BEA WebLogic Portal®

Client-Side Developer's Guide

Version 10.2
February 2008

Contents

4. Configuring JavaScript Libraries in a Portal Web Project

5. Portlet Publishing

6. Client-Side Development Best Practices

7. The WebLogic Portal REST API

CHAPTER 1

What's in this Guide?

In a world of increasingly interactive and responsive web applications, WebLogic Portals need to be designed and updated to keep pace with users' expectations. Browser-based development technologies such as Ajax, JavaScript, and JSON, web-based APIs for accessing data, architectural patterns such as REST, and client-side development toolkits such as Dojo, enable web developers to create web applications that approach the responsiveness of desktop applications.

This guide is for WLP developers who want to integrate these client-side technologies into their portals. This chapter introduces the client-side portal development topics and technologies that are discussed in this guide.

- The Disc Framework
- Toolkit Integration
- Portlet Publishing
- The REST API
- The Dynamic Visitor Tools Sample Application

The Disc Framework

Disc (Dynamic Interface SCripting) is a client-side JavaScript framework that handles the asynchronous updating of portlets, portlet events, and the retrieval of portal context objects.

To get started using Disc, see Chapter 3, "The WLP Disc Framework."

Toolkit Integration

Client-side developers using WLP can use their favorite toolkits for DHTML and JavaScript browser programming. Dojo, for instance, works well for portlet development and for extending the WLP rendering framework.

To get started using Dojo for WLP client-side development, see Chapter 4, "Configuring JavaScript Libraries in a Portal Web Project."

Portlet Publishing

Traditionally, portlets have been confined to use within a portal application. This traditional model required an application server running a portal container to surface portlets. Portlet Publishing makes portlets available to any web application over HTTP and allows portlets to be surfaced in any web page. With Portlet Publishing, for example, you can render portlets within a Struts or Spring application, or any other non-portal web page.

For detailed information on publishing your portlets, see Chapter 5, "Portlet Publishing.".

The REST API

WebLogic Portal provides a set of web-based, REST-style APIs for retrieving, modifying, creating and updating portal data dynamically from the client.

For detailed information on the REST API, see Chapter 7, "The WebLogic Portal REST API."

The Dynamic Visitor Tools Sample Application

To get a feel for the possibilities of client-side portal development, we recommend that you run the Dynamic Visitor Tools (DVT) sample application. This sample application shows what is possible using the Disc and REST APIs to create a rich, interactive web application.

For detailed information on running the DVT sample, see the *WLP Samples Guide*.

Note: **The DVT Sample is not supported for a production environment.** The DVT sample is intended to illustrate how the WLP client-side development technologies such as the REST and Disc APIs and the external Dojo toolkit, can be used to enhance a portal's interactivity.

CHAPTER 2

Client-Side Development Technologies

This chapter introduces technologies and techniques that enable you to create rich and interactive WebLogic Portal applications. Rich and interactive portal web applications use a variety of technologies such as Ajax, JavaScript, JSON, and patterns such as REST. These technologies and patterns allow developers to create increasingly responsive and highly interactive web applications.

This chapter includes these topics:

Brief History of Client-Side Technology and the Portal

Since version 9.2, WebLogic Portal has incorporated Ajax (Asynchronous JavaScript and XML) technology to enable the asynchronous rendering of individual portlets. In version 10.0, the

asynchronous desktop feature was added, enabling portal administrators to enable asynchronous rendering for an entire desktop, simply by setting a property on the portal desktop.

Table 2-1 summarizes the history of asynchronous capability in WebLogic Portal.

Table 2-1 History of Asynchronous Portals

WebLogic Portal Version	Support for Asynchrony
9.2	Individual portlets can be rendered asynchronously by setting a portlet property. You can select either IFrames or Ajax as the integration technology. For details, see Asynchronous Portlet Content Rendering in the *Portlet Development Guide*.
10.0	In addition to the 9.2 features, entire portal desktops could be rendered asynchronously by setting the Asynchronous Mode of a desktop to "enabled." Desktop asynchrony uses Ajax technology. For details, see Asynchronous Desktop Rendering in the *Portal Development Guide*.
10.2	Users can write their own asynchronous browser code using Ajax, JavaScript, the WLP Disc framework, and related technologies, as explained in this guide.

JavaScript

JavaScript is used to enable client-side interactivity. JavaScript is a compact, loosely typed language that is commonly used by web developers. JavaScript has become widely adopted in recent years largely because new frameworks and best practices have proliferated. The emergence of JavaScript toolkits, such as Dojo, make client-side development easier while hiding many of the problems surrounding cross-browser compatibility.

Ajax

Ajax refers to a collection of standards-based and open source browser technologies. While a classic web application refreshes the entire web page with each response from the server, an Ajax-enabled web application allows small amounts of data and UI markup to be returned from the server and rendered in the browser without refreshing the entire page. To the user, an Ajax-enabled web application responds more smoothly and quickly than a traditional web application.

At the heart of Ajax is the XMLHttpRequest object. Often referred to as XHR, XMLHttpRequest is an HTTP request object that makes asynchronous (typically), incremental requests to the server. The JavaScript API for XMLHttpRequest is a key component of Ajax-style web applications. This object allows the browser to respond to user events without reloading the entire page. For instance, a web application that allows users to explore folders in a content repository can be designed so that only the explorer part of the interface refreshes each time the user opens a folder.

As explained in "Portal-Aware XMLHttpRequest" on page 2-3, the standard XMLHttpRequest object is not always the best solution in a portal environment.

Disc JavaScript Framework

Disc is a WLP-specific, public JavaScript API for portlet development and interaction with the WLP rendering framework. Disc helps you write Ajax-enabled portlets and enable richly interactive web application features for your portal. See Chapter 3, "The WLP Disc Framework."

Portal-Aware XMLHttpRequest

When standard XHR objects are used in a portal environment, certain rendering, interportlet communication, and other problems can occur. XHR calls are scoped to an entire portal, while portal-aware XHR calls are scoped to individual portlets. A portal-aware XHR call retrieves some portion of a portlet's content and of any other portlets that are affected.

The Disc API class, bea.wlp.disc.io.XMLHttpRequest, is an extension of the standard XMLHttpRequest class. WLP's XMLHttpRequest allows you to make asynchronous, incremental calls to the portal server, and supports portal features such as interportlet communication and WSRP. For more information, see Chapter 3, "The WLP Disc Framework."

Web-Based and REST Style Services

Representational State Transfer, or REST, is an architectural style for interacting with resources at a given URL. A REST architecture is typically used to provide a simple HTTP interface to resources published on the web. Although a simplistic analogy, you can think of REST services as an easy-to-use equivalent to traditional web services.

The primary advantage of using REST style services with web applications, such as WebLogic Portal, is that REST allows the web application to access web-addressed data and UI directly from the browser interface. The REST model provides a loose coupling between the data and the UI. REST services are easy to invoke using an HTTP request from a browser.

WebLogic Portal provides a REST framework that exposes WLP data and methods as REST services. These services unlock data from a portal web application and allow the data to be shared between applications. See Chapter 7, "The WebLogic Portal REST API" for details.

Tip: For a good general introduction to REST, refer to the Wikipedia article Representational State Transfer.

JSON

JavaScript Object Notation (JSON) is a data exchange format based on JavaScript. JSON is commonly used to pass objects between web clients and servers over HTTP. JSON objects are easy to read and to manipulate using JavaScript. For details on JSON, refer to json.org.

Ajax Toolkits

You can use available JavaScript toolkits and frameworks with Disc. For example, Dojo is a toolkit for DHTML and JavaScript browser programming. Dojo provides low-level IO (XMLHttpRequest), an event framework, JavaScript extensions, and a rich set of interface widgets such as buttons, lists, forms, and trees. Dojo works well for use in WLP portlets.

Dojo is discussed in Chapter 4, "Configuring JavaScript Libraries in a Portal Web Project." The Dojo website provides additional documentation: www.dojotoolkit.org.

CHAPTER 3

The WLP Disc Framework

This chapter discusses the Dynamic Interface SCripting (Disc) framework. Sometimes used in combination with the WLP REST API, Disc provides a client-side JavaScript API to assist with developing rich, interactive portlets and portal applications. Disc helps you write Ajax-enabled portlets that enable rich, interactive portal web application features.

This chapter includes these topics:

What is Disc?

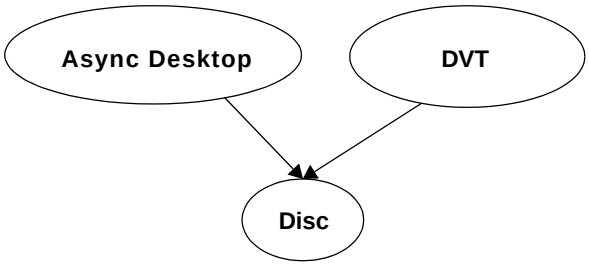
Disc provides a client-side, JavaScript, object-oriented programming framework for handling events, making asynchronous portlet updates, and for accessing portal context objects. Like their Java-based WLP API counterparts, Disc context objects encapsulate descriptive portal information, such as portlet positions within placeholders, titles, labels, and so on. The information returned in context objects can be used as parameters to REST commands to modify

the portal. For instance, the WLP feature called placeable movement (drag and drop portlets) is implemented using Disc and REST features (see Enabling Placeable Movement in the *Portal Development Guide*).

Tip: Reference documentation for the Disc JavaScript API is provided on e-docs.

The Asynchronous Desktop Mode and DVT features both use Disc, as illustrated in Figure 3-1. If you enable either Asynchronous Desktop Mode or DVT for a portal desktop, Disc is implicitly enabled.

Figure 3-1 Features That Depend on Disc



The DVT flag enables the placeable movement (drag and drop portlets) feature for the portal desktop. Asynchronous desktop rendering and placeable movement are both discussed in the *Portal Development Guide*.

Enabling Disc

To use Disc in client-side portal code, you have to enable it first. You can enable Disc in WorkSpace Studio or in the WebLogic Portal Administration Console. You can also use Disc outside the context of a portal.

This section includes these topics:

- What Enabling Disc Means
- Enabling Disc in WorkSpace Studio
- Enabling Disc in the Administration Console
- Using Disc Outside of a Portal

What Enabling Disc Means

When Disc is enabled, you have access to the following WLP client-side development features:

- **Portal-Aware XHR (XMLHttpRequest) objects** – If you are familiar with Ajax, then you know that XMLHttpRequest (XHR) objects are used to make asynchronous requests to the server. Because the browser's native XHR object is designed to operate in the context of an entire web page, it does not work well in a portal environment where the page is composed of many separate portlets. Disc provides a *portal-aware* extension of XHR that makes requests that allow portlet markup to be updated asynchronously. The term portal-aware means that when the Ajax request is sent to the server from a portlet, the request can fire events, if required, and render its own view for further processing on the client. See "Using Portal-Aware XMLHttpRequest to Update Portlets" on page 3-6.

  The portal-aware XHR object, bea.wlp.disc.io.XMLHttpRequest is described in detail in the Disc API reference documentation on e-docs.

- **Context Objects** – Disc lets you work with context objects that encapsulate information about a portal, such as the portlet placeholder information, titles, labels, and so on. Context objects are generally read-only, and let you obtain information that you can use to create client-side interactivity and to pass to REST commands to update the portal.

  The set of context objects in the bea.wlp.disc.context module are described in detail in the Disc API reference documentation on e-docs.

- **XIE** – The XHR Interaction Engine (XIE) provides the infrastructure for handling asynchronous requests, interportlet communication, and event handling. For instance, when asynchronous rendering is enabled for a desktop, XIE enables interportlet communication (IPC) to work correctly. When a request is made from a portlet, markup is automatically returned for that portlet and any portlet that is affected by the request through IPC.

  The set of XIE objects in the bea.wlp.disc.xie module are described in detail in the Disc API reference documentation on e-docs.

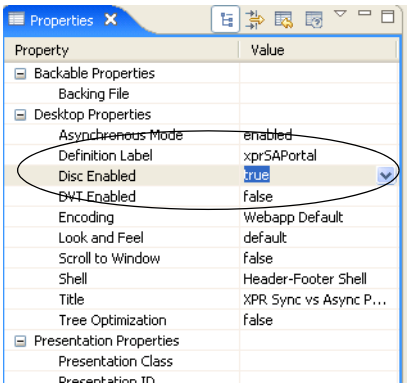
- **Event Handling** – Disc provides event handling mechanisms that work as callback functions and that can carry payloads. Events are tied to the "lifecycle" of asynchronous portlet requests. This means you can use events to trap information before a request is sent or before a response is processed by the client.

  For detailed information on the Event object in the bea.wlp.disc.event module, see Disc API reference documentation on e-docs.

Enabling Disc in WorkSpace Studio

To enable Disc for a portal desktop in WorkSpace Studio, set Disc Enabled to true in the desktop Properties view, as shown in Figure 3-2.

Figure 3-2 Enabling Disc in WorkSpace Studio



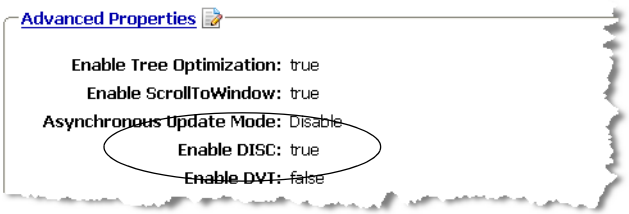
Note: Enabling either Asynchronous Mode or DVT implicitly enables Disc for a desktop.

Enabling Disc in the Administration Console

To enable Disc in the Administration Console:

1. In the Portal Resources tree, select **Portals** and navigate to a desktop.
2. Select the Details tab.
3. Click Advanced Properties (Figure 3-3) and use the dialog to set Enable Disc to true.

Figure 3-3 Enable Disc



Note: If you enable Asynchronous Mode or DVT, Disc is implicitly enabled.

Using Disc Outside of a Portal

The WLP Portlet Publishing feature uses Disc outside of a portal. Portlet Publishing lets you render portlets in any HTML page. For more information on Portlet Publishing, see Chapter 5, "Portlet Publishing."

Using Portal-Aware XMLHttpRequest to Retrieve Data from a Non-Portal Source

The standard XMLHttpRequest object, familiar to Ajax developers, can be used to update discrete amounts of data within a portlet. The Disc API class, bea.wlp.disc.io.XMLHttpRequest, is an extension of the standard XMLHttpRequest class. WLP's XMLHttpRequest allows you to make asynchronous, incremental calls to the portal server, and supports portal features such as interportlet communication and WSRP. However, you can also use the portal-aware XHR in the same way you use a standard XHR. Like a standard XHR request, a portal-aware XHR request can be used from any JSP portlet page to retrieve any arbitrary data, which can then be inserted into the portlet.

Use Case

For an example use case, you could write a portlet that includes an auto-complete search form. When a user types text in the search field, the portlet sends portal-aware XHR requests to the server to retrieve suggestions for the user. In this process, the portlet UI does not refresh itself, except for the suggestions shown in the text field.

Example

In Listing 3-1, a portal-aware XHR object is instantiated and used to asynchronously retrieve data from a non-portal source. This code is intended to be embedded in a JSP page.

Listing 3-1 Using Portal-Aware XHR to Retrieve Data from a Non-Portal Source

```
<script type="text/javascript">
    var dataUrl = 'data.json';
    var xmlhttp = new bea.wlp.disc.io.XMLHttpRequest();
    xmlhttp.onreadystatechange = function() {
        if ((xmlhttp.readyState == 4) && (xmlhttp.status == 200)) {
            var data = eval('(' + xmlhttp.responseText + ')');
```

```
            var table = document.getElementById('data');
            for (var i = 0; i < data.length; i++) {
                // insert rows into "data" table by referencing properties of
                // data[i] objects.
            }
        }
    }
    xmlhttp.open('GET', dataUrl, true);
    xmlhttp.send();
</script>
```

Using Portal-Aware XMLHttpRequest to Update Portlets

Portal-aware XMLHttpRequest allows client-side JavaScript code to interact with the portal framework on the server on behalf of a portlet. When a portal-aware XHR call is made to the server, the request is processed by the portal, which decides what to do with the request. The portal may decide to invoke a given portlet and provide it with the appropriate context, such as user properties, portlet preferences, and request and response objects. Because the portal manages dependencies between portlets, interportlet communication is possible with portal-aware XHR.

Use Cases

Two use cases for using portal-aware XHR are:

- A portlet wants to implement a multi-step wizard process, replacing the current form with a new form after each form submission and without completely re-rendering the portlet.
- When a user interacting with a portlet selects a check box for a product ID, the portlet fires an event on the server side. Other portlets handle this event, fetch product related information from a backend data source, and render that information.

As a developer, you have the option of using the WLP-specific XMLHttpRequest object (bea.wlp.disc.io.XMLHttpRequest) to update portal-specific content. When you make a request through this object, features such as IPC and WSRP are supported. For detailed information on the portal-aware XHR class, see the Disc API reference documentation on e-docs.

To enable portal-aware XHR in WorkSpace Studio, follow the instructions in the section "Enabling Disc in WorkSpace Studio" on page 3-4.

Example

Listing 3-2 is basically the same as Listing 3-1. The difference is that Listing 3-2 refers to a portlet as the source of the data. This example code is expected to be embedded directly in a JSP page.

Listing 3-2 Using Portal-Aware XHR to Update Portlets

```
<%@ taglib prefix="render"
uri="http://www.bea.com/servers/portal/tags/netuix/render" %>
<render:jspContentUrl contentUri="/path/to/portlet/data.jsp"
forcedAmpForm="false" var="dataUrl"/>

<script type="text/javascript">
    var dataUrl = '${dataUrl}';
    var xmlhttp = new bea.wlp.disc.io.XMLHttpRequest();
    xmlhttp.onreadystatechange = function() {
        if ((xmlhttp.readyState == 4) && (xmlhttp.status == 200)) {
            var data = eval('(' + xmlhttp.responseText + ')');
            var table = document.getElementById('data');
            for (var i = 0; i < data.length; i++) {
                // insert rows into "data" table by referencing properties
                // of data[i] objects.
            }
        }
    }
    xmlhttp.open('GET', dataUrl, true);
    xmlhttp.send();
</script>
```

Using Context Objects

Disc lets you work with portal context objects on the client, using JavaScript. Context objects encapsulate certain information about portal components. Context objects contain information about the following portal components:

- portlets

- pages
- books
- layouts
- themes

Disc content classes are loosely representative of the server-side PresentationContext classes and serve a similar function in client-side programming. The set of context objects in the bea.wlp.disc.context module are described in detail in the Disc API reference documentation on e-docs.

Note: Only context objects for visible components are available; context objects for non-visible components such as books peer to the current book or hidden portlets are not available through the Disc APIs.

Using Context Objects

Disc context objects are generally available only after the entire HTML page has loaded. As a best practice, do not attempt to use context objects from inline script blocks. Instead, register an on-load handler that specifies interaction with context objects. For example, the code in Listing 3-3, when placed inline in a portlet JSP, results in an error:

Listing 3-3 Wrong Way to Access a Context Object

```
//portlet will be null
var portlet = bea.wlp.disc.context.Portlet.findByLabel("myPortletLabel");
var title = portlet.getTitle(); // error
```

However, if you add the same code in a Dojo (for example) on-load function, the code works as expected:

Listing 3-4 Recommended Way to Access a Context Object

```
dojo.addOnLoad(function() {
    var portlet = bea.wlp.disc.context.Portlet.findByLabel("myPortletLabel");
    var title = portlet.getTitle();
});
```

Listing 3-5 illustrates a simple debugging example. The best practice is to place this code in a render dependencies file. See "Creating a Render Dependencies File" on page 4-4 for more information.

Listing 3-5 Simple Debugging Example

```
dojo.addOnLoad(function() {
     var portlets = bea.wlp.disc.context.Portlet.getAll();
     for (var i = 0; i < portlets.length; i++) {
         bea.wlp.disc.Console.debug(portlets[i]);
     }
});
```

Listing 3-6 presents another example that uses context objects.

Resources from render dependencies files are always included for a portlet regardless of mode or state (except for minimized, which is a special case). If a portlet has a general function that tries to inject content into a portlet that always runs when loaded from a render dependencies file (attached to an onload handler for example), the function should probably only try to do the content injection when the portlet is in "normal view" mode. Listing 3-6 is intended to be loaded from a render dependencies file.

Listing 3-6 Using Disc to Determine Portlet State

```
function createDataTables(label) {
    var portlet = bea.wlp.disc.context.Portlet.findByLabel(label);
    if (portlet.getWindowMode() == "view") {
        // retrieve data and create corresponding tables...
        if (portlet.getWindowState() == "maximized") {
            // create table with extended details
        }
        else {
            // create table with common details
        }
```

```
    }
}
```

The XMLHttpRequest Interaction Engine

The Disc module bea.wlp.disc.xie defines public APIs for Disc's XMLHttpRequest Interaction Engine (XIE). XIE is the client-side foundation for Ajax-driven interactions with WebLogic Portal. XIE is also the platform on which various other Ajax-based, public Disc APIs are implemented, including:

- bea.wlp.disc.io.XMLHttpRequest and
- bea.wlp.disc.publishing.PortletSource

The XIE API is divided into two main areas:

- XIE Events – Located in bea.wlp.disc.xie.Events.*, these global events are fired when Ajax requests are made to WLP and when subsequent responses are processed. These events provide public access to the underlying client-server interaction lifecycle.
- Async Request Overlay – Located in bea.wlp.disc.xie.AsyncRequestOverlay, this object provides control over the blocking overlay introduced on a page by some XIE-driven interactions.

For more information on this classes described in this section, see the Disc API documentation on e-docs.

Event Handling

The bea.wlp.disc.xie.Events object contains the set of public, global events that are fired during XIE's WLP interaction lifecycle. The interaction lifecycle involves setting up and executing an Ajax request to the WLP server, receiving a response, and subsequently processing the response. WLP Ajax responses are encoded in an internal JSON format (the form of which is reserved and subject to change), and XIE manages the evaluation and handling of the body of these responses. The suite of public XIE events provides public access to key moments of interest during this lifecycle; listening code can use these event hooks to respond to or even influence the outcome of the interaction. For more information on this class, see the Disc API documentation on e-docs.

1. **OnPrepareUpdate** – Fired when an interaction has been initiated, but before the request is made. This event can be cancelled; cancellation terminates the interaction without making the underlying request.
2. **OnHandleUpdate** – Fired immediately after XIE receives the response, but before processing has begun.
3. **OnRedirectUpdate** – Fired if the server is forcing the client to perform a redirect; processing will complete as soon as possible and then a client-side redirect is performed.
4. Then, for each piece of response markup returned by the server, the following events are fired:
    a. **OnPrepareMarkup** – Fired before converting a response markup fragment into a DOM subtree; this event fires once for each markup fragment that was returned in the response.
    b. **OnPrepareContent** – Fired immediately after converting a markup fragment into a DOM subtree, but before doing additional processing (such as rewriting anchor hrefs and form actions); this event fires once for each markup fragment that was returned in the response.
    c. **OnInjectContent** – Fired immediately after injecting the fully processed (for example, rewritten) DOM subtree into the document, but before executing any scripts associated with that content; this event fires once for each markup fragment that was returned in the response.
5. **OnCompleteUpdate** – Fired after XIE has completed all response processing for the interaction, including the execution of any scripts defined by the markup updated during the interaction
6. **OnError** – Fired any time an error occurs during interaction processing; processing may or may not continue once an OnError event has fired.

Each event delivers a payload object to its listeners when the event is fired. The type and capabilities of each payload object differ from event to event. See the documentation for each individual event for more information about specific event payloads in the Disc API documentation on e-docs.

Logging

One way to implement logging is to use the Firebug logging model. For information on this model, see http://getfirebug.com/logging.html. Both Firebug and Firebug Lite provide a global Console object that you can use to monitor logging output.

Disc provides a Console object (bea.wlp.disc.Console) that replicates the API found in the Firebug Console object. You can use this object anytime Disc is available. For information on the Disc Console object, refer to bea.wlp.disc.Console in the Disc API documentation on e-docs.

Example

Listing 3-7 shows a simple example of how to set up a listener and send messages through the Console object. All messages logged to the Console are passed to each listener on the object. The listener function can be passed the following arguments when the debug call is made:

op	“debug”
args	[“label=”, “myPortlet”]

This configuration will output the following line of HTML text into a div named `myConsoleOutput`, which needs to have been previously defined somewhere on the page:

```
DEBUG: label=myPortlet
```

Listing 3-7 Simple Example: Setting Up a Console Listener

```
bea.wlp.disc.Console.addListener(function(op, args) {
    var output = document.getElementById("myConsoleOutput");
    output.appendChild(document.createElement("br"));
    output.appendChild(document.createTextNode(op.toUpperCase() + ": " +
    args.join(",")));
});
// Sometime later...  Assume myPortlet.getLabel() returns "myPortlet"
bea.wlp.disc.Console.debug("label=", myPortlet.getLabel());
```

CHAPTER 4

Configuring JavaScript Libraries in a Portal Web Project

A wide variety of JavaScript libraries and toolkits exist today, many of which can be used to build WebLogic Portal projects. For example, the Dojo Toolkit works well for client-side WLP development. This section explains how to set up a WLP project that uses the Dojo Toolkit. You can follow the same basic steps to integrate the JavaScript libraries of your choice into a project.

This chapter includes these topics:

Creating a Simple Portal Project with Dojo

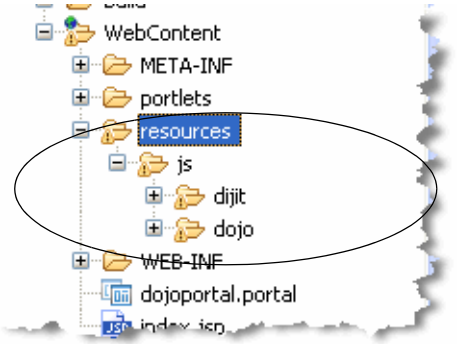
Dojo is a toolkit for DHTML and JavaScript browser programming. Dojo provides low-level I/O (XMLHttpRequest), an event framework, JavaScript extensions, and a rich set of interface widgets such as buttons, lists, forms, and trees. Dojo has a good set of features to use for developing portals and portal components.

This section explains how to set up a WebLogic Portal project that includes the Dojo Toolkit.

1. Locate and download the Dojo Toolkit. The toolkit is available at www.dojotoolkit.org.
2. Unzip the download file.
3. In WorkSpace Studio, create a Portal Web Project, a Portal EAR Project, a server, and a domain. See the WebLogic Portal Tutorial for details on these tasks.

4. In the WebContent folder of the Portal Web Project, create a folder called `resources`. In that folder, create a subfolder called `js`.
5. Import the Dojo Toolkit folder into the `resources/js` folder. To do this, right-click the `resources/js` folder and select **Import**. In the wizard, select **General > Filesystem**. Import both the `dojo` and `dijit` folders from the Dojo download directory. See Figure 4-1.

Figure 4-1 Adding the Dojo Toolkit to a Portal Project



With the Dojo Toolkit added to your portal project, you can now incorporate Dojo and other JavaScript code into your portlets, as explained in the following sections.

The Hello World JSP Portlet with Dojo

This section describes a simple JSP portlet that displays a message in a pop up. The sample demonstrates how to import the Dojo libraries and use a bit of JavaScript in a portlet.

1. Create a `portlets` folder under the WebContent folder of the web project.
2. In the portlets folder, create a new JSP portlet descriptor file, called `dojotest.portlet`. To do this, right-click the folder and select **New > Portlet**. Use the portlet wizard to create a JSP portlet. The wizard automatically creates both the portlet file and an empty JSP file called, by default, `dojotest.jsp`.
3. Create a render dependencies file. This file will contain a reference to the Dojo Toolkit file, dojo.js, a CSS file reference, and JavaScript function definitions used by the portlet. See "Creating a Render Dependencies File" on page 4-4 for details.

4. Configure the portlet to refer to the render dependencies file. To do this, open the portlet in the Portlet Editor view. Set the **LAF Dependencies Path** in the portlet Properties editor to point to the dependencies file you created.
5. Add the code in Listing 4-1 to the JSP file. The sample code performs these tasks:

   Imports a button dijit Button element or widget. This is a common pattern for importing widgets. This code ensures that the Button's JavaScript has been loaded before a Button instance is declared.

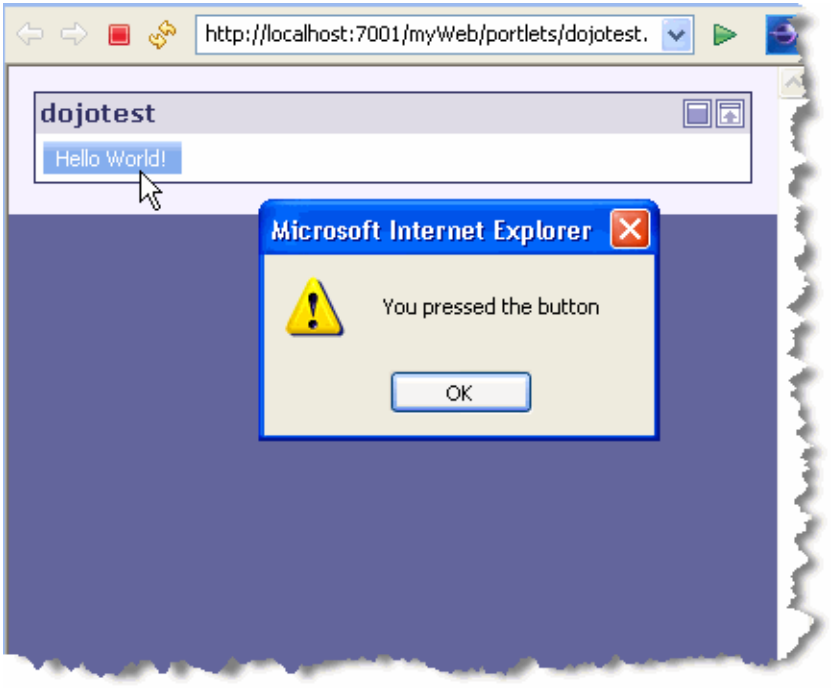
   Instantiates a button widget and configures an event handler for the button's click event. When the button is pressed, the event handler shows a JavaScript alert containing a greeting and the portlet label.

Listing 4-1 Hello World with Dojo

```
<script type="text/javascript">
    // Load the Dojo Button widget
    dojo.require("dijit.form.Button");
</script>

<button dojoType="dijit.form.Button" id="helloButton">
        Hello World!
    <script type="dojo/method" event="onClick">
        alert('You pressed the button');
    </script>
</button>
```

6. Add the portlet to a portal and run it on the server. When you click the button in the portlet, the popup displays the text defined in the callback.

Figure 4-2 Hello World JSP Portlet with Dojo



This section explained how to use a bit of Dojo code in a JSP portlet. In Chapter 6, "Client-Side Development Best Practices," we discuss specific problems related to using JavaScript code in portlets and suggest best practices to avoid the problems.

Creating a Render Dependencies File

This section explains the purpose and importance of render dependency files and how to create a render dependencies file.

The Importance of Render Dependencies

A render dependencies file is XML that defines page-level events and resources such as external JavaScript and CSS that are needed by a portlet. In a non-portal web page, these are often included inside the <head> tags of an HTML page; however, portlets should avoid using this approach, as it can lead to a variety of problems in a portal or mashup environment including creating pages that are not HTML/XHTML compliant and causing unexpected behavior when

WebLogic Portal's Desktop Ajax feature is enabled. The render dependencies mechanism addresses these problems by allowing portlets to declare resource references and to wire into page-level events including load and unload. Blocks of JavaScript can even be included in a render dependencies file, wrapped inside of the usual <script> element.

Creating the Sample Render Dependencies File

The simplest way to create a render dependencies file is to right-click a folder in your web application and select **New > Other > WebLogic Portal > Markup Files > Render Dependencies**. The resulting file looks like Listing 4-2.

Listing 4-2 Empty Render Dependencies File

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

Next, add a <render-dependencies> XML tag to the file. Include in the tag the path to the Dojo toolkit, as shown in Listing 4-3. This listing includes certain elements that Dojo requires for this example: the Tundra theme and the djConfig object, which is created when the page loads and configures Dojo.

The <path-element> must be a relative reference from the render dependencies file to the directory that contains the script sources referenced by the <script> tags.

Listing 4-3 Render Dependencies File Includes Dojo Toolkit

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

```
    <render-dependencies>
       <html>
          <links>
              <search-path>
                  <path-element>../resources/js</path-element>
              </search-path>
               <link href="dijit/themes/tundra/tundra.css" type="text/css"
               rel="stylesheet"/>
          </links>
          <scripts>
              <search-path>
                  <path-element>../resources/js</path-element>
              </search-path>
              <script type="text/javascript">
                  var djConfig = {parseOnLoad:true, isDebug:true};
              </script>
              <script src="dojo/dojo.js" type="text/javascript"/>
          </scripts>
       </html>
    </render-dependencies>
</window>
```

CHAPTER 5

Portlet Publishing

This chapter discusses Portlet Publishing, a feature that lets you surface portlets remotely over HTTP.

This chapter includes these topics:

- What is Portlet Publishing?
- Portlet Publishing URL Forms
- Consuming Published Portlets
- Advanced Topics
- Portlet Publishing vs. WSRP
- Limitations

What is Portlet Publishing?

Portlet Publishing refers to a family of WebLogic Portal features that make portlets available over HTTP. Once published through the WLP Portlet Publishing service, a portlet can be reused in a variety of heterogeneous host environments. For example, you can embed published portlets in an <iframe> tag or include the portlet into a non-portal web page using JavaScript.

This section includes these topics:

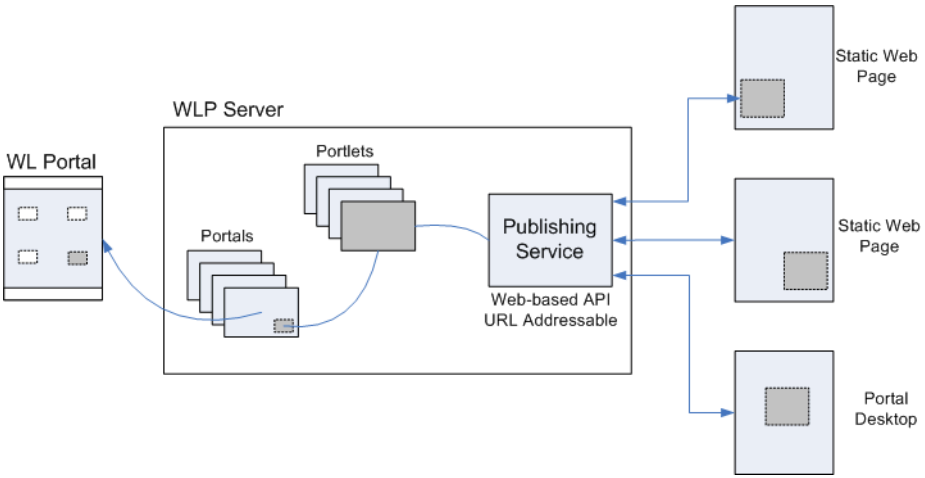
- What is the Portlet Publishing Service?
- Asking the Portlet Publishing Service for Portlets

- Consuming a Published Portlet

What is the Portlet Publishing Service?

Portlets are published through the WLP Portlet Publishing service. The publishing service is a REST-style service that runs in the WLP server. As shown in Figure 5-1, the Portlet Publishing service serves individual portlet instances to remote consumers over HTTP. A remote consumer simply uses a URL to request that the Portlet Publishing service return a portlet. The service then returns the portlet markup to the consumer.

Figure 5-1 Portlet Publishing Service



Asking the Portlet Publishing Service for Portlets

You can ask the Portlet Publishing service to return a *library instance* of a portlet or a *desktop instance*. The type of portlet instance you want determines the structure of the URL sent to the service.

You can think of a library portlet instance as a non-customizable instance of a portlet. When you use the Administration Console to add a library portlet to a page in a portal desktop, the added portlet becomes a desktop instance, which makes it possible for each user of the portlet to have their own customized version of it. You can see library and desktop portlets listed in the Portal Resources Tree of the WebLogic Portal Administration Console.

You might decide to ask for a library instance of a portlet if you do not expect users to customize it or if it was not designed to be customizable. For example, a simple mortgage calculator portlet

might not require customization. You might choose to consume a desktop instance of a portlet, on the other hand, if you do expect users to customize it. A stock quote portlet might allow users to select their favorite stocks to display and then remember their individual choices.

As noted previously, the type of portlet you want the publishing service to return determines the type of URL that you send to the service. The two Portlet Publishing URL patterns are explained in "Portlet Publishing URL Forms" on page 5-4.

Consuming a Published Portlet

As explained previously, you formulate a specific kind of URL to ask the WLP publishing service for a library instance or for a desktop instance of a portlet. The publishing service then returns the portlet markup to the requesting consumer. The consumer can render the returned portlet markup in three ways:

- Render the portlet in an HTML inline frame tag (an <iframe> tag). This is by far the simplest and safest technique. The <iframe> tag provides a loose coupling between the portlet and the rest of your rendered web page. Portlets rendered in <iframe> tags are effectively isolated from one another, reducing possible security risks and adverse side-effects.
- Use a JavaScript API to integrate the portlet into the DOM. WLP provides a Disc API called PortletSource that lets you retrieve a portlet and render it directly into an HTML tag. This technique provides greater flexibility and control to the web developer. For instance, integrated portlets can interact and affect each other. Greater care is required on the part of the client-side developer to ensure that integrated portlets work and interoperate properly and securely.
- Use the <portalFacet> JSP tag. To surface a portlet in a JSP page, you can insert a portalFacet tag into a JSP page. For example:

  ```
  <%@taglib uri="http://www.bea.com/servers/portal/tags/netuix/render"
  prefix="render"%>
  ...
  <render:portalFacet label="_myportlet" path="/myPortlet.portlet" />
  ```

  The web application includes a file called `myPortlet.portlet`. The `label` tag attribute must be unique to all <portalFacet> tags on the page because it identifies the portlet instance. At runtime, the <portalFacet> tag renders the specified portlet on the page. You may include multiple portlets on a single page by inserting multiple <portalFacet> tags with unique label attributes.

See also "Consuming Published Portlets" on page 5-5.

Portlet Publishing URL Forms

This section describes the two Portlet Publishing URL forms. The first form retrieves a library instance of a portlet. The second form retrieves a desktop instance. The WLP Portlet Publishing service processes these commands and returns the appropriate portlet to the consumer. Library and desktop portlet instances are explained in more detail in "Asking the Portlet Publishing Service for Portlets" on page 5-2.

Library Instance URL Form

To return a library instance of a portlet, use this URL form:

`http://<domain_name>/<webapp_name>/<path_to_portlet>/<portlet_name>.portlet`

For example:

```
http://foo.com/myWebapp/myPortlet.portlet
```

Note that the part of the URL that includes the web application name, the path to the portlet, and the portlet name is called the publishing context. The publishing context can be used as a parameter to certain Disc JavaScript API calls. See "DOM Integration" on page 5-8 for more information. See also "Finding Portlet Publishing Context" on page 5-10.

Because a library instance is, by definition, not directly associated with a portal desktop, it does not include certain look and feel and other information associated with a desktop instance. The look and feel of a library instance consists of the default WLP look and feel and cannot be changed.

Desktop Instance URL Form

The desktop instance URL form is somewhat more complicated than the form used for library instances. The desktop instance URL is composed of these required parts and parameters:

- Domain name and web application name:

  For example: `http://foo.com/myWebapp`

- WLP REST API namespace identifier:

  `bea/wlp/api`

- Portlet Publishing service command:

  `portlet/publish`

- Context parameter. The context parameter value is the path to a streaming portal desktop. For information on obtaining the context path, see "Finding Portlet Publishing Context" on page 5-10.

Tip: The difference between streaming and file-based portals is explained in more detail in the *WebLogic Portal Development Guide*.

For example:

```
/myWebApp/appmanager/myPortal/myDesktop
```

Note that `appmanager` is a WLP servlet name that is part of every streaming desktop path.

- Portlet parameter. This parameter specifies the instance label of the library portlet. You can determine the instance label from either WorkSpace Studio or from the WLP Administration Console. For example, a portlet with instance identifier `myPortlet_1` would appear as:

```
portlet=myPortlet_1
```

Putting it all together, here is an example URL to retrieve a desktop instance portlet from the publishing service:

```
http://foo.com/myWebApp/bea/wlp/api/portlet/publish?
context=/myWebApp/appmanager/myPortal/myDesktop&portlet=myPortletInstanceLabel
```

There is one more optional parameter you can use in the desktop instance URL form. The `decoration` parameter is explained in the section "Advanced Topics" on page 5-11.

Consuming Published Portlets

This section discusses two techniques for consuming published portlets: inline frame (<iframe> tag) integration and DOM integration. This section includes these topics:

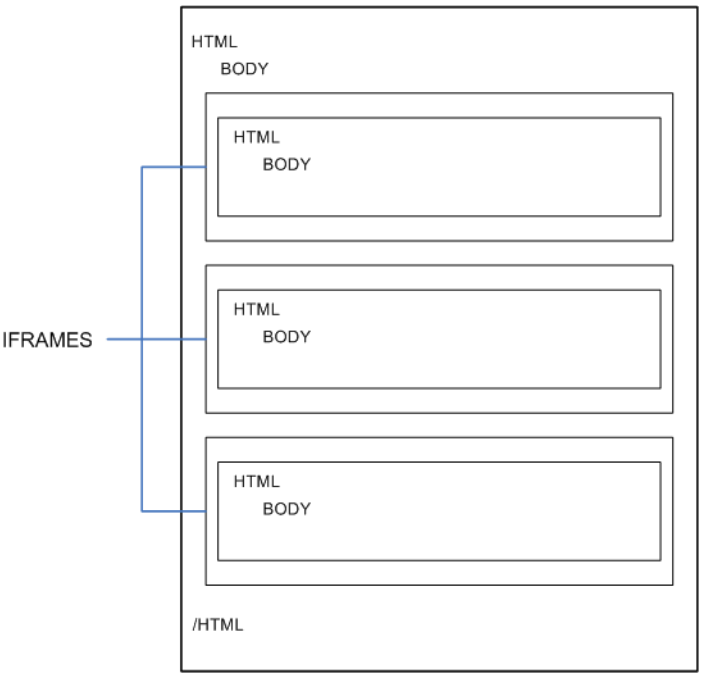
Inline Frame Integration

The HTML <iframe> tag creates an inline frame that contains another HTML document. You can use an inline frame to include a WLP portlet in an HTML page. The value of the tag's src

parameter can be a Portlet Publishing URL of either the library instance or desktop instance form as explained in "Portlet Publishing URL Forms" on page 5-4.

Figure 5-2 illustrates that inline frames are rendered as separate HTML documents within the parent document. The loose coupling between the parent page and the inline frame pages makes inline frames relatively secure. Inline frames are a good choice if you don't trust a portlet to interact securely with the rest of the page. Only limited interaction between the inline frame portlets and the rest of the page is possible. Note that some portlets will not work properly if they are embedded using the DOM integration technique, described in "DOM Integration" on page 5-8. In these cases, inline frame integration is the best option.

Figure 5-2 Inline Frames Provide Separation Between Portlets and the Rest of the Page



Example Code

Listing 5-1 shows an inline frame tag that includes a library instance portlet. Listing 5-2 shows an example that embeds a desktop instance of a portlet. The value of the <iframe> src parameter is the Portlet Publishing URL of the appropriate form for a library instance and a desktop instance. See also "Portlet Publishing URL Forms" on page 5-4.

Listing 5-1 Simple Inline Frame that Embeds a Library Instance Portlet

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <body>
    <iframe src ="http://foo.com/myWebapp/myPortlet.portlet"> </iframe>
  </body>
</html>
```

Listing 5-2 Simple Inline Frame that Embeds a Desktop Instance Portlet

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <body>
    <iframe src ="http://foo.com/myWebApp/bea/wlp/api/portlet/publish/?
       context=myWebApp/appmanager/myPortal/myDesktop&
       portlet=myPortletInstanceLabel"> </iframe>
  </body>
</html>
```

You can use inline frames to embed either static or dynamic portlets, as explained in the following sections.

Embedding Static Portlets

Inline frames offer a relatively safe and easy way to integrate legacy portlets into newly developed, static applications. Inline frames are ideal for portlets that use basic HTML, do not make extensive use of JavaScript, post navigation commands back to the server, and contain logic that is captured in Struts or Page Flow actions.

Embedding Dynamic Portlets

Dynamic portlets make use of rich user interface technology and asynchronous browser-server communication. More and more sites are being built using mashups of content from one or more providers that are often not known and possibly not trusted by one another. Inline frames provide

a convenient and relatively secure sandboxing mechanism in environments where rich widgets and portlets must be integrated together without mutual trust.

DOM Integration

WLP provides a JavaScript API called Disc that lets you integrate portlets directly into the DOM of an HTML page.

Tip: For a comprehensive introduction to Disc, see Chapter 3, "The WLP Disc Framework."

DOM integration provides a level of integration and capability that exceeds the inherent limitations of inline frame integration. DOM integration allows a portlet to be treated as if it were a native part of the HTML page's DOM tree. This added capability increases both security risks and integration complexity. Portlets integrated using this approach have complete access to data and UI that exist in other parts of the page. In addition, the developer must take care to prevent collisions in the JavaScript global namespace. See also "Avoiding Namespace Collisions" on page 6-7.

Basic JavaScript Coding Steps

The basic JavaScript coding steps for integrating a remote portlet into the DOM of an HTML page follow a consistent pattern. This section lists and explains each step and then provides a complete example.

1. Include the Disc Portlet Publishing JavaScript module in your page, as shown in the following fragment:

   ```
   <script type="text/javascript"
      src="/<portal_web_app>/framework/scripts/bundles/disc-publishing.js">
   </script>
   ```

   This JavaScript module is contained in a WLP Shared J2EE Library, and is always addressed by the web application-scoped path shown in the fragment. You only need to include this module once per page, even if you are embedding portlets from several different web applications.

2. Build a variable to hold the publishing context of the portlet. The publishing context is either the path to a `.portal` file or a desktop of a streaming portal. See also "Finding Portlet Publishing Context" on page 5-10.

   For example, the following fragment specifies the context for a file-based portlet, where `<portal_web_app>` and `<portal_name>` are the names of the WLP web application and

the portal that contain the portlet. The context for a file-based portlet is the physical path from the web application root to the `.portal` file.

```
var publishingContext = "/<portal_web_app>/<portal_name>.portal";
```

To embed a desktop instance of a portlet from a streaming portal, you construct the context variable as follows:

```
var publishingContext =
    "/<portal_web_app>/appmanager/<portal_name>/<desktop_name>
```

See also "Portlet Publishing URL Forms" on page 5-4.

3. Use the Disc API to retrieve a PortletSource instance for the publishing context. Pass the publishing context variable you constructed in Step 2 as the parameter in this call. For example:

   ```
   var source = bea.wlp.disc.publishing.PortletSource.get(publishingContext);
   ```

4. Call the PortletSource object's `render()` function, passing it a JavaScript object with two properties. The first property, `portlet`, specifies the instance label of the portlet you want to embed. The second property, `to`, specifies the value of an id attribute for an HTML div tag in which to embed the portlet. For example:

   ```
   source.render({ portlet: "<portlet_instance_label>", to: "<div_id_value>" });
   ```

   The render() method also accepts an array of JavaScript objects. See "Advanced Topics" on page 5-11 for details.

5. Place the portlet in a `<div>` element on the HTML page. For example:

   ```
   <div id="div_id_value"></div>
   ```

Putting it all together, Listing 5-3 shows a complete example where a portlet library instance is embedded into the DOM of an HTML page. In this example, most of the code is wrapped in a JavaScript function called handleOnLoad() that is called when the page loads.

Tip: Note that the text "Loading..." appears in the tag. This text provides a visual cue to the user. "Loading..." appears momentarily in the page and is replaced by the portlet after the portlet markup is received.

Listing 5-3 Example Code for Portlet DOM Integration

```
<html>
    <head>
        <title>My Portlet</title>
```

```
        <script type="text/javascript"
           src="/myWebApp/framework/scripts/bundles/disc-publishing.js">
        </script>
        <script type="text/javascript">
            function handleOnLoad() {
                var publishingContext = "/myWebApp/portals/main.portal";
                var source =
                   bea.wlp.disc.publishing.PortletSource.get(publishingContext);
                source.render({ portlet: "myPortlet_1", to: "myPortlet" });
            }
        </script>
    </head>
    <body onload="handleOnLoad();">
            <p>Basic tests for rendering portlets on a host page.</p>
            <div style="border: 1px solid black;">
                <div id="myPortlet">Loading...</div>
            </div>
    </body>
</html>
```

Finding Portlet Publishing Context

This section explains how to find the value of the context parameter for library and desktop instance portlets.

Finding the Publishing Context for a Library Instance Portlet

To find the publishing context for library instance portlet, in WorkSpace Studio, right-click the `.portal` file and select **Run As** > **Run On Server**. The path shows up in the browser. For example, if the full URL to the portal is:

```
http://foo.com/myWeb/portal-1.portal
```

then the publishing context is:

```
/myWeb/portal-1.portal
```

Finding the Publishing Context for a Desktop Instance Portlet

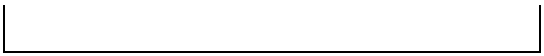
The easiest way to obtain the context path for a desktop instance portlet, is to look it up in the WLP Administration Console. In the Portal Resources tree, navigate to the desktop and select it.

The path shows up on the Details tab in the URL to Access Desktop field. A complete path looks like this, where the context part is everything from *<webapp_name>* on.

`http://<domain_name>/<webapp_name>/appmanager/<portal_name>/<desktop_name>`

Here is an example desktop URL with the publishing context highlighted:

`http://foo.com/myWebApp/appmanager/myPortal/myDesktop`



Publishing Context

Advanced Topics

This section discusses several advanced topics of interest to developers.

- Using the Decoration Parameter
- Integrating Multiple Portlets into the DOM
- Integrating Portlets from Multiple Publishing Contexts

Using the Decoration Parameter

The desktop instance URL form, discussed in "Desktop Instance URL Form" on page 5-4, includes an optional `decoration` parameter. The `decoration` parameter lets you specify how the look and feel of the portlet will be rendered. The possible values for the decoration portlet are:

- `light` – (default) Only the content of the portlet is rendered. Light decoration does not include a border or titlebar if they were part of the original desktop portlet instance. This style portlet tends to blend in more with the surrounding web page elements.
- `full` – This option renders the portlet almost exactly it appears in its original desktop context. If defined in the original portlet, the border and title bar (with mode and state buttons) will be rendered. Essentially, the portlet is rendered with the same look and feel as the original portlet appeared in its host desktop.

For example:

```
http://foo.com/myWebApp/bea/wlp/api/portlet/publish/?
context=myWebApp/appmanager/myPortal/myDesktop&
portlet=myPortletInstanceLabel&decoration=full
```

For more information, see "Desktop Instance URL Form" on page 5-4.

Integrating Multiple Portlets into the DOM

"DOM Integration" on page 5-8 discussed how to use the Disc API to integrate a single portlet into the DOM. If you want to integrate multiple portlets onto a page, you can pass the render() function an array of {portlet: , to: } objects. In this case, Disc only makes one request to the server to retrieve all of the portlets.

For example, Listing 5-4 illustrates how to integrate multiple desktop instances of portlets into the DOM by passing an array of objects to the Disc render() function.

Listing 5-4 Embedding Multiple Portlets

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Portlet Publishing Example</title>
         <script type="text/javascript"
            src="/publishingPortal/framework/scripts/bundles/disc-publishing.js">
        </script>
        <script type="text/javascript">

            function handleOnLoad() {
                var publishingContext =
              "/publishingPortal/appmanager/mainPortal/mainDesktop";
                var source =
               bea.wlp.disc.publishing.PortletSource.get(publishingContext);

                source.render([{ portlet: "myPortlet_1", to: "portlet1Div" },
                        { portlet: "myPortlet_2", to: "portlet2Div" },
                        { portlet: "myPortlet_3", to: "portlet3Div" },
                        { portlet: "myPortlet_4", to: "portlet4Div" }]);
            }
        </script>
    </head>
    <body onload="handleOnLoad();">

         <div style="border: 1px solid black;" id="portlet1Div">Loading...</div>
         </br>
         <div style="border: 1px solid black;" id="portlet2Div">Loading...</div>
         </br>
         <div style="border: 1px solid black;" id="portlet3Div">Loading...</div>
         </br>
         <div style="border: 1px solid black;" id="portlet4Div">Loading...</div>

    </body>
</html>
```

Integrating Portlets from Multiple Publishing Contexts

You can integrate portlets from more than one Portlet Publishing context into page. If you are doing DOM integration, you must construct a separate PortletSource API call for each publishing context. If you are using inline frame integration, just use the appropriate Portlet Publishing URL form as the value of the <iframe> src tag.

Note that each PortletSource.render() call produces one interaction with the server. If you group several portlets into one render() call, the function will not return until all of the portlets are processed on the server. If one of the portlets requires excessive processing time on the server, users might experience a delay in the rendering of the entire group of portlets. In this case, consider placing the slower portlet in a separate render() call. This causes the portlet to be rendered in a separate, asynchronous call to the server.

Portlet Publishing vs. WSRP

You can think of Portlet Publishing and WSRP (Web Services for Remote Portlets) as complementary technologies. Both technologies allow you to consume portlets deployed in remote portals.

In the case of WSRP, portlets from one or more portals, called *producers*, are aggregated into a common, unified portal, called the *consumer*. The consumer communicates with producers over a SOAP-based protocol. Because WSRP is an OASIS standard, the producers can use entirely different vendor technologies. Also, because the consumer and producer communicate through standard web services, UDDI registries and service busses can be used to locate portlets and route traffic. Traditional WSRP portlets are closely tied to the producer portals in which they are deployed.

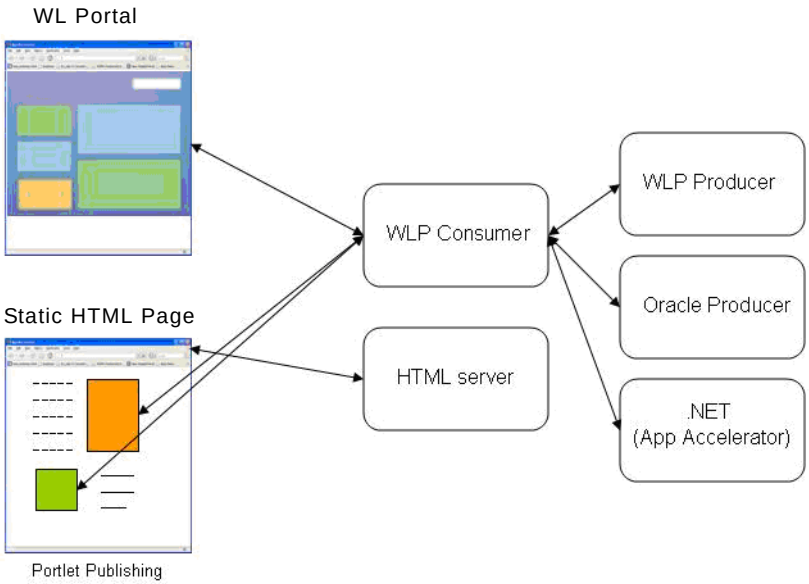
Portlet Publishing, on the other hand, aggregates content (typically portlets) on the *browser*. Client-side developers make inline frame (iframe) or XMLHttpRequest calls to the server to retrieve content. Portlet Publishing is typically used to expose portlets in a non-portal applications, although you can also use Portlet Publishing techniques in a portal.

Tip: The WSRP approach to creating federated portals is described in the *Federated Portals Guide*.

Figure 5-3 illustrates the fundamental differences between Portlet Publishing and WSRP. Portlet Publishing obtains content directly from a WLP application. WSRP aggregates content in a

consumer portal from one or more producers. Because WSRP is a standard, producers can be from different vendors.

Figure 5-3 Portlet Publishing and WSRP



Limitations

- Portlet Publishing does not support server-side interportlet communication. Portlets that implement IPC will not send or receive IPC events in the Portlet Publishing context. For more information on IPC, see the *Portlet Development Guide*.
- In some cases, the capabilities of the original host desktop will not be available to published portlets. For example, the host desktop's backing context will not be available.
- In some cases, it is possible that some parts of the portal control tree are not available to published portlets.
- Desktops with URL compression enabled should not be published. URL compression is discussed in the *Portal Development Guide*.
- Disc context objects will not be available on the published page. See Chapter 3, "The WLP Disc Framework." for more information.

CHAPTER 6

Client-Side Development Best Practices

This chapter discusses tips and best practices for developing client-side portal code. The chapter includes these topics:

- Namespacing
- Avoiding Namespace Collisions

Namespacing

If you are writing browser-based JavaScript code in a WLP environment, namespacing conflicts can easily arise if you are not careful. This section illustrates the kind of JavaScript namespace conflict that can occur when you place multiple portlets on a page.

- A Simple Dynamic Table Portlet
- Namespace Collisions

A Simple Dynamic Table Portlet

The following example uses JavaScript to add rows to an HTML table dynamically. The Dojo Toolkit is used to provide the primary UI element (a button) and the event handling mechanism. As you will see, you will encounter problems if you intend to use code like this in a portlet/portal context.

Note: The following code is designed according to the best, recommended practice of using a render dependencies file for including toolkit references, CSS references, and JavaScript functions in a portlet. For an introduction to this technique and step by step instructions

on creating and referencing a render dependencies file, see Chapter 4, "Configuring JavaScript Libraries in a Portal Web Project."

The render dependencies file is shown in Listing 6-1. A render dependencies file is XML that defines page-level events and resources such as external JavaScript and CSS that are needed by a portlet. The overall pattern of the file follows the pattern discussed in Chapter 4, "Configuring JavaScript Libraries in a Portal Web Project."

The JSP file is shown in Listing 6-2. It defines a JavaScript function that adds a row to an HTML table and includes the Dojo button that fires the onClick event that calls the addRowsToTable() function.

Listing 6-1 Render Dependencies File

```
<?xml version="1.0" encoding="UTF-8"?>
<window xmlns="http://www.bea.com/servers/portal/framework/laf/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.bea.com/servers/portal/framework/laf/1.0.0
      laf-window-1_0_0.xsd">
    <render-dependencies>
        <html>
            <links>
                <search-path>
                    <path-element>../resources/js</path-element>
                </search-path>
                <link href="dijit/themes/tundra/tundra.css" type="text/css"
                       rel="stylesheet"/>
            </links>
            <scripts>
                <search-path>
                    <path-element>../resources/js</path-element>
                </search-path>
               <script src="dojo/dojo.js" type="text/javascript"/>
                <script type="text/javascript">
                       var djConfig = {parseOnLoad:true, isDebug:true};
                </script>
            </scripts>
        </html>
    </render-dependencies>
</window>
```

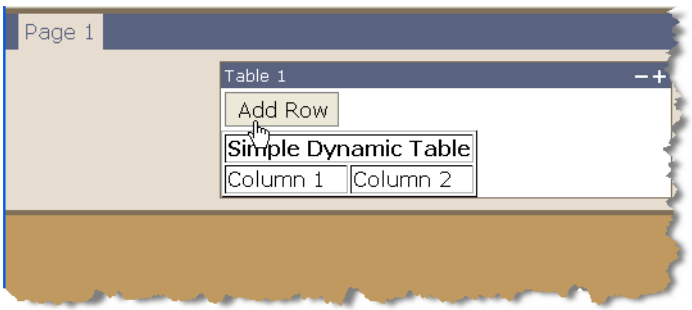
Listing 6-2 JSP File

```
<script type="text/javascript">
    // Load the Dojo Button widget
    dojo.require("dijit.form.Button");
</script>

<script type="text/javascript" >
    function addRowToTable() {
    var tbl = document.getElementById("table_1");
    var lastRow = tbl.rows.length;
    var row = tbl.insertRow(lastRow);

    //-- Update left cell
    var cellLeft = row.insertCell(0);
    var node1 = document.createTextNode(lastRow);
    cellLeft.appendChild(node1);

    //-- Update right cell
    var cellRight = row.insertCell(1);
    var node2 = document.createTextNode(lastRow);
    cellRight.appendChild(node2); }
    </script>

<button dojoType="dijit.form.Button" id="addRowButton">Add Row <script
type="dojo/method" event="onClick">
        addRowToTable();
    </script></button>

<br>

<table border="1" id="table_1">
    <tr>
        <th colspan="3">Simple Dynamic Table</th>
    </tr>
    <tr>
        <td>Column 1</td>
        <td>Column 2</td>
```

```
    </tr>
</table>
```

Figure 6-1 shows the JSP portlet added to a portal page. Each time the Add Row button is clicked, a new row is added to the table, and the row number is displayed in the new table cells.

Figure 6-1 Simple Dynamic Table Portal

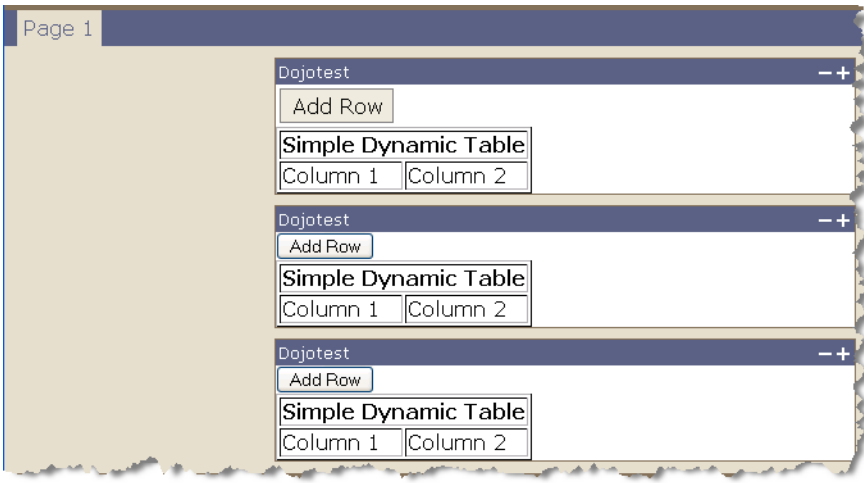


The addRowToTable() JavaScript function retrieves the table element using the table's ID. While the portlet works perfectly well in this scenario, what happens if you add a second identical portlet to the same portal page?

Namespace Collisions

Suppose you add the dynamic table portlet to the same page multiple times. For example, consider the following portal (Figure 6-2), which contains three dynamic table portlets.

Figure 6-2 Multiple Dynamic Table Portlets on a Page



The current configuration causes a JavaScript error to be thrown. After the event handler for the first button on the page is registered, subsequent attempts to register it fail. The error message is something like this:

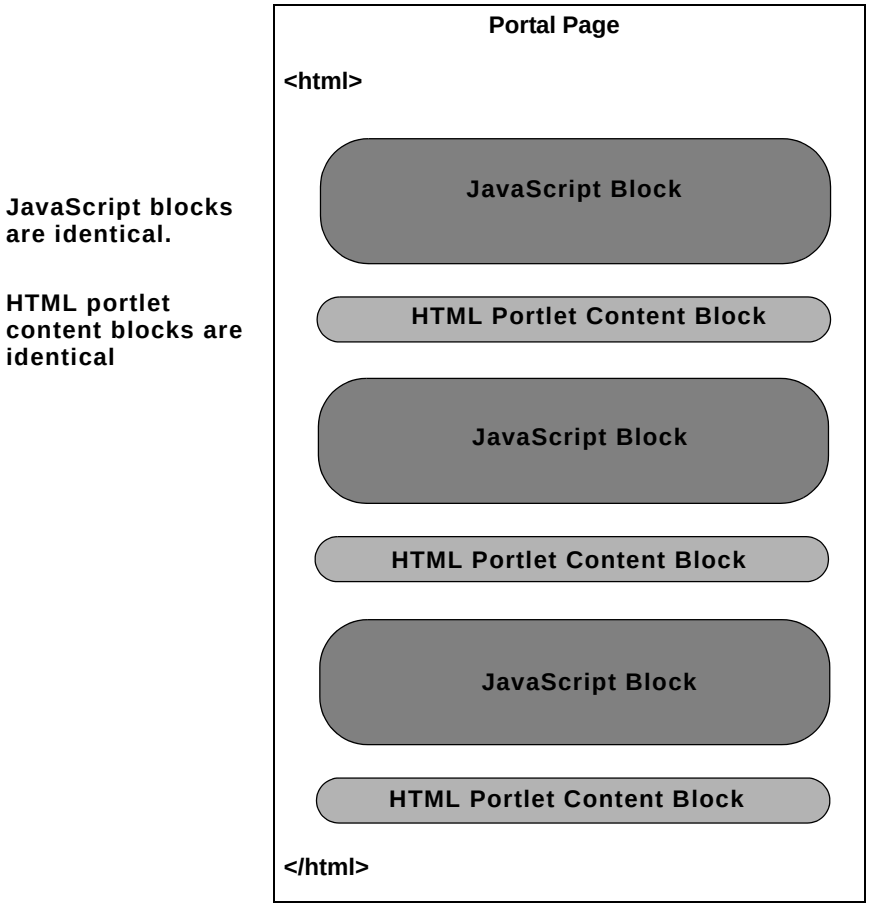
```
Exception... "'Error: Tried to register widget with id==helloButton but that
id is already registered' ...
```

As a result of this error, only the first button is registered, and it is the only button that works.

When you understand why multiple dynamic table portlets fail to work properly in a portal page, you can learn avoid many common browser-side programming problems encountered by portal developers.

The reason why the dynamic table fails to function properly when the portlet is duplicated is clear when you look at the HTML source code that is generated for the portal page. The portal framework on the server returns exactly the same HTML markup for each portlet. Both the JavaScript block and the HTML table code are duplicated verbatim three times in the same HTML page, as illustrated in Figure 6-3. The WebLogic Portal framework assigns the portlets themselves unique IDs; however, all of the JSP and HTML code is identical for each rendered portlet.

Figure 6-3 Source Code Repetition with Three Identical Portlets on a Page



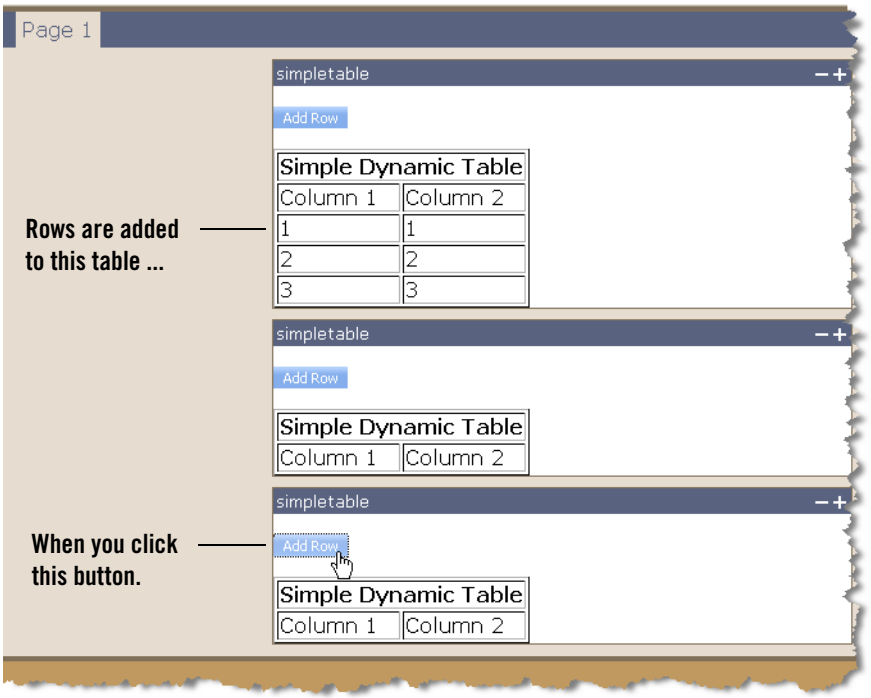
The Mysterious Table Row

It is possible to assign the dijit button in Listing 6-2 a unique name, such as by prefixing it with the portlet's unique ID (see techniques discussed in "Using Ad Hoc Namespacing" on page 6-8). Such a technique will avoid the error condition; however, assigning a unique name to the button ID is insufficient to solve the namespacing problem.

Note that when a JavaScript function block is included multiple times, each time a duplicate variable is declared, it is overwritten by subsequent variables with the same name. In this example, the variable addRowButton is declared three times (because the script block is included

three times in the page). Therefore, whichever button you click will only affect (add rows to) the first table, as illustrated in Figure 6-4.

Figure 6-4 Rows Mysteriously Added to the Wrong Table



Techniques for avoiding these namespace problems are discussed in the next section, "Avoiding Namespace Collisions" on page 6-7.

Avoiding Namespace Collisions

This section discusses techniques and best practices for avoiding the kinds of namespace collisions illustrated in "Namespace Collisions" on page 6-4. The techniques include:

- Using Ad Hoc Namespacing
- Using Rewrite Tokens
- Parameterizing Your JavaScript Functions

- Using the Disc APIs

Using Ad Hoc Namespacing

One way to avoid namespace collisions in client-side browser code is to use ad hoc namespacing. Ad hoc namespacing means giving a unique name to:

- Global functions
- Global variables
- DOM IDs (for example: <table id=”someUniqueId”>)
- DOM names (apply only to anchor and form tags)
- References to all of the above

For example, the JSP listed in “Namespace Collisions” on page 6-4 can be rewritten so that all of the *global* functions, variables, and IDs are unique and scoped to a specific portlet. One simple technique for achieving this in a portlet is to append the portlet’s instance label to the appropriate global JavaScript identifiers, as shown in Listing 6-3. Additions to this code sample are highlighted. As you can see, the example relies on obtaining the portlet instance label from the PortletPresentationContext object. This ID is unique for each portlet that appears in a portal. JSP expressions are then used to append the portlet label to appropriate global identifiers. In this case, we need to uniquely namespace the HTML table ID, the global function addRowToTable(), the Button widget ID, and the Button widget variable.

Tip: JavaScript variables within functions are locally scoped, and therefore do not need to be namespaced for the global context.

Listing 6-3 Dynamic Table JSP with Ad Hoc Namespacing

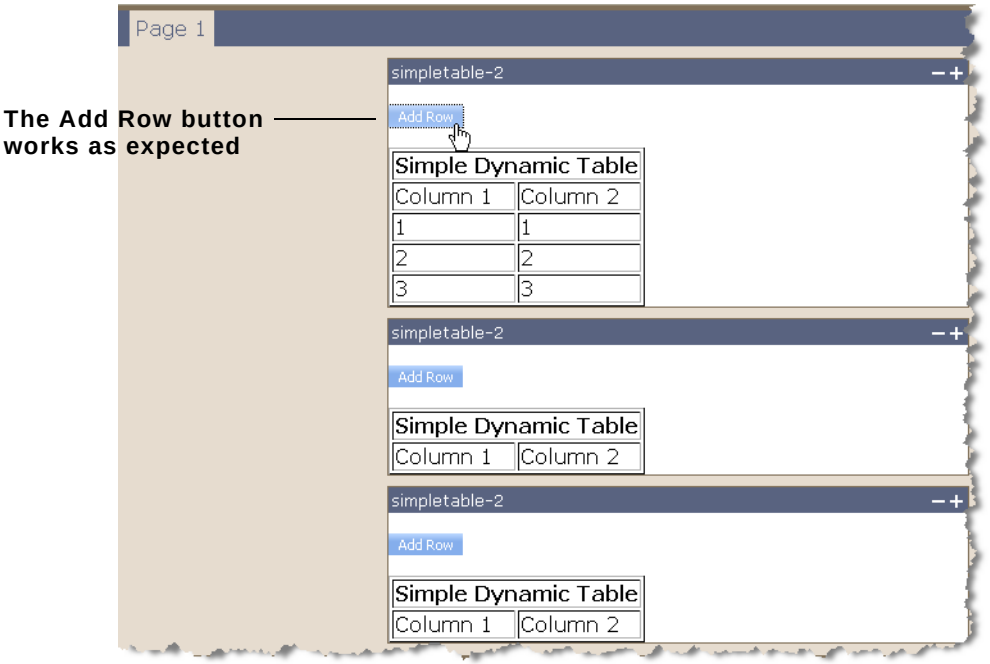
```
<%@ page
import="com.bea.netuix.servlets.controls.portlet.PortletPresentationContext"%>
<%
// Tip: you can also use the JSP tag <render:encodeName name="someName" .../>
// to accomplish the same task as this scriptlet (obtaining the portlet
// instance label.)
PortletPresentationContext portletCtx = (PortletPresentationContext)
PortletPresentationContext.getPortletPresentationContext(request);
```

```
String portletId = portletCtx.getInstanceLabel();
pageContext.setAttribute("prefix", portletCtx.getInstanceLabel());
%>

<script type="text/javascript">
    // Load the Dojo Button widget
    dojo.require("dijit.form.Button");
</script>

<script type="text/javascript" >
    function ${prefix}_addRowToTable() {
                var tbl = document.getElementById("${prefix}_table_1");
                var lastRow = tbl.rows.length;
                var row = tbl.insertRow(lastRow);

                //-- Update left cell
                var cellLeft = row.insertCell(0);
                var node1 = document.createTextNode(lastRow);
                cellLeft.appendChild(node1);

                //-- Update right cell
                var cellRight = row.insertCell(1);
                var node2 = document.createTextNode(lastRow);
                cellRight.appendChild(node2);
        }
 </script>

<button dojoType="dijit.form.Button" id="${prefix}_helloButton"> Add Row
      <script type="dojo/method" event="onClick">
          ${prefix}_addRowToTable();
       </script>
</button>

<br>

<table border="1" id="${prefix}_table_1">
        <tr>
                <th colspan="3">Simple Dynamic Table</th>
        </tr>
        <tr>
                <td>Column 1</td>
```

```
                <td>Column 2</td>
        </tr>

</table>
```

If you generate a portlet from this JSP and add the portlet multiple times to a portal, you will see that the portlets function independently and correctly, as shown in Figure 6-5.

Figure 6-5 Namespaced Portlet Code Functions Correctly



While ad hoc namespacing is an effective way to ensure that portlets with browser code function properly, other techniques and best practices are examined in the following sections.

Using Rewrite Tokens

A well known best practice among JavaScript developers is to externalize functions into .js files. An alternative approach to the ad hoc namespacing technique described in "Using Ad Hoc

Namespacing" on page 6-8 is to externalize your JavaScript functions and use the WLP render dependencies feature and the rewrite token to provide unique scoping of variables. For additional information on the rewrite token, see the section Portlet Dependencies in the *Portlet Development Guide*.

The first step is to place the JavaScript script blocks that require namespacing into an external .js file. You then reference this file in the render dependencies file.

Tip: The trick to including the `.js` file is to include it with the `<script content-uri ...>` tag rather than with the `<script src ...>` tag. Although the `src` tag is the most standard mechanism, if you use it, the rewrite tokens will not be expanded. For information on creating a dependency file and attaching a dependency file to a portlet, see Portlet Dependencies in the *Portlet Development Guide*.

Listing 6-4 shows a sample render dependencies file. The included `.js` file is highlighted in bold text.

Listing 6-4 Render Dependencies File

```
<?xml version="1.0" encoding="UTF-8"?>
<window xmlns="http://www.bea.com/servers/portal/framework/laf/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.bea.com/servers/portal/framework/laf/1.0.0
      laf-window-1_0_0.xsd">
    <render-dependencies>
        <html>
            <links>
                <search-path>
                    <path-element>../resources/js</path-element>
                </search-path>
                <link href="dijit/themes/tundra/tundra.css" type="text/css"
                  rel="stylesheet"/>
            </links>
            <scripts>
                <search-path>
                    <path-element>../resources/js</path-element>
                </search-path>
                <script src="dojo/dojo.js" type="text/javascript"/>
                <script type="text/javascript">
```

```
                    var djConfig = {parseOnLoad:true, isDebug:true};
                </script>
                <script type=
                  "text/javascript" content-uri="dojotest.js"/>
            </scripts>
        </html>
    </render-dependencies>
</window>
```

The contents of the external .js file are shown in Listing 6-5. Note that `wlp_rewrite_` is appended to all of the function names and identifiers that require unique names within the page.

Listing 6-5 Contents of External dojotest.js File with Rewrite Tokens

```
function wlp_rewrite_addRowToTable() {
    var tbl = document.getElementById("wlp_rewrite_table_1");
    var lastRow = tbl.rows.length;
    var row = tbl.insertRow(lastRow);

    //-- Update left cell
    var cellLeft = row.insertCell(0);
    var node1 = document.createTextNode(lastRow);
    cellLeft.appendChild(node1);

    //-- Update right cell
    var cellRight = row.insertCell(1);
    var node2 = document.createTextNode(lastRow);
    cellRight.appendChild(node2);
     }
```

The portal framework takes care of replacing this token with the portlet's instance label, which is a unique identifier.

Listing 6-6 shows the JSP file with the JavaScript function removed. Note that the .js file does not have to be imported into the JSP file if it is included in a render dependencies file. Namespacing of the dijit button id and the table id with the unique portlet id is still required in the JSP.

Listing 6-6 JSP File

```
<%@ page
import="com.bea.netuix.servlets.controls.portlet.PortletPresentationContext"%>
<%
PortletPresentationContext portletCtx = (PortletPresentationContext)
PortletPresentationContext.getPortletPresentationContext(request);
String portletId = portletCtx.getInstanceLabel();
pageContext.setAttribute("prefix", portletCtx.getInstanceLabel());
%>

<script type="text/javascript">
    // Load the Dojo Button widget
    dojo.require("dijit.form.Button");
</script>

<button dojoType="dijit.form.Button" id="${prefix}_helloButton"> Add Row
      <script type="dojo/method" event="onClick">
          ${prefix}_addRowToTable();
       </script>
</button>

<br>

<table border="1" id="${prefix}_table_1">
        <tr>
                <th colspan="3">Simple Dynamic Table</th>
        </tr>
        <tr>
                <td>Column 1</td>
                <td>Column 2</td>
        </tr>

</table>
```

Parameterizing Your JavaScript Functions

This section discusses another version of the dynamic table example in which parameterized JavaScript functions are factored out into a `.js` file. This external file is shown in Listing 6-7. Note that both the addRow() and init() functions are parameterized. Note that it is a best practice to include external .js files in a portlet using a render dependencies file. See "Creating a Render Dependencies File" on page 4-4.

Listing 6-7 External JavaScript File (.js)

```
function addRow(id) {

        var table = document.getElementById(id+"_table");
        var numrows = table.getElementsByTagName("tr").length;
        var row = table.insertRow(numrows);

        //-- Add text to the row cells.
        var cellLeft = row.insertCell(0);
        var textLeft = document.createTextNode(numrows-1);
        cellLeft.appendChild(textLeft);

        var cellRight = row.insertCell(1);
        var textRight = document.createTextNode(numrows-1);
        cellRight.appendChild(textRight);

}
function initialize(id) {
        var addHandler =
                function() {
                        addRow(id);
                };

    var addRowButton = dijit.byId(id+"_addRowButton");
    dojo.connect(addRowButton, 'onClick', addHandler);
}
```

The JSP file from which the portlet is generated must be written so that the ID parameter is passed to the init() function. Note that the addOnLoad() function is called by passing in an anonymous function that returns the init() function. This technique is necessary, and allows the parameter passed to init() to persist after the addOnLoad() function returns.

Listing 6-8 Modified JSP File

```
<%@ page
import="com.bea.netuix.servlets.controls.portlet.PortletPresentationContext"%>
<%
PortletPresentationContext portletCtx = (PortletPresentationContext)
PortletPresentationContext.getPortletPresentationContext(request);
pageContext.setAttribute("prefix", portletCtx.getInstanceLabel());
%>

<script type="text/javascript">
    // Load the Dojo Button widget
    dojo.require("dijit.form.Button");
</script>

<script type="text/javascript">
    dojo.addOnLoad( function() {
         initialize("${prefix}");
    });
</script>


<button dojoType="dijit.form.Button" id="${prefix}_addRowButton"> Add Row
</button>

<br>

<table border="1" id="${prefix}_table">
        <tr>
                <th colspan="3">Simple Dynamic Table</th>
        </tr>
        <tr>
                <td>Column 1</td>
                <td>Column 2</td>
        </tr>
```

```
</table>
```

In summary, the original dynamic portlet described in "A Simple Dynamic Table Portlet" on page 6-1 has been redesigned to function in a portal/portlet environment. First, global variables and functions were addressed with ad hoc namespacing to prevent namespace collisions. Second, JavaScript code was externalized into a `.js` file. To accomplish this, we parameterized functions and used techniques of closure and anonymous functions. Of course, externalizing JavaScript code is always a best practice, as it allows for code reuse and reduces clutter in JSP or HTML files.

Using the Disc APIs

You can use the Disc APIs to retrieve unique portlet labels that can be used to scope JavaScript variables. The basic technique is similar to the one described in "Using Ad Hoc Namespacing" on page 6-8. With Disc, you can get the portlet label from the portlet's context object. For more information, see "Using REST and Disc" on page 7-10.

CHAPTER 7

The WebLogic Portal REST API

WebLogic Portal provides a set of web-based, REST-style APIs for retrieving, modifying, and updating portal data dynamically from the client. This chapter discusses the WLP REST API and provides use cases and examples.

Tip: See the WLP REST API Reference online for detailed information on each of the WLP REST commands.

What is REST?

REST (REpresentational State Transfer) is an approach for building services that make specific resources available at a URL. A REST service has well defined operations for manipulating the resource. Typically, these operations include reading, writing, editing, and removing. In the case of WLP, a resource might be a portlet, page, or book. For example, the following WLP REST command retrieves a list of portlets in the specified web application:

```
http://localhost:7001/myWebApp/bea/wlp/api/portlet/list?webapp=myWebApp
```

Tip: For a good general introduction to REST, see the Wikipedia article "Representational State Transfer" at http://en.wikipedia.org/wiki/Representational_State_Transfer.

What is the WLP REST API?

The WLP REST API focuses on a number of straightforward use cases, such as “retrieving a list of portlets” and “adding a page to a book.” Following the general pattern for REST APIs, the WLP REST API provides commands that read, create, update, and delete portal artifacts. For example, WLP REST commands exist for listing portlets, adding portlets, updating portlets, and deleting portlets. The results of REST commands are persisted through the WLP customization framework. The REST commands discussed in this chapter are not intended for portlet rendering.

Note: Some of the REST API commands only work with streaming portals. Generally, a streaming portal is one that has been configured using the Portal Administration Console. For more information, see File Based and Streaming Portals.

WLP REST API Reference Documentation

You can find an online reference that describes each of the WLP REST commands on e-docs at: http://edocs.bea.com/wlp/docs102/apidocREST/index.html.

REST API Use Cases

The WLP REST commands currently provide access to WLP data to support visitor customization features of WLP. The commands allow browser-based tools to interact with and modify a user’s desktop. For example, REST commands are used extensively by the placeable movement feature as well as in the Dynamic Visitor Tool sample application (see Chapter 8, “The Dynamic Visitor Tools Sample”). The commands provide services such as adding, removing, and moving portlets on a page. In general, the WLP REST commands provide a more natural and easy to use alternative to a web services approach.

Note: Some REST commands only return what is not currently visible on the desktop. This supports the use case where users would not want or expect to see a list of, for instance, portlets to add if they were already added to the desktop.

REST Command Format

The format of a WLP REST command is:

`<protocol>://<host>:<port>/<webapp>/bea/wlp/api/<type>/<action>/<label>?<params>`

Table 7-1 describes the parts of a WLP REST command:

Table 7-1 REST Command Format

Command Part	Description
<protocol>	The transport protocol, typically http.
<host>	The IP host name, such as localhost.
<port>	The port number, such as 7001
<webapp>	The name of the web application hosting the services, such as myWebApp. Note that all REST commands also require a webapp parameter. The webapp parameter specifies the specific web application for the command to operate on. The web application specified in the parameter can be different from the web application you post to, as long as it is deployed in the same EAR file.
bea/wlp/api	Standard namespace path used for all REST commands
<type>	The type of portal artifact the command operates on, such as desktop, book, page, portlet, lookandfeel, shell, menu, layout, them.
<action>	The specific action to take, such as list, delete, add, and get.
<label>	The unique label of the object.
command-specific parameters	A list of command-specific URL parameters. See the REST API reference documentation on e-docs for a complete list for each command. Commonly used parameters are described in "Commonly Used REST Command Parameters" on page 7-3.

Commonly Used REST Command Parameters

Each REST command takes a list of parameters that are described in the REST API reference documentation on e-docs. This section describes three commonly used parameters in greater detail.

The webapp Parameter

The webapp parameter is always required. It specifies the name of the web application on which you are calling the REST command. If you have more than one web application deployed in a given EAR, this parameter lets you easily switch between them. The web application specified

with the parameter can be different web application you are posting to (that is, the application specified in the base URL).

For example, in the following command, myWebApp is the application that receives the command, and yourWebApp is the one for which the portlet list is retrieved. Both web applications must be deployed in the same EAR.

```
http://localhost:7001/myWebApp/bea/wlp/api/portlet/list?webapp=yourWebApp
```

The format Parameter

The format parameter can be either `xml` or `json`. The default is `xml`. JSON is an object notation for JavaScript, and is generally easier to work with in a browser environment with JavaScript. See "JSON" on page 2-4 for more information For example:

```
http://localhost:7001/myWebApp/bea/wlp/api/portlet/list?webapp=yourWebApp&
format=json
```

The scope Parameter

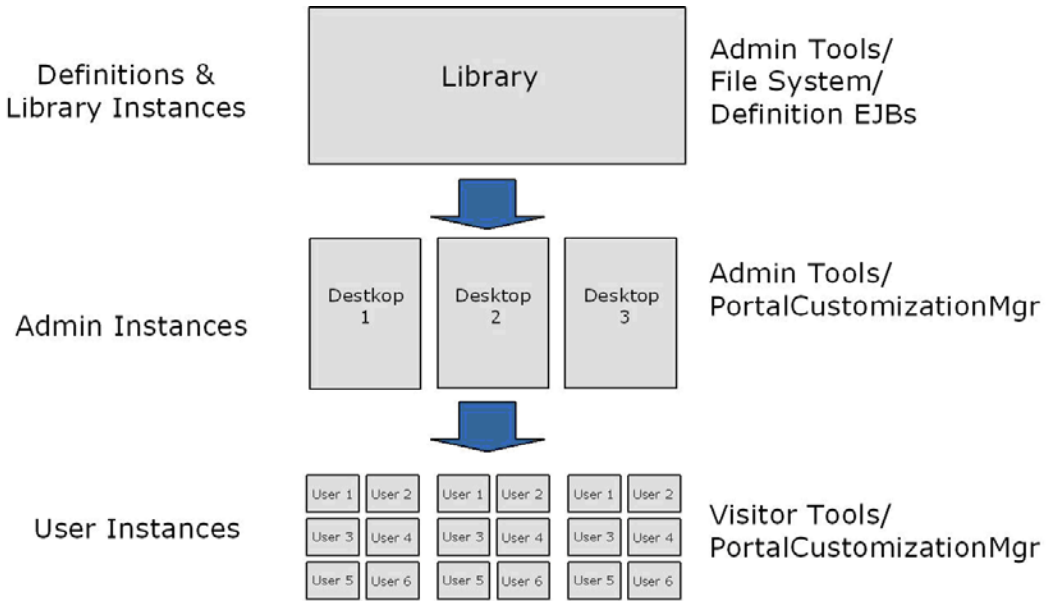
The scope parameter lets you fine tune certain REST commands. This parameter can affect the data retrieved as well as the data updated. The scope parameter can take one of these three values:

- **library** – An object (book, page, or portlet) can exist in the library and not be referenced by any desktop. A developer or administrator may create a page outside the context of a desktop. This page can then be placed on a desktop at a later point in time. Changes made to objects in the library are cascaded down through all desktops.
- **admin** – The admin scope represents the "default desktop" this is where the administrator makes changes to a individual desktop. So an admin or community leader can make changes to a page and those changes are not reflected in the library or on other desktops. Those changes may get cascaded down to the visitors view though.
- **visitor** – (default) Visitors, or end users, are allowed to customize their desktop according to their personal preferences. Changes made to one visitor's view do not show up in the default desktop nor do they show up in other visitor's views.

  Note: If scope is not specified then the default value is visitor for regular non-privileged users. If, however, the user has administrator's rights, then the default values are read from the session. This value can be set using the adminscope REST command.

Figure 7-2 shows the hierarchy of this scoping in a portal application, the direction of proliferated changes and what tools are typically used to modify at the different levels.

Figure 7-1 Scoping Hierarchy



If the value of the scope parameter is set to visitor, then you can optionally specify a user name if you have permissions to do so. WLS administrators, WLP administrators, and community leaders may make changes to other user's desktops; however, a regular user can only affect his or her own desktop.

Example:

```
/bea/wlp/api/adminscope/item?webapp=mywebapp&portal=portalpath&desktop=
desktoppath&scope=visitor
```

This command puts a special token in the HttpSession. This same token is used when an individual user makes a request for a desktop. So if the user "weblogic" logs into a desktop the framework looks at this value to determine what view to bring back. If the value is admin then bring back the default view, otherwise bring back the administrator's personal view.

Tip: A best practice is to always specify the scope parameter instead of trying to rely on what is in the session because the session can timeout at any point.

REST Command Example

The following REST command retrieves a list of portlets from a portal web application called myWeb that contains just two portlets.

```
http://localhost:7001/myWeb/bea/wlp/api/portlet/list?webapp=myWeb
```

Listing 7-1 shows an example of the type of XML data (the default format) returned by this REST command. This portlet information (such as the portlet label) can then be parsed out and used as input to another REST command, such as a command to add a portlet to a page.

Listing 7-1 Example Response from the List Portlets REST Command

```
<rsp>
   <portlet_summaries>
      <portlet_summary>
         <label>p1</label>
         <title>Portlet One</title>
      </portlet_summary>
      <portlet_summary>
         <label>p2</label>
         <title>Portlet Two</title>
      </portlet_summary>
   </portlet_summaries>
</rsp>
```

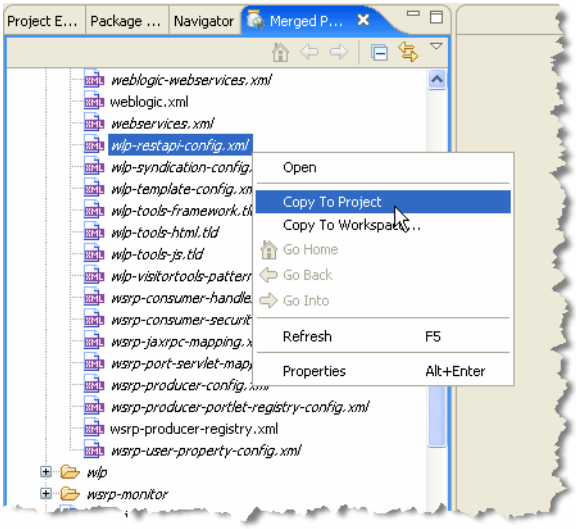
Tip: By default, WLP REST commands return data in XML format. You can add the parameter `format=json` to a command to return a JSON (JavaScript Object Notation) object rather than XML to the client. It is easy to convert the JSON response to a JavaScript object. Once the data is in object form, you can use JavaScript to iterate and access values using familiar dot notation, such as `item.title` and `item.markupName`. See also "JSON" on page 2-4.

Disabling REST Commands

You can disable specific REST commands by editing the config file `wlp-restapi-config.xml`. To disable REST commands, do the following:

1. Select the Merged Projects view in WorkSpace Studio.
2. Open the `WEB-INF` folder of the web application.
3. Right-click `wlp-restapi-config.xml` and select Copy To Project, as shown in Figure 7-2. This copies the file from its library module to your project so that you can edit it.

Figure 7-2 Selecting the REST Config File



4. Open the file in the editor.
5. Locate the REST command and set the value of the <enable> element to **false**, as shown in Listing 7-2.

Listing 7-2 REST Command Definition

```
<rest-handler>
    <name>getPortlets</name>
    <path>portlet/list</path>
    <handler-class>com.bea.wlp.rest.portlet.command.GetPortlets
    </handler-class>
    <methods>
        <method>
            <name>get</name>
```

```
            <enable>false</enable>
        </method>
    </methods>
</rest-handler>
```

Basic WLP REST API Examples

This section gives examples of some WLP REST commands and their responses. For a more complete example that includes using Disc to obtain portal information, see "Using REST and Disc" on page 7-10.

Retrieving Information About a Single Portlet

The following REST command returns details on a single portlet with the label revenue_1 from the web application called "banking." This result is shown in Listing 7-3.

```
http://blaster:7041/banking/bea/wlp/api/portlet/details/revenue_1?webapp=dvt
```

Listing 7-3 Portlet Details

```
<rsp>
    <title>5 Largest Customers</title>
    <content_uri>/portlets/revenue/index.jsp</content_uri>
    <forkable>false</forkable>
    <fork_render>false</fork_render>
    <is_public>true</is_public>
    <cacheable>false</cacheable>
    <cache_expires>-1</cache_expires>
    <portlet_file>/portlets/revenue/revenue.portlet</portlet_file>
    <deleted>false</deleted>
    <webapp>dvt</webapp>
    <wsrp_user_properties_mode>3</wsrp_user_properties_mode>
    <state_change_flag>0</state_change_flag>
    <requires_url_templates>false</requires_url_templates>
    <templates_stored_in_session>false</templates_stored_in_session>
    <producer_offered_portlet>false</producer_offered_portlet>
    <created_date>2007-10-31 15:54:19.0</created_date>
```

```
    <modified_date>2007-10-31 15:54:19.0</modified_date>
</rsp>
```

Tip: Each of the various portal artifacts (such as pages and books) can be queried to return similar information. Refer to the REST API reference documentation on e-docs for the appropriate command syntax.

Changing the Title of a Portlet

REST commands that update portal data require an HTTP POST method. Listing 7-4 shows a simple script that uses the XMLHttpRequest.open method to post the REST URL.

Listing 7-4 Updating with a POST Method

```
<script type="text/javascript">
  var urlStem =
     'http://localhost:7001/myWeb/bea/wlp/api/portlet/item/restTest_1';
  var params =  "webapp=myWeb&title=New Title";
  var xmlhttp = new bea.wlp.disc.io.XMLHttpRequest();
  var url = urlStem + "?" + params;

  xmlhttp.open('POST', urlStem, true);
  xmlhttp.send(params);

</script>
```

Moving a Book or Page

The REST commands let you create, delete, and modify objects. Here is an example that illustrates how to move a book from one page to another.

```
<protocol>://<hostname>:<port>/<webapp>/bea/wlp/api/<type>/<action>/<childtype>/
<label>?webapp=<webapp>&format=<format>
```

Where childtype is a child of the object specified by the type and with the label. For example, moving a page within a book would have a URL such as:

```
http://wlp.bea.com/flatweb/bea/wlp/api/book/move/bookorpage/mainbook_01
```

Where mainbook_01 is the label of the book to move the page on. The POST parameters would look like the following:

```
webapp=flatweb
portal=flatirons
desktop=flatweb
scope=visitor
label=mypage
position=0
alignment=0
```

The label is the label of the page to move and the position is the position within the book to move to. The scope can be specified as visitor, admin, or library, allowing an admin user to make changes to default instances, library instances, and so on.

Note: The DOM does not automatically reflect changes. Use DHTML on the client to update the page.

Using REST and Disc

REST commands by themselves are not sufficient to satisfy all use cases. You need to use the Disc JavaScript API to retrieve portal context objects and the information they contain. This information can then be added to a REST call to update the portal. For example, to move a portlet, you need to use the Disc APIs to get information about the portlet's current location and add this information to the REST call. The Disc framework is discussed in Chapter 3, "The WLP Disc Framework."

The following example sections illustrate many of the key concepts around using REST and Disc for client-side portal UI development.

The following sections discuss parts of a sample JSP portlet that is shown in its entirety in "Putting It All Together" on page 7-14. The portlet lets the user change the page layout by selecting a new layout from a drop down list. The list is populated by querying the portal with the Disc API, and a user's selection is communicated to the server with a REST command.

Constructing a REST URL Using Disc Context Objects

Using the information obtained from a Disc context object, you can call REST API commands to update the portal. Most REST commands require parameters that can only be obtained from

context objects. REST command parameters are listed and described in the WLP REST Command Reference on e-docs.

The function in Listing 7-5 uses Disc context objects to obtain information from the portal that is used to construct REST command parameters. The code is straightforward JavaScript. The REST command requires such parameters as the portlet context, page context, portal path, desktop path, and others. After the parameters are gathered, the REST command URL is constructed.

Listing 7-5 Constructing a REST Command

```
...
function ${uid}setLayout(layoutName) {
      // Get the canvas element to use for output
      var canvas = document.getElementById("${uid}canvas");

      // Get the portlet context from Disc
      var portletContext = bea.wlp.disc.context.Portlet.findByElement(canvas);

      // Get the parent page context for the portlet
      var pageContext = portletContext.getParentPage();

      // Get the application instance from Disc
      var appContext = bea.wlp.disc.context.Application.getInstance();

      // Create the parameters to pass to the REST command
      var params = "";
      params += "&portal=" + appContext.getPortalPath();
      params += "&desktop=" + appContext.getDesktopPath();
      params += "&webapp=" + appContext.getWebAppName();
      params += "&layout=" + layoutName;

      // Construct the URL for the REST command to change page attributes
      var url = "/" + appContext.getWebAppName();
      url += "/bea/wlp/api/page/item/" + pageContext.getLabel();
...
```

Open and Send an XHR Request

After you construct a request to send a REST command, you need to send it to the server. The recommended practice is to use an XHR object to send the command and retrieve the results.

Listing 7-6 shows a code sample that opens an XHR request using a URL and sends a parameter list. The URL and parameter list could be constructed in a similar manner to the techniques discussed in Listing 7-5.

Listing 7-6 Opening and Sending an XHR Request

```
...
// Get a portal-aware XMLHttpRequest from Disc
      var xmlHttpReq = new bea.wlp.disc.io.XMLHttpRequest();
      xmlHttpReq.onreadystatechange = function() {
        if (xmlHttpReq.readyState == 4) {
          if(xmlHttpReq.status == 200) {
            var msg = "Layout changed to " + layoutName + ".\n\n";
            msg += "Press OK to refresh desktop.";
            if (confirm(msg)) {
              window.location.reload();
            }
          } else {
            alert("Unable to change layout.");
          }
        }
      };
      xmlHttpReq.open('POST', url, true);
      xmlHttpReq.send(params);
    }
...
```

Handle the REST Response

Some REST commands return a simple status response. Other commands return additional data, such as detailed information about a portlet. The code sample in Listing 7-7 shows how response data from a REST command might be handled. In this case, the response is a list of available portal layouts that were requested. The response data is passed to a function that parses the list

and displays it in a drop down menu. Note that the REST command and its parameters are constructed in a similar manner to that described previously in Listing 7-5.

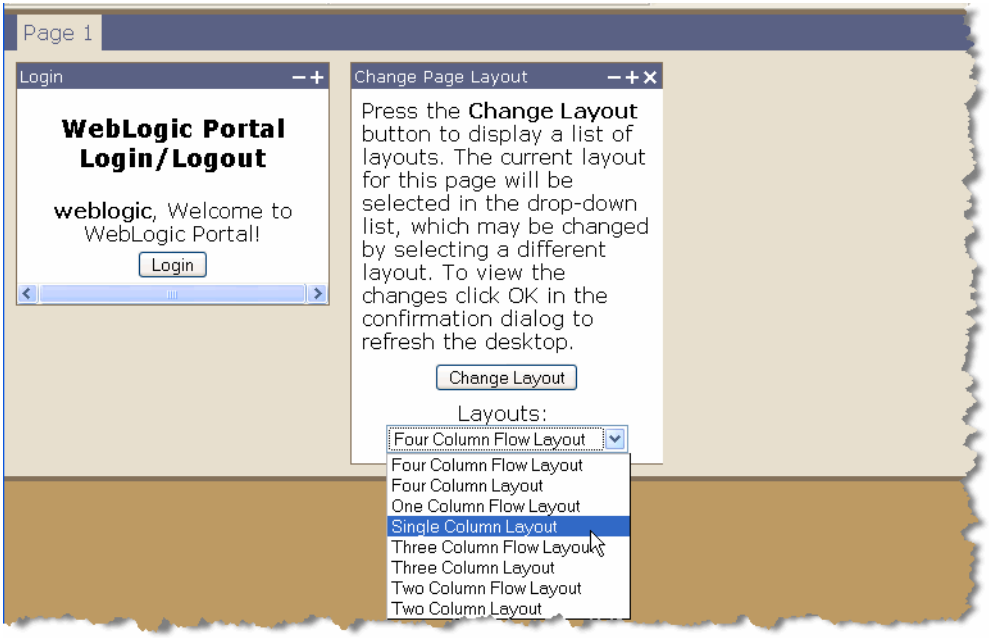
Listing 7-7 Handling REST Response Data

```
...
      // Get the application instance from Disc
      var appContext = bea.wlp.disc.context.Application.getInstance();

      // Construct the parameters for the REST command
      var params = "";
      params += "?portal=" + appContext.getPortalPath();
      params += "&desktop=" + appContext.getDesktopPath();
      params += "&webapp=" + appContext.getWebAppName();
      params += "&format=json";

      // Construct the URL for getting the list of layouts from REST
      var url = "/" + appContext.getWebAppName();
      url += "/bea/wlp/api/layout/list";
      url += params;

      // Get a porttal-aware XMLHttpRequest from Disc
      var xmlHttpReq = new bea.wlp.disc.io.XMLHttpRequest();
      xmlHttpReq.onreadystatechange = function() {
        if (xmlHttpReq.readyState == 4) {
          if (xmlHttpReq.status == 200) {
            // Get the data from the response
            var data = eval('(' + xmlHttpReq.responseText + ')');
            ${uid}displayLayouts(data)
          } else {
            alert("Unable to retrieve layouts.");
          }
        }
      };
      xmlHttpReq.open('GET', url, true);
      xmlHttpReq.send(null);
...
```

Putting It All Together

Listing 7-8 shows a complete JSP file that uses REST and Disc to provide UI for changing the layout of a portal. Note that all of the UI code is on the client-side. Disc is used to obtain information from the portal, REST is used to update the portal, and XHR is used to make asynchronous requests to the server. Figure 7-3 shows the Change Layout portlet in a portal. Note that the portlet requires that the user be logged in.

Figure 7-3 Sample Change Layout Portlet



Listing 7-8 Sample Portlet JSP

```
<%--  Copyright (c) 2006-2008 by BEA Systems, Inc. All Rights Reserved. --%>
<jsp:root
  version="2.0"
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:render="http://www.bea.com/servers/portal/tags/netuix/render"
>
  <jsp:directive.page session="false" />
  <jsp:directive.page isELIgnored="false" />
```

```
<render:encodeName name="" var="uid"/>
<div id="${uid}canvas" style="display: none;">
  <div style="text-align: left; margin: 8px; white-space: normal;">
  Press the <span style="font-weight: bold">Change Layout</span> button
  to display a list of layouts.  The current layout for this page will
  be selected in the drop-down list, which may be changed by selecting
  a different layout.  To view the changes click OK in the confirmation
  dialog to refresh the desktop.
  </div>
  <div style="text-align: center; margin: 8px;">
    <button name="changeLayout" onclick="${uid}getLayouts();">
      Change Layout
    </button>
  </div>
  <div
    id="layouts"
    style="text-align: center; margin: 8px; visibility: hidden;">
    Layouts: 
    <select
      id="${uid}layoutSelect"
      onchange="${uid}setLayout(this.options[this.selectedIndex].value)">
    </select>
  </div>
</div>
<div id="${uid}message" style="text-align: center; padding: 16px;">
  <span id="${uid}discMsg" style="display: none">
    Please enable Disc to use this portlet.
  </span>
  <span id="${uid}desktopMsg" style="display: none">
    Please use this portlet on a portal desktop.
  </span>
  <span id="${uid}loginMsg" style="display: none">
    Please login to use this portlet.
  </span>
  <span id="${uid}customizeMsg" style="display: none">
    Please enable customization to use this portlet.
  </span>
</div>
<script type="text/javascript">
  function ${uid}loadInit() {
    if (typeof bea == "undefined") {
```

```
      document.getElementById("${uid}discMsg").style.display = "";
    } else {
      // Get the application instance from Disc
      var appContext = bea.wlp.disc.context.Application.getInstance();

      if (appContext.getDotPortal()) {
        // Inform the user that the portlet must be on a desktop
        document.getElementById("${uid}desktopMsg").style.display = "";
      } else if (!appContext.getUserName()) {
        // Inform the user that they must login go use the portlet
        document.getElementById("${uid}loginMsg").style.display = "";
      } else if (!appContext.getCustomizationEnabled()) {
        // Inform the user that Disc must be enabled
        document.getElementById("${uid}customizeMsg").style.display = "";
      } else {
        // Display the canvas div and hide the message div
        document.getElementById("${uid}canvas").style.display = "";
        document.getElementById("${uid}message").style.display = "none";
      }
    }
  }

  function ${uid}getLayouts() {
    // Get the application instance from Disc
    var appContext = bea.wlp.disc.context.Application.getInstance();

    // Construct the parameters for the REST command
    var params = "";
    params += "?portal=" + appContext.getPortalPath();
    params += "&desktop=" + appContext.getDesktopPath();
    params += "&webapp=" + appContext.getWebAppName();
    params += "&format=json";

    // Construct the URL for getting the list of layouts from REST
    var url = "/" + appContext.getWebAppName();
    url += "/bea/wlp/api/layout/list";
    url += params;

    // Get a porttal-aware XMLHttpRequest from Disc
    var xmlHttpReq = new bea.wlp.disc.io.XMLHttpRequest();
    xmlHttpReq.onreadystatechange = function() {
```

```
    if (xmlHttpReq.readyState == 4) {
      if (xmlHttpReq.status == 200) {
        // Get the data from the response
        var data = eval('(' + xmlHttpReq.responseText + ')');
        ${uid}displayLayouts(data)
      } else {
        alert("Unable to retrieve layouts.");
      }
    }
  };
  xmlHttpReq.open('GET', url, true);
  xmlHttpReq.send(null);
}

function ${uid}displayLayouts(data) {
  // Get the select element to use for output
  var select = document.getElementById("${uid}layoutSelect");

  // Get the portlet context from Disc using the select element
  var portletContext = bea.wlp.disc.context.Portlet.findByElement(select);

  // Get the parent page context for the portlet
  var pageContext = portletContext.getParentPage();

  // Get the current layout for the page
  var layoutContext = pageContext.getLayout();

  // Get the markup name for the layout
  var layoutName = layoutContext.getMarkupName();

  // Set the visibilty of the parent div for the select
  var div = select.parentNode;
  div.style.visibility = "visible";

  // Clear the select box
  select.options.length = 0;

  // Get the layout details from the response data
  var layoutDetails = data.content.layouts;
  var layoutDetail = null;
```

```
  // Iterate over the layoutDetails and create options for each layout
  for (var i = 0; i &lt; layoutDetails.length; i++) {
    layoutDetail = layoutDetails[i];

    // Create a new option element for the layout details
    option = document.createElement("option");
    option.value = layoutDetail.markup_name;
    option.innerHTML = layoutDetail.title;

    // Compare the markup name to the current layout and select the match
    if (layoutDetail.markup_name == layoutName) {
      option.selected = "selected";
    }

    select.appendChild(option);
  }
}

function ${uid}setLayout(layoutName) {
  // Get the canvas element to use for output
  var canvas = document.getElementById("${uid}canvas");

  // Get the portlet context from Disc
  var portletContext = bea.wlp.disc.context.Portlet.findByElement(canvas);

  // Get the parent page context for the portlet
  var pageContext = portletContext.getParentPage();

  // Get the application instance from Disc
  var appContext = bea.wlp.disc.context.Application.getInstance();

  // Create the parameters to pass to the REST command
  var params = "";
  params += "&portal=" + appContext.getPortalPath();
  params += "&desktop=" + appContext.getDesktopPath();
  params += "&webapp=" + appContext.getWebAppName();
  params += "&layout=" + layoutName;

  // Construct the URL for the REST command to change page attributes
  var url = "/" + appContext.getWebAppName();
  url += "/bea/wlp/api/page/item/" + pageContext.getLabel();
```

```
      // Get a portal-aware XMLHttpRequest from Disc
      var xmlHttpReq = new bea.wlp.disc.io.XMLHttpRequest();
      xmlHttpReq.onreadystatechange = function() {
        if (xmlHttpReq.readyState == 4) {
          if(xmlHttpReq.status == 200) {
            var msg = "Layout changed to " + layoutName + ".\n\n";
            msg += "Press OK to refresh desktop.";
            if (confirm(msg)) {
              window.location.reload();
            }
          } else {
            alert("Unable to change layout.");
          }
        }
      };
      xmlHttpReq.open('POST', url, true);
      xmlHttpReq.send(params);
    }

    // Add an onload function for this portlet
    var ${uid}oldonload = (window.onload) ? window.onload : function () {};
    window.onload = function () { ${uid}oldonload(); ${uid}loadInit(); };
  </script>
</jsp:root>
```