17.4.1, "Register the WebCenter Portal: Framework Application Producer with the WebLogic Portal Consumer"
- Section 17.4.2, "Add an Authentication Mechanism To Your Portal"
- Section 17.4.3, "Configuring the WebLogic Portal Consumer"
- Section 17.4.4, "Configuring the WebCenter Portal: Framework Application Producer"

17.4.1 Register the WebCenter Portal: Framework Application Producer with the WebLogic Portal Consumer

Follow the steps in Section 4.3.3, "Locating and Consuming a Portlet" to register your Framework application producer with the WebLogic Portal consumer. Make a note of the Producer Handle that you specify (for example, `my_wc_producer`), as this will be used later.

17.4.2 Add an Authentication Mechanism To Your Portal

For information on how to add a programmatic authentication mechanism to your portal, see "Implementing Authentication Programatically" in *Oracle Fusion Middleware Security Guide for Oracle WebLogic Portal*.

17.4.3 Configuring the WebLogic Portal Consumer

This section explains how to configure the consumer web application.

17.4.3.1 Add a New Policy to the Consumer Web-App

Add the following policy definition to your WebLogic Portal consumer to configure it to match the default policy configuration on a Framework application producer.

1. In your web project, create a directory `WEB-INF/classes/policies`.
2. In that directory, create a file named `wcPolicy.xml`, with the following contents:

```
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-u
tility-1.0.xsd"
  xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part">
  <wssp:Identity>
    <wssp:SupportedTokens>
      <wssp:SecurityToken
TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-prof
ile-1.0#SAMLAssertionID">
        <wssp:Claims>
```

```
          <wssp:ConfirmationMethod>sender-vouches</wssp:ConfirmationMethod>
        </wssp:Claims>
      </wssp:SecurityToken>
    </wssp:SupportedTokens>
  </wssp:Identity>
</wsp:Policy>
```

3. Save your changes to this file.

17.4.3.2 Update the Producer's Security Policy on the Consumer

1. Copy the file `WEB-INF/wsrp-consumer-security-config.xml` to your workspace and open it for editing. The procedure for copying files to your workspace is described in "Copying J2EE Library Files Into a Project" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

2. In `wsrp-consumer-security-config.xml`, add the following code to the bottom of the file, inside the <wsrp-consumer-security-config> element:

```
<!-- Setup for services producer -->
    <producer-security>
        <!-- The producer's handle -->
        <producer-handle>my_wc_producer</producer-handle>

        <!-- The policy to use when the policy is not included in the WSDL. -->
        <policy-name>wcPolicy</policy-name>
        <policy-port>{urn:oasis:names:tc:wsrp:v2:wsdl}WSRP_v2_Markup_
Service</policy-port>

<policy-port>{urn:oasis:names:tc:wsrp:v1:wsdl}WSRPBaseService</policy-port>


        <!-- When doing 8.1 compatibility, should the <wsse:security> header
-->
        <!-- be removed. -->
        <strict-compatibility>false</strict-compatibility>

        <!--
            Should 8.1 compatibility be done even if a policy is in the
            WSDL
        -->
        <!-- (9.0 producer). -->
        <compatibility-forced>false</compatibility-forced>

        <!--
            Should 8.1 compatibility be done even if a policy is NOT in
            the WSDL
        -->
        <!--
            If both compatibility-forced is true and
            compatibility-enabled false
        -->
        <!-- no compat is sent -->
        <compatibility-enabled>false</compatibility-enabled>

        <!-- Should WLP specific handlers be deployed. -->
        <!--
            EXPERT ONLY: Disabling may cause the consumer to act
            incorrectly.
        -->
        <!-- Default: true -->
```

```
        <wlp-handlers-deployed>true</wlp-handlers-deployed>

        <!-- Should anonymous users be allowed? -->
        <!-- If disabled only logged in users may use this producer. -->
        <!-- Default: true -->
        <anonymous-users-allowed>true</anonymous-users-allowed>
    </producer-security>
```

3. Populate the value of the `<producer-handle>` element with the handle you created in Section 17.4.1, "Register the WebCenter Portal: Framework Application Producer with the WebLogic Portal Consumer."
4. Populate the value of the `<policy-name>` element with the filename of the policy created Section 17.4.3.1, "Add a New Policy to the Consumer Web-App." Enter the value without its `.xml` extension (or example, `wcPolicy`).

Note: The following steps must be completed for a new web application before deploying/publishing the web application to the server.

---

5. Open the WebLogic Server Administration Console on the consumer server and log in.
6. Select **Security Realms**.
7. Select a security realm, such as **myrealm**.
8. Select the **Providers** tab.
9. Select the **Credential Mapping** tab.
10. Select the **SAML Credential Mapper**.
11. Select **Configuration**.
12. Select **Provider Specific**.
13. Set the **Issuer URI** to www.oracle.com, which is the default for Oracle WebCenter Producers.
14. Click **Save**.
15. Now you need to turn off signing. In the WLS Console, select **Security Realms**.
16. Select **myrealm**.
17. Select **Providers**.
18. Select **Credential Mapping**.
19. Select **SAML Credential Mapper**.
20. Select **Management**.
21. Select **rp_00001**.
22. Uncheck **Sign Assertion**.
23. Click **Save**.

17.4.4 Configuring the WebCenter Portal: Framework Application Producer

See the "Configuring WS-Security" in the *Oracle Fusion Middleware Administrator's Guide for Oracle WebCenter* for detailed information on securing your Framework application producer with SAML. At a minimum, the following steps are required:

1. Using Fusion Middleware Control, assign the `oracle/wss10_saml_token_service_policy` policy to all of your web application's WebServices WSRP End Points, and remove the default `no authentication service policy` from the non-markup End Points.

17.5 (Optional) Additional Configuration for a WebLogic Portal Consumer

If you have set up your WebLogic Portal producer's security to interoperate with a Framework application consumer (as explained in Section 17.2, "SAML Security Between a WebCenter Portal: Framework Application Consumer and a WebLogic Portal Producer"), and you wish to consume portlets from that producer in a WebLogic Portal consumer, then the following steps are required:

1. Section 17.5.1, "Register the WebLogic Portal producer with the WebLogic Portal Consumer"
2. Section 17.5.2, "Update the Producer's Security Policy on the Consumer"

17.5.1 Register the WebLogic Portal producer with the WebLogic Portal Consumer

Follow the steps in Section 4.3.3, "Locating and Consuming a Portlet" to register your Framework application producer with the WebLogic Portal consumer. Make a note of the Producer Handle that you specify (for example, `my_wc_producer`), as this will be used later.

17.5.2 Update the Producer's Security Policy on the Consumer

1. Copy the file `WEB-INF/wsrp-consumer-security-config.xml` to your workspace and open it for editing. The procedure for copying files to your workspace is described in "Copying J2EE Library Files Into a Project" in the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.
2. Add a new `<producer-security>` element with the following contents:

```
<producer-security>
            <!-- The producer's handle -->
            <producer-handle>my_wlp_producer</producer-handle>

           <!-- The policy to use when the policy is not included in the WSDL.
-->
            <policy-name>wsrp81compatPolicy</policy-name>

            <!-- When doing 8.1 compatibility, should the <wsse:security>
header -->
            <!-- be removed. -->
            <strict-compatibility>false</strict-compatibility>

            <!-- Should 8.1 compatibility be done even if a policy is in the
WSDL -->
            <!-- (9.0 producer). -->
            <compatibility-forced>false</compatibility-forced>

            <!-- Should 8.1 compatibility be done even if a policy is NOT in
the WSDL -->
```

```
            <!-- If both compatibility-forced is true and compatibility-enabled
false -->
             <!-- no compat is sent -->
             <compatibility-enabled>true</compatibility-enabled>

             <!-- Should WLP specific handlers be deployed. -->
             <!-- EXPERT ONLY: Disabling may cause the consumer to act
incorrectly. -->
             <!-- Default: true -->
             <wlp-handlers-deployed>false</wlp-handlers-deployed>

             <!-- Should anonymous users be allowed? -->
             <!-- If disabled only logged in users may use this producer. -->
             <!-- Default: true -->
             <anonymous-users-allowed>true</anonymous-users-allowed>
        </producer-security>
```

3. Populate the value of the `<producer-handle>` element with the handle that was created earlier in Section 17.5.1, "Register the WebLogic Portal producer with the WebLogic Portal Consumer."
4. Save the changes, and republish the application.

17.5.3 Create a New PKI Credential Mapping to the Consumer

This section explains how to create a new PKI credential mapping to the consumer, if one is not already present.

1. Follow the instructions "Create PKI Credential Mappings" in the *WebLogic Server Administration Console Online Help* to create a new security credential map on the consumer for the producer. Supply the following values as appropriate:

Parameter	Value
Protocol	Leave this field blank.
Remote Host	Leave this field blank.
Remote Port	Leave this field blank.
Path	Leave this field blank.
Method	Leave this field blank.
Credential Type	Key Pair
Principal Name	Enter the value of the `<consumer-name>` element in `WEB-INF/wsrp-consumer-security-config.xml`.
Principal Type	User
Credential Action	Leave this field blank.
Keystore Alias	The key alias. See Section 17.2.1.1, "Generate a Key Pair."
Password	The key passphrase. See Section 17.2.1.1, "Generate a Key Pair."
Confirm Password	The key passphrase. See Section 17.2.1.1, "Generate a Key Pair."

18

Adding Remote Resources to the Library

The WebLogic Portal Administration Console lets you locate producers and add their remote resources to the Portal Resources Library. Remote resources can include books, pages, and portlets. When a remote resource is added to the Library, it becomes available to you to incorporate into a portal desktop.

> **Tip:** This chapter assumes that you are familiar with the Portal Resources Library and how to use it to assemble WebLogic Portal desktops. For detailed information on the Library and on assembling portals using the Administration Console, see the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal* This chapter also assumes you are familiar with basic federated portal concepts and terms, such as producer, consumer, and WSDL. For detailed information on federated portals, see Chapter 2, "What are Federated Portals?" and Chapter 3, "Federated Portal Architecture."

This chapter explains how to locate producers and incorporate their remote resources into the Portal Resources Library. The chapter includes these sections:

18.1 Introduction

You can use the WebLogic Portal Administration Console to locate remote producers, discover the resources they offer, and add them to the Portal Resources Library. After a remote resource, such as a book, page, or portlet, is added to the Library, you can add the resource to a desktop just as you would a local book, page, or portlet.

The primary advantage of remote books and pages is that they act as containers for other remote resources. For example, a producer can offer a remote book that contains several remoteable pages, each of which contain multiple remoteable portlets. When you consume that book, the remoteable pages and portlets it contains are consumed as well, with no additional steps.

> **Tip:** The term remoteable refers to a book, page, or portlet that is deployed in a producer application and that is offered as remote. Producer application developers decide whether or not books, pages, and portlets they create are offered as remote. For detailed information on creating remoteable pages and books in a producer application, see Chapter 6, "Offering Books, Pages, and Portlets to Consumers."

After you consume a remote book or page, an administrator can edit it using the Administration Console. For example, an administrator can add other portlets, books, or pages to the remote book or page. Remember that such changes are not reflected back to the producer; therefore, after a remote book or page is modified on the consumer, it can become inconsistent with the original book, page, or portlet in the producer application.

To add remote books, pages, and portlets to the Library:

1. Locate and add the producer in which the remote resources are deployed.
2. If necessary, register the producer.
3. Add remote books, pages, and portlets to your Portal Resources Library.

After the remote resources are in the Library, you add them to your portal desktop as you would any other book, page, or portlet.

18.2 Adding a Producer

To consume remote resources, such as portlets, books, and pages that are deployed in a producer, you need to first add the producer to your Portal Resources Library. After you add a WSRP-compliant producer to the Portal Resources Library, you can make that producer's remoteable resources available for consumption by your portal.

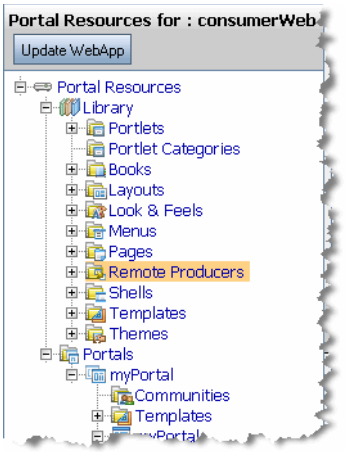
During registration, the producer stores information about the consumer and returns a handle to the consumer. Registration is an optional feature described in the WSRP specification. A WebLogic Portal complex producer implements this option and, therefore, requires consumers to register before discovering and interacting with portlets offered by the producer. See Section 3.4.2.2, "Complex Producers" for more information.

> **Tip:** In the WebLogic Portal Administration Console, producer registrations are scoped to individual consumer web applications. Because there can be multiple consumer web applications in an enterprise application, it is possible that a given producer will need to be registered multiple times within an enterprise application (that is, registered for each consumer web application in which it is used).

To locate and register a producer using the Administration Console:

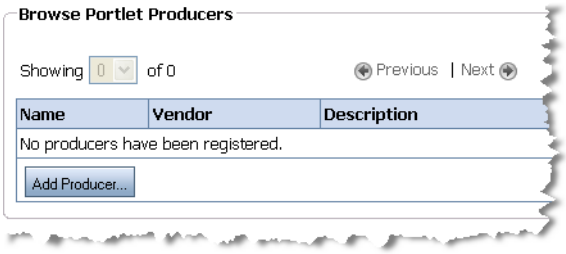
1. Expand the Library node in the Portal Resources tree and select **Remote Producers**, as shown in Figure 18–1.

Figure 18–1 Selecting Remote Producers



2. In the Browse Remote Producers window, select **Add Producer**, as shown in Figure 18–2. The Add Producer wizard appears.

Figure 18–2 Select Add Producer

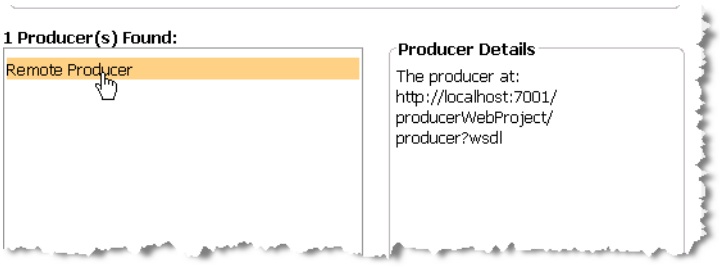


3. In the Add Producer wizard, select a producer. To do this, specify a producer directly by entering its WSDL URL, and click **Search**. For example:

   `http://myhost:7001/producerWebProject/producer?wsdl`

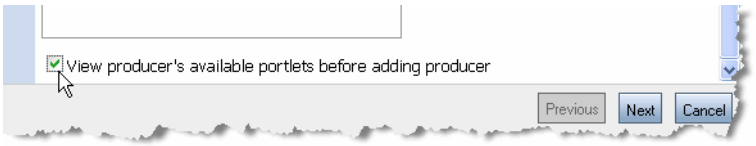
4. Select the producer you wish to add from the Producer(s) Found list, as shown in Figure 18–3.

Figure 18–3 Selecting a Producer



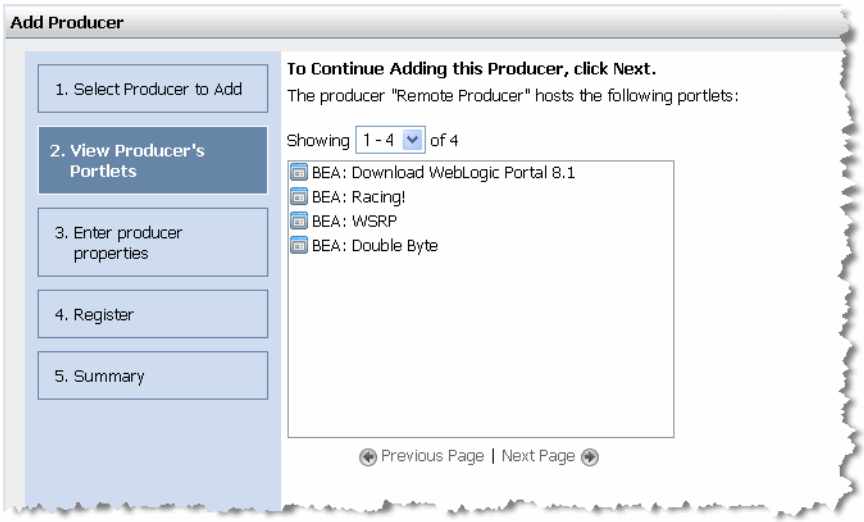
5. If you want to view a list of portlets hosted by the producer, select the **View producer's portlets before adding producer** checkbox, as shown in Figure 18–4.

Figure 18–4 View Producer's Portlets Checkbox



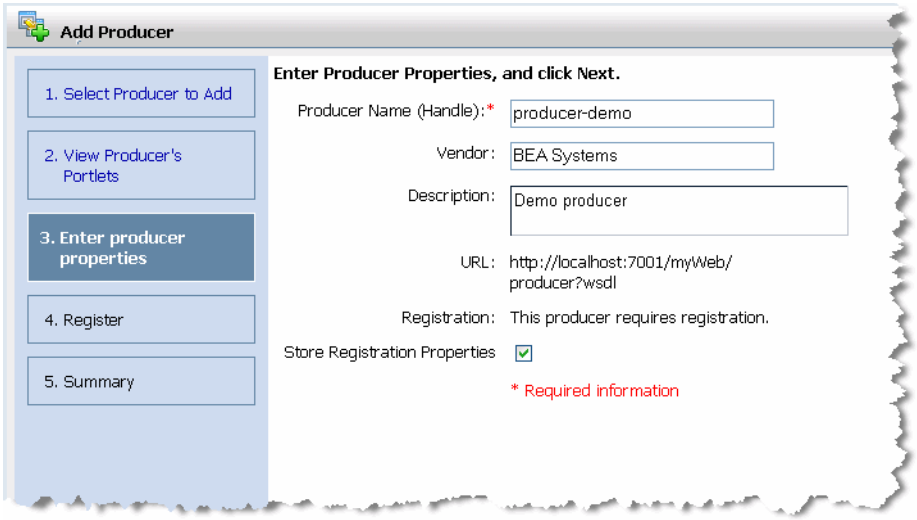
6. Click **Next**.
7. If the View Producer Portlets dialog appears, click **Next**. This dialog, shown in Figure 18–5, appears only if you selected the **View producer's portlets before adding producer** checkbox. This dialog simply lists the portlets hosted by the selected producer to help you decide if you want to add the producer or not.

Figure 18–5 View Producer's Portlets



8. In the Enter Producer Properties dialog (Figure 18–6), enter or select:
   - A name for the producer – (Required) This name is used by the consumer to identify the producer.
   - Vendor and description information – (Optional) Use these fields to enter optional metadata.
   - Store Registration Properties – (Optional) If you select this option, the registration properties that you enter in the Add Producer dialog are stored on the consumer. It is recommended that you select this option. For detailed information on this option, see Section 14.19, "Storing Registration Properties".

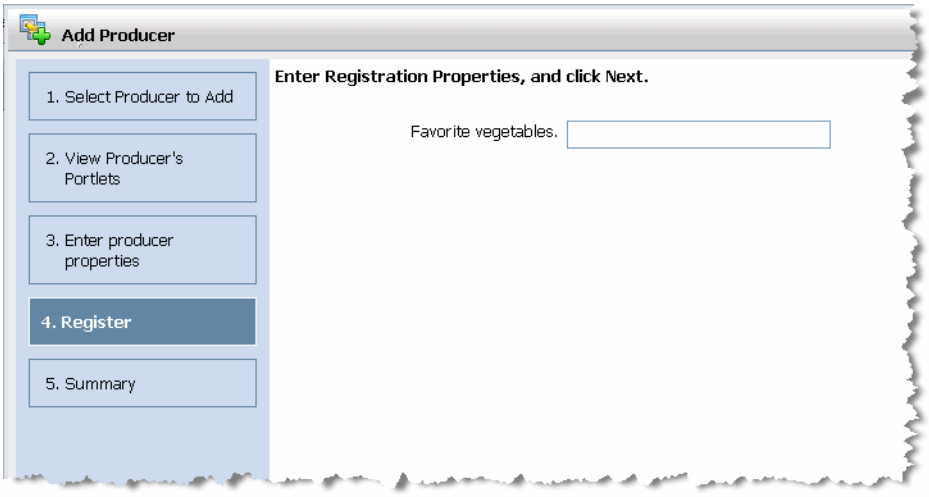
Figure 18–6 Enter Producer Name



9. Click **Add Producer** to go to the Register dialog.

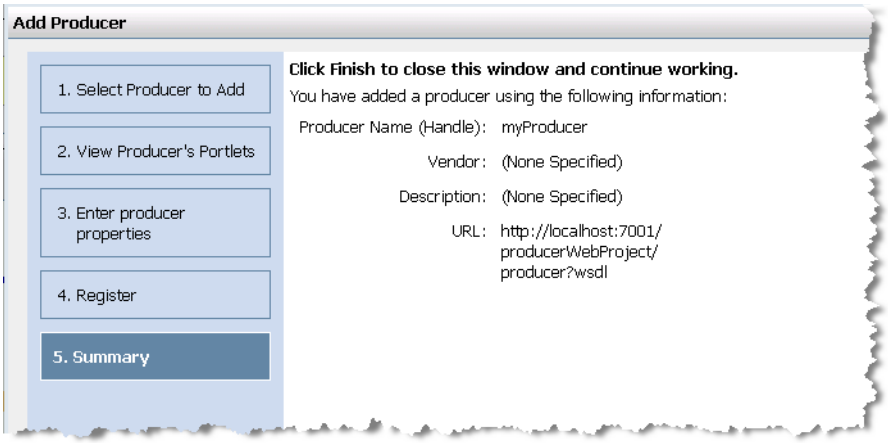
10. In the Register dialog, enter the registration information, if any is required. In the example shown in Figure 18–6, a property set called "Favorite vegetables" was defined on the producer. The dialog requires the user to enter value(s) from that property set. The value(s) entered here are validated on the producer and are used to determine which resources this consumer is entitled to retrieve. For detailed information on consumer entitlement, see Chapter 11, "Consumer Entitlement."

Figure 18–7 Enter Registration Properties (Sample)



11. Click **Add Producer**. The Summary dialog appears, as shown in Figure 18–8.

Figure 18–8 Summary Dialog



12. Click **Finish**.

    Checkpoint: Now that you have located and added a producer, you can view and select portlets, books, and pages to add to the consumer from that producer, as explained in the following sections.

18.3 Adding a Remote Portlet to the Portal Library

If you have added a producer that contains a remoteable portlet, you can add that portlet to your Portal Resources Library. After the remote portlet is added to the Library, you can incorporate the portlet into a page in your portal desktop.

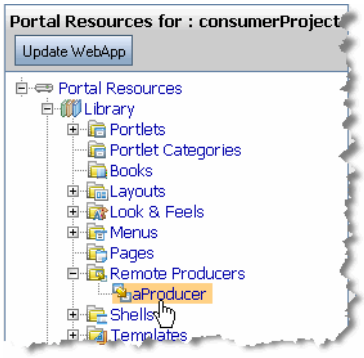
There are two ways to incorporate remote portlets into a portal using the Administration Console:

- Add a remote page that contains one or more remote portlets. For details adding remote pages, see Section 18.4, "Adding a Remote Page to the Portal Library".
- Add a remote portlet directly. This method is described in this section.

To add a remote portlet to your Portal Resources Library directly:

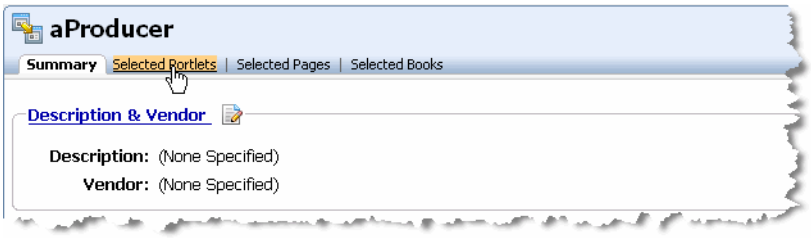
1. Open the WebLogic Portal Administration Console.
2. If you haven't done so, locate and add the producer that contains the remote portlet(s) that you want to add to your portal. The procedure for adding a producer is explained in Section 18.2, "Adding a Producer".
3. In the Portal Resources tree, open the Library > Remote Producers folder, and select the producer that contains the remote portlet that you want to use, as shown in Figure 18–9.

Figure 18–9 Selecting a Producer



4. In the producer window click the **Selected Portlets** tab, as shown in Figure 18–10.

Figure 18–10 Selected Portlets Tab



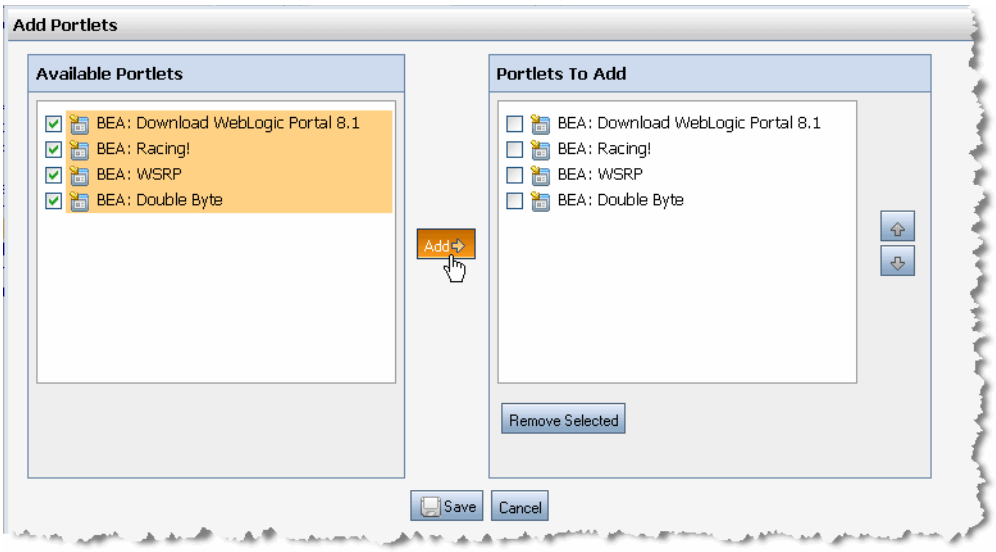
5. In the Browse Selected Portlets panel, click **Add Portlets**, as shown in Figure 18–11.

> **Tip:** If the producer offers a large number of portlets, use the Search feature to narrow the selections. For instance, you can search for all portlets that begin with "a," and only those portlets will show up in the Browse Selected Portlets table.

Figure 18–11 Add Portlet Button

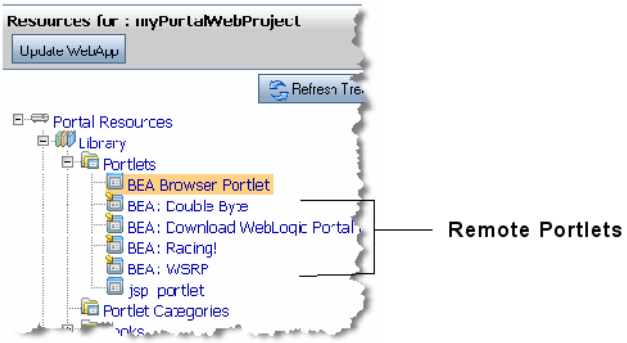
6. In the Add Portlets dialog, select the remote portlet(s) that you want to add to the Library, and click **Add** to move the selected portlets to the Portlets To Add column, as shown in Figure 18–12.

Figure 18–12 Selecting Portlets to Add



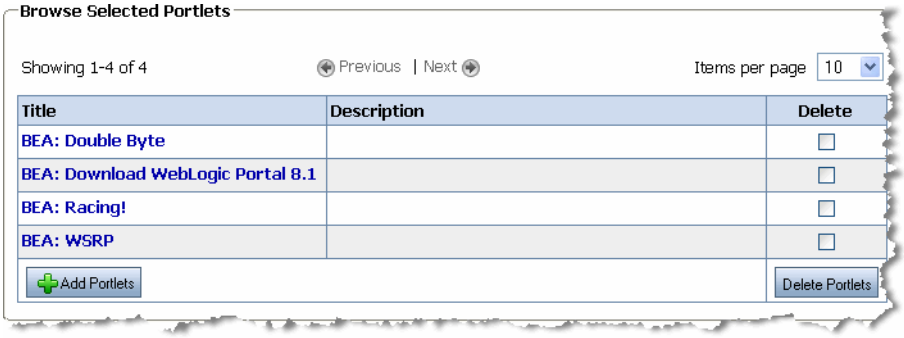
7. After moving the portlet to the Portlets To Add column, click **Save**. The portlets you added appears in the Library under the Portlets folder, as shown in Figure 18–13.

Figure 18–13 Remote Portlets Added to the Library



The added portlets also appear in the Browse Selected Portlets table in the producer's Selected Portlets tab, as shown in Figure 18–14.

Figure 18–14 Table Displays Added Portlets



> **Tip:** When you add a remote portlet to the Library, it is placed in the Portlets folder. This is the same folder where local portlets appear. WebLogic Portal treats the remote portlet exactly as if it were a local portlet.

Checkpoint: You can now add the portlet to a page in your desktop. For details on adding Library resources to a desktop, see the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

18.4 Adding a Remote Page to the Portal Library

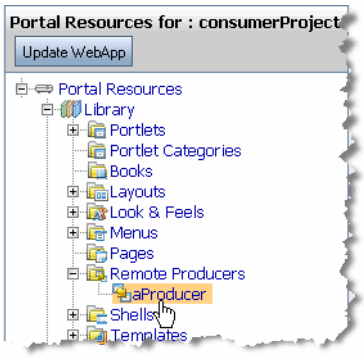
If you have added a producer that contains a remoteable page, you can add that page to your Portal Resources Library. After the remote page is added to the Library, you can incorporate it into your portal desktop as if it were a local page.

This section explains how to add a remote page to your Portal Resources Library.

> **Tip:** To be remoteable, the page's Offer As Remote property must have been set to true when it was created and the page must include some content. A remote page can contain any combination of remote books and portlets. Books and portlets contained within a remote page must be offered as remote. By default, books, pages, and portlets are offered as remote. For more information on creating remoteable books and pages in a producer application, see Chapter 18, "Adding Remote Resources to the Library."

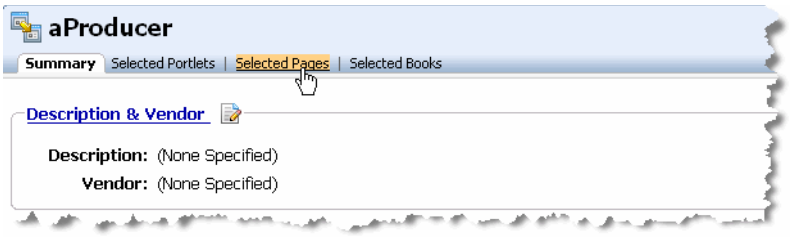
1. Open the WebLogic Portal Administration Console.
2. If you haven't done so, locate and add the producer that contains the remote page(s) that you want to add to your portal. The procedure for adding a producer is explained in Section 18.2, "Adding a Producer".
3. In the Portal Resources tree, open the Library > Remote Producers folder, and select the producer that contains the remote page that you want to use, as shown in Figure 18–15.

Figure 18–15 Selecting a Producer



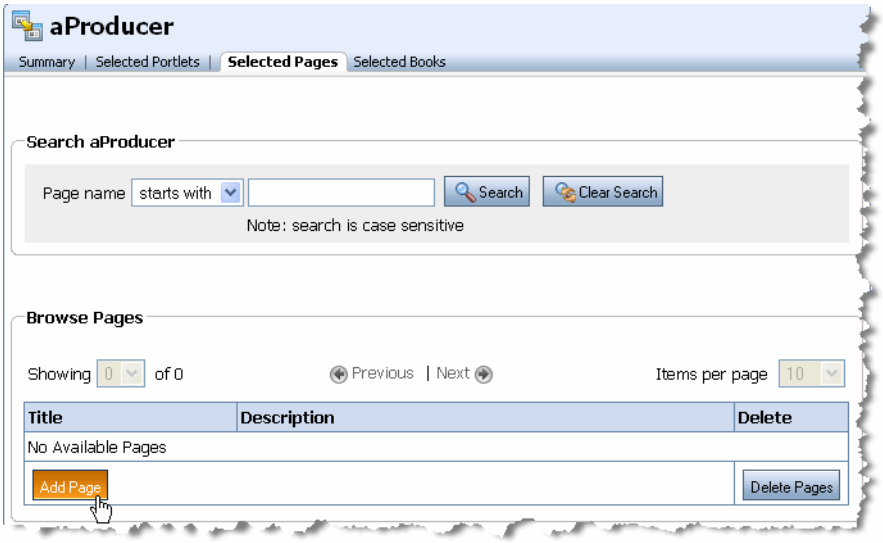
4. In the producer window, click the **Selected Pages** tab, as shown in Figure 18–16.

Figure 18–16 Selected Pages Tab



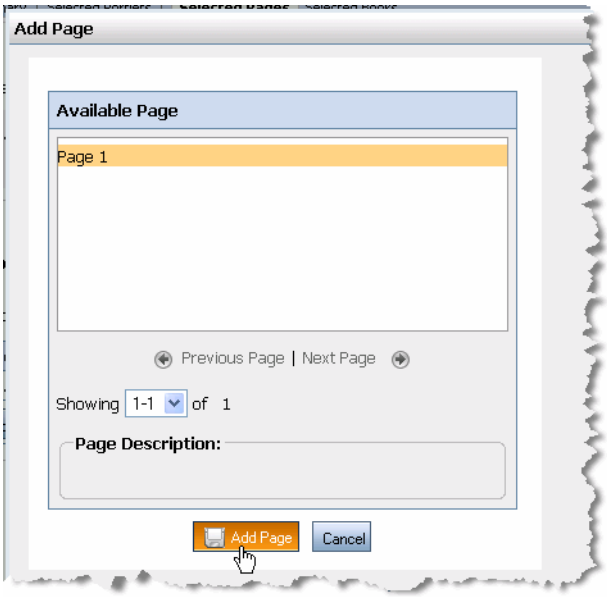
5. In the Browse Pages section, click **Add Page,** as shown in Figure 18–17.

Figure 18–17 Add Page Button



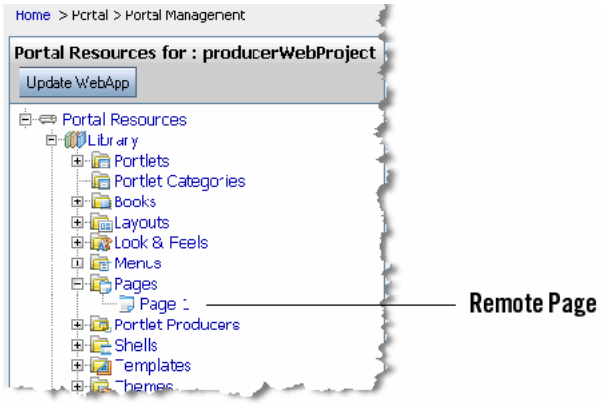
6. In the Add Page dialog, select the remote page that you want to add to the Library, and click **Add Page**. In Figure 18–18, the remote page is called Page 1.

Figure 18–18 The Add Page Dialog



Checkpoint: The remote page is added to the Library, as shown in Figure 18–24. You can now add the page to a desktop. For details on adding Library resources to a desktop, see the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

Figure 18–19 Remote Page Added to Library



18.5 Adding a Remote Book to the Portal Library

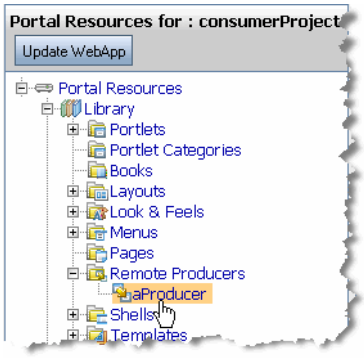
If you have added a producer that contains a remoteable book, you can add that book to your Portal Resources Library. After the remote book is added to the Library, you can incorporate it into your portal desktop as if it were a local book.

> **Tip:** To be remoteable, the book's Offer As Remote property must have been set to true when it was created, and the book must include some content. A remote book can contain any combination of remote pages and portlets. Pages and portlets contained within a remote page must be offered as remote. By default, books, pages, and portlets are offered as remote. For more information on creating remoteable books and pages in a producer application, see Chapter 18, "Adding Remote Resources to the Library."

This section explains how to add a remote book to your Portal Resources Library.

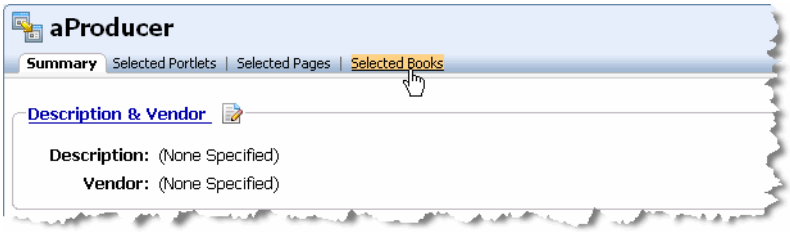
1. Open the WebLogic Portal Administration Console.
2. If you haven't done so, locate and add the producer that contains the remote book(s) that you want to add to your portal. The procedure for adding a producer is explained in Section 18.2, "Adding a Producer".
3. In the Portal Resources tree, open the Library > Remote Producers folder, and select the producer that contains the remote book that you want to use, as shown in Figure 18–20.

Figure 18–20 Selecting a Producer



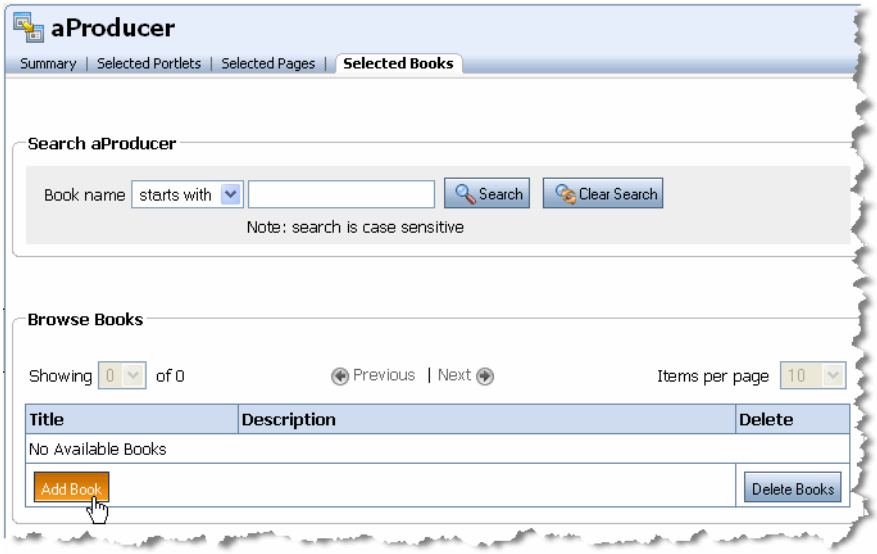
4. In the producer window, click the **Selected Books** tab, as shown in Figure 18–21.

Figure 18–21 Selected Books Tab



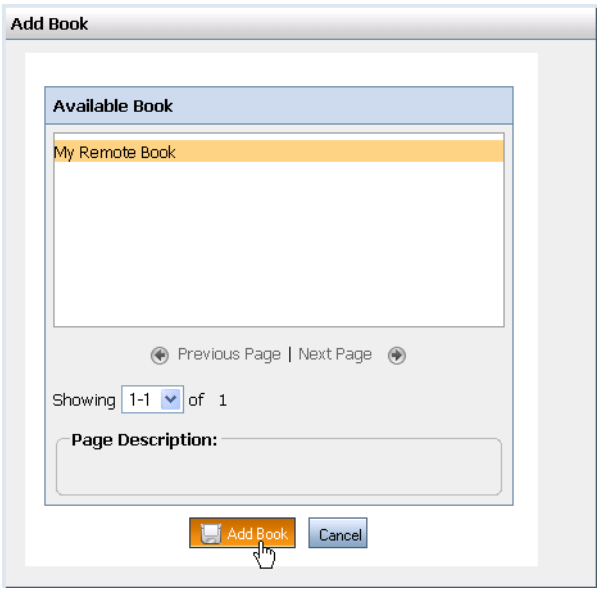
5. In the Browse Books section, click **Add Book**, as shown in Figure 18–22.

Figure 18–22 Add Book Button



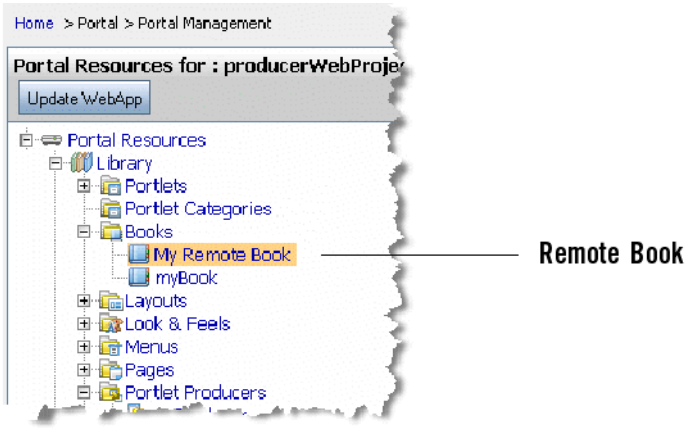
6. In the Add Book dialog, select the remote book that you want to add to the Library, and click **Add Book**. In Figure 18–23, the remote book is called My Remote Book.

Figure 18–23 The Add Book Dialog



Checkpoint: The remote book is added to the Library, as shown in Figure 18–24. You can now add the book to a desktop. For details on adding Library resources to a desktop, see the *Oracle Fusion Middleware Portal Development Guide for Oracle WebLogic Portal*.

Figure 18–24 Remote Book Added to Library



19 Configuring Two-Way SSL

To connect to servers configured for the two-way SSL communication, clients like Oracle Enterprise Pack for Eclipse and WSRP consumers must supply a certificate and a private key to the producers residing on these servers. These clients can provide certificates and private keys through SSL interceptors. This chapter describes how to configure SSL interceptors using Oracle Enterprise Pack for Eclipse. WSRP consumers can also use the interceptors discussed in this chapter.

For information about configuring two-way SSL, see http://download.oracle.com/docs/cd/E15523_01/apirefs.1111/e13952/taskhelp/security/ConfigureTwowaySSL.html .

This chapter contains the following sections:

- Section 19.1, "Creating the WSDL and SOAP Interceptors"
- Section 19.2, "Configuring Producers to Use SSL for All Ports"
- Section 19.3, "Configuring WebLogic Portal to Use Interceptors"
- Section 19.4, "Configuring Oracle Enterprise Pack for Eclipse to use Interceptors"

19.1 Creating the WSDL and SOAP Interceptors

This section describes how to create WSDL and SOAP SSL interceptors to enable clients like Oracle Enterprise Pack for Eclipse to connect to the servers that host producers and are configured for two-way SSL communication:

- IWSDLInterceptor: Fetches the WSDL as well as imported WSDLs and XSDs from the producer.
- ISOAPInterceptor: Used on all SOAP calls to the producer including non-markup, such as Service Description. .

To create these interceptors using Oracle Enterprise Pack for Eclipse (the IDE), perform the steps described in the following sections:

- Section 19.1.1, "Creating a Java Project"
- Section 19.1.2, "Creating a Java Package"
- Section 19.1.3, "Creating a Java Class"
- Section 19.1.4, "Creating a JAR File"

19.1.1 Creating a Java Project

To create a Java project that you will use for WSDL and SOAP interceptors:

1. Start Oracle Enterprise Pack for Eclipse. You can run the executable file *<MW_HOME>*`/oepe_11gR1PS3/eclipse/eclipse.exe`. On Windows, you can also start the IDE from the Start menu by selecting **Start** > **My Programs** > **Oracle WebLogic** > **Eclipse for WebLogic 10.3.6**.
2. From the **File** menu, select **New**, then **Java Project**.
3. In the Create a Java Project dialog, enter a meaningful name for your project, for example `Interceptors`, then click **Next**.
4. In the **Libraries** tab, click **Add External JARs**.
5. In the JAR Selection dialog, select the following JARs:
   - *<MW_HOME>*`/wlserver_10.3/server/lib/weblogic.jar`
   - *<MW_HOME>*`/patch_wlp1032/patch_jars/wsrp-client.jar`
6. Click **Finish**.

19.1.2 Creating a Java Package

To create a Java package in which you will create a Java class for WSDL and SOAP interceptors:

1. In the Package Explorer, ensure that the appropriate project is active. In this example, the project is called `Interceptors`.
2. From the **File** menu, select **New**, then **Package**.
3. In the New Java Package dialog box, in the **Name** field, enter `com.bea.wsrp.qa.sampl`, and click **Finish**.

19.1.3 Creating a Java Class

To create a Java class that implements the `IWSDLInterceptor` and `ISOAPInterceptor` methods:

1. In the Package Explorer, ensure that the package **com.bea.wsrp.qa.sampl** is selected.
2. From the **File** menu, select **New**, then **Class**.
3. In the Java Class dialog box, in the **Name** field, enter `EchoWsdlSoapInterceptor`, and click **Finish**.
4. In the Package Explorer, under the **Interceptor** project, double-click **EchoWsdlSoapInterceptor.java** to open it, if it is not already open.
5. In the **Javadoc** tab for `EchoWsdlSoapInterceptor.java`, add the sample code provided in Example 19–1.

Example 19–1 EchoWsdlSoapInterceptor

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.Proxy;
import java.net.URL;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
```

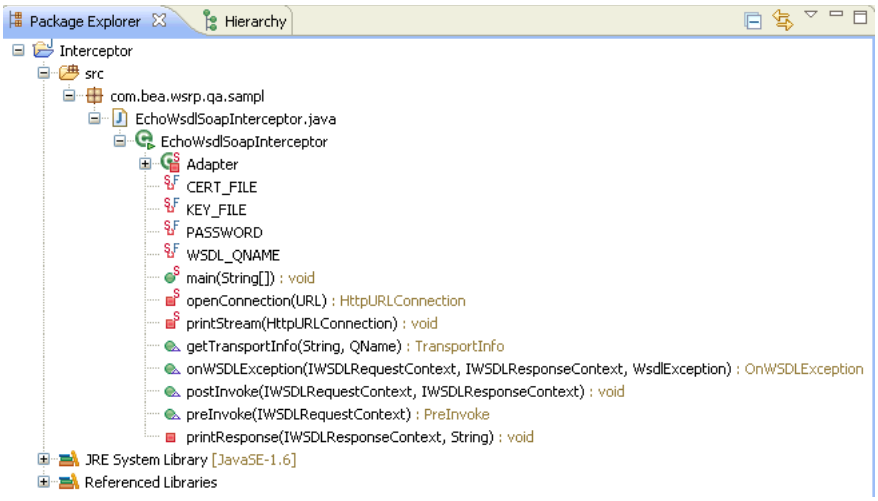
```
import java.security.UnrecoverableEntryException;
import javax.xml.namespace.QName;

import weblogic.wsee.connection.transport.TransportInfo;
import weblogic.wsee.connection.transport.https.HttpsTransportInfo;
import weblogic.wsee.connection.transport.https.SSLAdapter;
import weblogic.wsee.wsdl.WsdlException;
import weblogic.net.http.HttpsURLConnection;

//import com.bea.net.http.HttpsURLConnection;
import com.bea.wsrp.consumer.soap.ISOAPInterceptor;
import com.bea.wsrp.consumer.wsdl.IWSDLInterceptor;
import com.bea.wsrp.consumer.wsdl.IWSDLRequestContext;
import com.bea.wsrp.consumer.wsdl.IWSDLResponseContext;

public class EchoWsdlSoapInterceptor implements IWSDLInterceptor, ISOAPInterceptor
{
private static final File CERT_FILE = new File("/home/nlipke/wl/src1034GA_wlp_
16014jr/bea/user_projects/domains/cs2_domain/certfile.cer.pem");
private static final File KEY_FILE = new File("/home/nlipke/wl/src1034GA_wlp_
16014jr/bea/user_projects/domains/cs2_domain/keyfile.key.pem");
private static final char[] PASSWORD = "password".toCharArray();

private static final QName WSDL_QNAME = new
QName("http://schemas.xmlsoap.org/wsdl/", "definitions");

@Override
public void postInvoke(IWSDLRequestContext requestCtx, IWSDLResponseContext
responseCtx) throws IOException {
System.out.println("postInvoke: " + requestCtx.getWsdlUrl());

printResponse(responseCtx, "postInvoke");
responseCtx.setV1MarkupPortUrl(null);
responseCtx.setV1ServiceDescriptionPortUrl(null);
responseCtx.setV1RegistrationPortUrl(null);
responseCtx.setV1PortletManagementPortUrl(null);
responseCtx.setV1WlpEntensionMarkupPortUrl(null);
}
private void printResponse(IWSDLResponseContext responseCtx, String method) {
System.err.println(method + ": " + responseCtx.getV1MarkupPortUrl());
System.err.println(method + ": " + responseCtx.getV1ServiceDescriptionPortUrl());
System.err.println(method + ": " + responseCtx.getV1PortletManagementPortUrl());
System.err.println(method + ": " + responseCtx.getV1RegistrationPortUrl());
System.err.println(method + ": " + responseCtx.getV1WlpEntensionMarkupPortUrl());

System.err.println(method + ": " + responseCtx.getV2MarkupPortUrl());
System.err.println(method + ": " + responseCtx.getV2ServiceDescriptionPortUrl());
System.err.println(method + ": " + responseCtx.getV2PortletManagementPortUrl());
System.err.println(method + ": " + responseCtx.getV2RegistrationPortUrl());
System.err.println(method + ": " + responseCtx.getV2WlpEntensionMarkupPortUrl());

}

@Override
public PreInvoke preInvoke(IWSDLRequestContext requestCtx) throws IOException {
String wsdlUrl = requestCtx.getWsdlUrl();
System.err.println("preInvoke: " + wsdlUrl);
if (wsdlUrl.startsWith("http://")) {
System.err.println("got one! " + wsdlUrl);
wsdlUrl = wsdlUrl.replaceFirst("http://", "https://");
```

```
}
wsdlUrl = wsdlUrl.replaceFirst("7001", "7002");
requestCtx.setWsdlUrl(wsdlUrl);
System.err.println("preInvoke: " + wsdlUrl);

requestCtx.setTransportInfo(getTransportInfo(wsdlUrl, WSDL_QNAME));
return PreInvoke.FETCH_WSDL;
}

@Override
public OnWSDLException onWSDLException(IWSDLRequestContext requestCtx,
IWSDLResponseContext responseCtx, WsdlException e) throws IOException {
System.err.println("onWSDLException: " + requestCtx.getWsdlUrl());
e.printStackTrace();
printResponse(responseCtx, "onWSDLException");

// TODO Auto-generated method stub
return OnWSDLException.ABORT_WITH_FAILURE;
}

@Override
public TransportInfo getTransportInfo(String url, QName methodName) {
System.err.println("getTransportInfo: " + url + ", " + methodName);
final HttpsTransportInfo httpsTransportInfo = new HttpsTransportInfo();
httpsTransportInfo.setSSLAdapter(new Adapter());
return httpsTransportInfo;
}
private static class Adapter implements SSLAdapter {

@Override
public HttpURLConnection openConnection(URL url, Proxy proxy,
TransportInfo info) throws IOException {
System.err.println("openConnection: " + url + ", " + proxy + ", " + info);
return EchoWsdlSoapInterceptor.openConnection(url);
}

@Override
public void setClientCert(String arg0, char[] arg1)
throws KeyStoreException, NoSuchAlgorithmException,
UnrecoverableEntryException {
System.err.println("setClientCert");
}

@Override
public void setKeystore(String arg0, char[] arg1, String arg2) {
System.err.println("setKeystore");
}

}

private static HttpURLConnection openConnection(URL url) throws
FileNotFoundException {
final HttpsURLConnection connection = new HttpsURLConnection(url);
connection.loadLocalIdentity(new FileInputStream(CERT_FILE), new
FileInputStream(KEY_FILE), PASSWORD);
return connection;
}
public static void main(String[] args) throws Exception {
final HttpURLConnection connection = openConnection(new
URL("https://localhost:7002/www/producer?WSDL"));
```

```
printStream(connection);
}

private static void printStream(final HttpURLConnection connection)
throws IOException {
InputStream inputStream = connection.getInputStream();
int c;
while ((c = inputStream.read()) != -1) {
System.out.print((char) c);
}
}
}
```

6. Save your project. The Package Explorer should look like Figure 19–1.

Figure 19–1 EchoWsdlSoapInterceptor Java Class in the Package Explorer



19.1.4 Creating a JAR File

You need to create a JAR file from the Interceptors project that includes the `EchoWsdlSoapInterceptor` class. You will need this JAR file later to import the Interceptors project into a fragment project.

To create a JAR file:

1. In the Package Explorer, ensure that the appropriate project is active. In this example, the project is called `Interceptors`.
2. From the **File** menu, select **Export**.
3. In the Export dialog box, under **Select**, expand **Java**, then select **JAR file**, and click **Next**.
4. In **JAR File Specification**, ensure that the appropriate project and directories are selected.
5. Under **Select the export destination**, in the **JAR file** field, specify the directory in which you want to create the JAR file.
6. Click **Finish**.

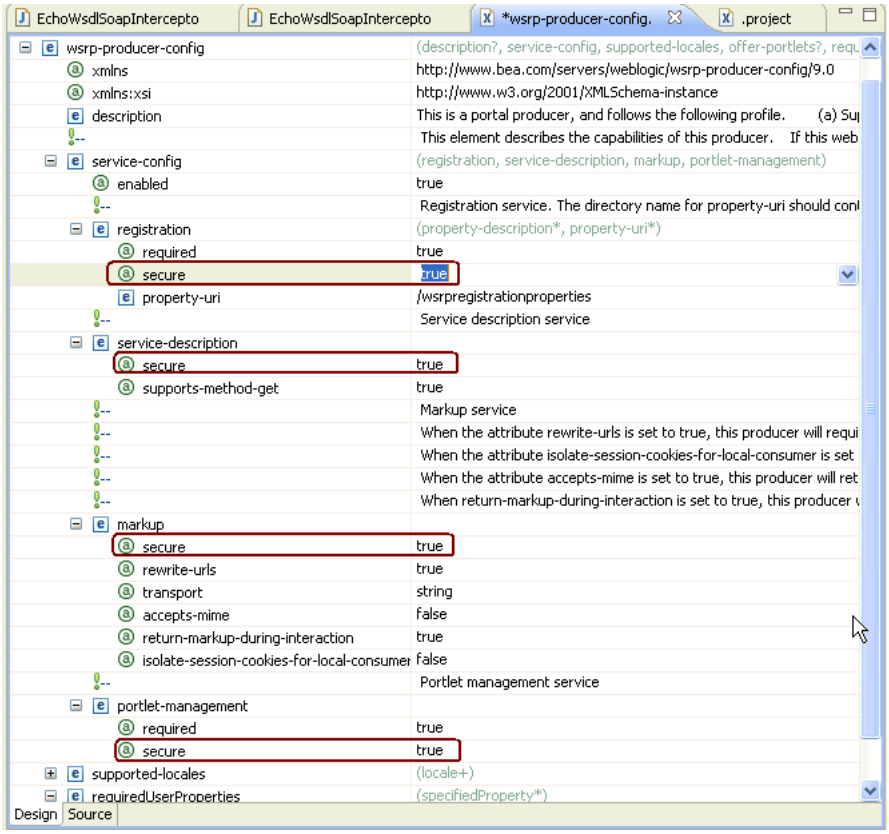
19.2 Configuring Producers to Use SSL for All Ports

By configuring security on WSRP producers, you enable them to accept certificates and primary keys from WSDL and SOAP interceptors. As a result, clients like Oracle Enterprise Pack for Eclipse and WSRP consumers can successfully communicate with the producers.

To configure your producers to use SSL for all ports:

1. In the Package Explorer, ensure that the appropriate project is active and select the **Merged Project** tab.
2. Copy the `wsrp-producer-config.xml` file located in the `WEB-INF` directory under the Merged Projects tab, to your web application.
   a. Expand the `WEB-INF` directory and select the `wsrp-producer-config.xml` file.
   b. Right-click and select **Copy to Project**.
3. Open the `WEB-INF/wsrp-producer-config.xml`file.
4. For each port , set the **secure** property to `true` by selecting **true** from the dropdown list, as shown in Figure 19–2.

Figure 19–2 The secure Property in wsrp-producer-config.xml



5. Save the file.

19.3 Configuring WebLogic Portal to Use Interceptors

19.4 Configuring Oracle Enterprise Pack for Eclipse to use Interceptors

Once you have created the SSL (SOAP and WSDL) interceptors, you must configure Oracle Enterprise Pack for Eclipse to use these interceptors to communicate with producers that are enabled with two-way SSL.

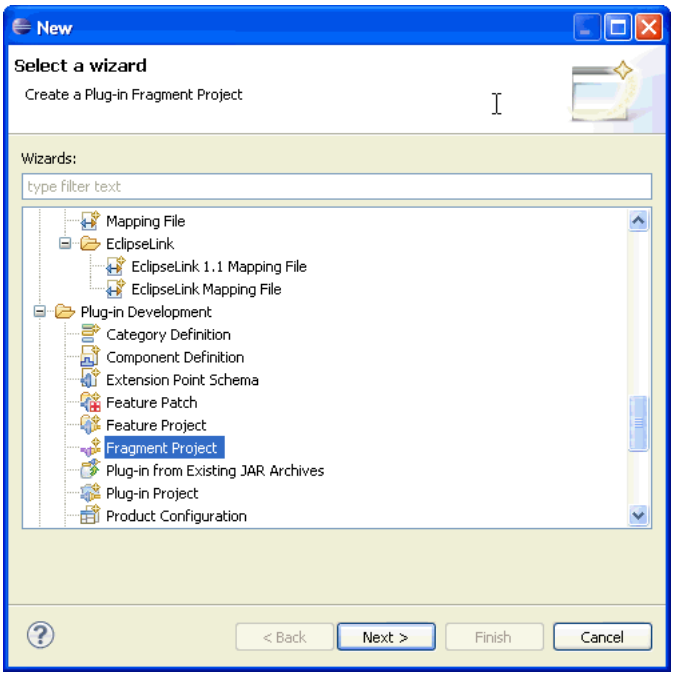
This section includes the following subsections:

19.4.1 Creating a Fragment Project

To configure Oracle Enterprise Pack for Eclipse to use the `EchoWsdlSoapInterceptor` Java class:

1. In the Package Explorer, ensure that the appropriate project is active. In this example, the project is called `Interceptors`.
2. From the **File** menu, select **New**, then **Others**.
3. In the New dialog box, under **Wizards**, expand **Plug-in Development**, then select **Fragment Project**, and click **Next**.

Figure 19–3 Fragment Project



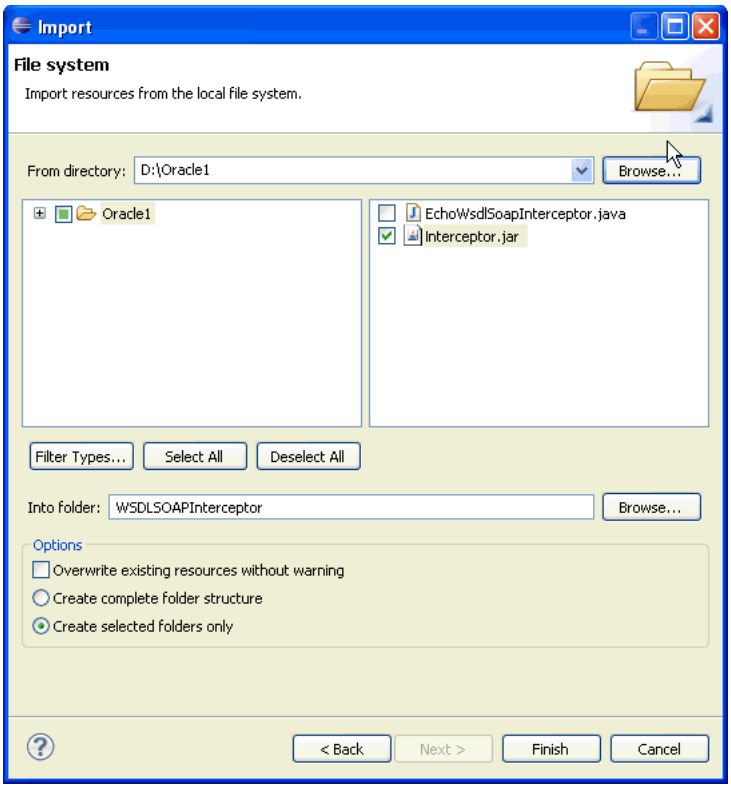
4. In the New Fragment Project dialog box, in the **Project name** field, enter a name for your project, for example `WSDLSOAPInterceptor`, and click **Next**.
5. Under **Host Plug-in**, in the **Plug-in ID** field, enter `com.bea.wlp.eclipse.wsrp`, and click **Finish**.

19.4.2 Importing the JAR file into the Fragment Project

To import the `Interceptors.jar` file that you created in Section 19.1.4, "Creating a JAR File":

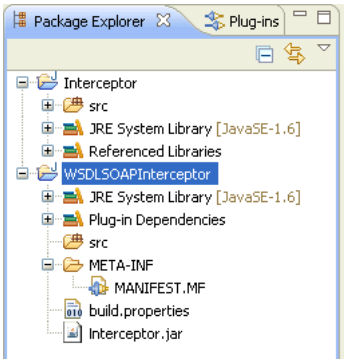
1. In the Package Explorer, ensure that the appropriate project is active. In this example, the project is called `WSDLSOAPInterceptors`.
2. From the **File** menu, select **Import**.
3. In the Import dialog box, under **Select an import source**, expand **General** and select **File System**, then click **Next**.
4. Next to the **From directory** field, click **Browse**.
5. In the Import from directory dialog box, select the interceptors JAR file. Select the checkbox against the interceptors JAR option in the column on the right side, as shown in Figure 19–4.

Figure 19–4 Import of Interceptors JAR



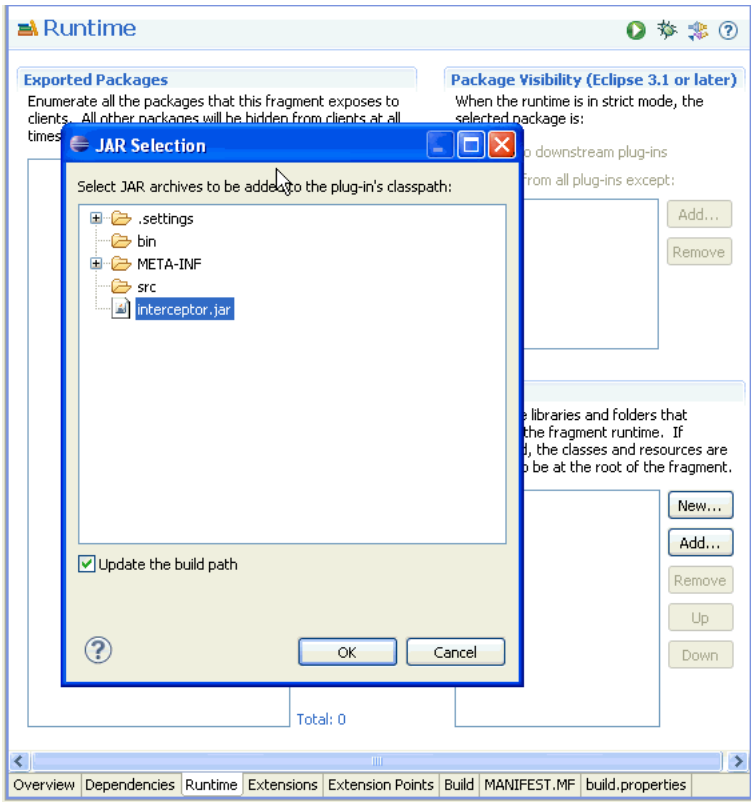
6. Click **Finish**. The Package Explorer should look like Figure 19–5.

Figure 19–5 Interceptors JAR in the Package Explorer



7. Select the **Runtime** tab of the fragment project.
8. In the **Classpath** section, click **Add**.
9. In the JAR Selection dialog, select the interceptors JAR file (Figure 19–6) and click **OK**.

Figure 19–6 Interceptors JAR in the Classpath



10. Save your project.

19.4.3 Exporting the Fragment Project

To export the fragment project as a plug-in: :

1. In the Package Explorer, ensure that the appropriate project is active. In this example, the project is called `WSDLSOAPInterceptors`.

2. From the **File** menu, select **Export**.
3. In the Export dialog box, under **Select an export destination**, expand **Plug-in Development** and select **Deployable Plug-ins and fragments**, then click **Next**.
4. In the **Destination** tab, select the *<MW_HOME>*/oepe_11gR1PS1/eclipse/plugins directory, then click **Finish**.

   The Export Plug-ins dialog shows the progress.

19.4.4 Importing the WLS Demo Certificates into the JVM's cacerts File

If you are using a demo certificate on your producer (WLS Default), import the WLS demo certificates into your JVM's cacerts file.

To import demo certificates:

1. Go to . *<domain home>*/bin/setDomainEnv.sh and open the command prompt.
2. Enter cd *$JAVA_HOME*/jre/lib/security/.
3. Import the certificate by entering keytool -importkeystore -v -destkeystore cacerts -srckeystore *<domain home>*/DemoTrust.jks.
   a. Enter the destination keystore password. The default cacerts password is changeit.
   a. Press the **[Enter]** key for the source password.

19.4.5 Adding System Properties to the eclipse.ini File

Adding the system properties for ISOAPInterceptor and IWSDLInterceptor to the eclipse.ini file will complete the configuration of these interceptors. So, every time Oracle Enterprise Pack for Eclipse will try to connect to a two-way SSL-enabled producer, these interceptors will supply the required certificate and private key.

To add the system properties:

1. Go to *<BEA_HOME>*/oepe_11gR1PS1/eclipse/, and open the eclipse.ini file.
2. Add the following system properties to the end of the file:
   - -Dcom.bea.wsrp.consumer.soap.ISOAPInterceptor=com.example.sample.EchoWsdlSoapInterceptor
   - -Dcom.bea.wsrp.consumer.wsdl.IWSDLInterceptor=com.example.sample.EchoWsdlSoapInterceptor
3. Save the file.

Part IV

Production

In the production phase of the portal life cycle, your portal is live. In this phase, you can perform some management functions, such as adding users. In a federated portal, you can add and remove remote portlets, and perform most of the tasks described in Part III, Staging. In the production phase, most of your work is done using the WebLogic Portal Administration Console.

For a detailed description of the production phase of the portal life cycle, see the *Oracle Fusion Middleware Overview for Oracle WebLogic Portal*.

Part IV contains the following chapter:

- Chapter 20, "Managing Federated Portals"

20

Managing Federated Portals

This chapter discusses operations you typically perform to a federated portal that is in production. This chapter includes the following topics:

- Section 20.1, "Modifying the Consumer Security Configuration"
- Section 20.2, "Modifying Producer Registration Properties"
- Section 20.3, "Updating the Producer URL"

20.1 Modifying the Consumer Security Configuration

Through the Service Administration panel of the WebLogic Portal Administration Console, you can modify the following consumer security settings. These settings are configured in the file `WEB-INF/wsrp-consumer-security-config.xml` associated with the consumer web application.

You can perform the following modifications:

- Section 20.1.1, "Changing the Web Application"
- Section 20.1.2, "Modifying Global Credentials"
- Section 20.1.3, "Modifying Producer Credentials"

20.1.1 Changing the Web Application

This section lets you change the consumer web application for the security configuration you want to modify. To change the web application:

1. In the Administration Console, select **Configuration Settings** > **Service Administration**.
2. In the Resource Tree, select **WSRP** > **Consumer Security**.
3. To change the web application, click **Change Web Application**. The Change Web Application dialog appears.
4. To search for a consumer web application, enter the full or partial name of the application to find in the Search for Webapps field, and click **Search**. Any web applications that are currently deployed to the server that match the search criteria are displayed in the dialog. The search is case sensitive.
5. Select the web application you want to change to, and click **Save**.

20.1.2 Modifying Global Credentials

You can edit the user name and password for the security credential that is used for all producers associated with this consumer. This change modifies the security credential that is managed by the server.

1. In the Administration Console, select **Configuration Settings** > **Service Administration**.
2. In the Resource Tree, select **WSRP** > **Consumer Security**.
3. Click **Edit** in the Global Credentials section. The Edit Credentials for All Producers dialog appears.
4. Enter the new user name and password.
5. Decide whether to check the Is Consumer Credential checkbox, as explained below, and click **Save**.

The Is Consumer Credential checkbox determines how the admin user is logged into the producer when destroyPortlets is called.

- Unchecked (default behavior in WebLogic Portal 8.1 and later versions)
  - The admin user's user name and password are sent to the producer through basic authentication (unsecure).
  - The user name and password must match the admin user on the producer.
- Checked
  - The admin user's credentials are sent to the producer securely via SAML or User Name Token (UNT).
  - The user name and password must match the admin user on the consumer.
  - Will interoperate with WebLogic Portal 9.2 (SAML or UNT) and 8.1 (SAML only) producers.

    The second (checked) method is the preferred configuration because it is secure.

20.1.3 Modifying Producer Credentials

You can edit the user name and password credentials associated with a specific producer.

1. In the Administration Console, select **Configuration Settings** > **Service Administration**.
2. In the Resource Tree, select **WSRP** > **Consumer Security**.
3. Click the producer handle for the producer whose credentials you want to change.
4. In the dialog, enter the new user name and password.
5. Decide whether to check the Is Consumer Credential checkbox, as explained below, and click **Save**.

The Is Consumer Credential checkbox determines how the admin user is logged into the producer when destroyPortlets is called.

- Unchecked (default behavior in WebLogic Portal 8.1 and later versions)
  - The admin user's user name and password are sent to the producer through basic authentication (unsecure).

  - The user name and password must match the admin user on the producer.
- Checked
  - The admin user's credentials are sent to the producer securely via SAML or User Name Token (UNT).
  - The user name and password must match the admin user on the consumer.
  - Will interoperate with WebLogic Portal 9.2 (SAML or UNT) and 8.1 (SAML only) producers.

    The second (checked) method is the preferred configuration because it is secure.

20.2 Modifying Producer Registration Properties

Using the WebLogic Portal Administration Console, you can modify the registration properties for a producer that has already been registered with a consumer. When the consumer re-registers the producer, some portlets that were previously in use might not be available or some additional portlets might be available to the consumer. Registration properties are discussed in Chapter 11, "Consumer Entitlement."

To modify a producer's registration properties, do the following:

1. In the WebLogic Portal Administration Console, select **Portal** > **Portal Management**.
2. In the Portal Resources Library tree, select **Remote Producers**, and then select the producer whose properties you want to modify.
3. In the Summary tab, select **Registration Details**.
4. In the Modify Producer Registration dialog, edit the values you want to change, and click **Modify Registration**.

   Tip: The Modify Registration dialog will be automatically filled in if you selected the Store Registration Properties option when you registered the producer. For more information on this option, see Section 14.19, "Storing Registration Properties".

Figure 20–1 Modify Producer Registration Dialog

20.3 Updating the Producer URL

From the Adminstration Console, you can change a producer's URL.

1. In the Administration Console, go to the Producer Summary page.
2. Click Producer Properties.
3. In the Update Producer Url dialog, enter a new URL.
4. Select Store Registration Properties to store the registration property set on the consumer. See Section 14.19, "Storing Registration Properties." If this option is not selected, registration properties are stored with the producer.
5. Select Re-register Producer to re-register the producer with the new URL value.