In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter NO.6 "Portals and AJAX"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Ramanujam Rao is a software engineer, architect, and trainer specializing in building large-scale enterprise applications. He has over 13 years of experience in designing and developing complex web architectures, including portals, and helps enterprises in building scalable, distributed applications on the JEE platform.

He writes frequently on enterprise architecture, and actively consults in the field of information technology management, including technology platforms, technology strategy, and application delivery.

He has a B.S. in Electrical Engineering, an M.S. in Computer Science, and an MBA from Ohio State University. He currently lives and works in Columbus, OH, USA.

I'd like to thank my wife, Bharathi, and my daughter Gitanjali, for their constant support and for managing things during my long absences in the course of writing this book. I'd also like to acknowledge our parents, whose sacrifices and encouragement have always been part of everything I do.

I'd like to acknowledge the technology community in general, whose shoulders I stand on, including the folks at JBoss portal, and my colleagues at Nationwide Insurance.

Finally, a big thanks goes to the technical reviewers Russ and Prabhat, whose feedback was invaluable, and the entire Packt editorial team for their dilligence in getting the book out in a great shape.

For More Information:
JBoss Portal Server Development

Enterprises need more than just basic services; they need value-creating entities, which are crucial for running a successful business. Portals offer tremendous value to enterprises, and JBoss Portal Server is a popular, feature-rich open-source server that provides a standards-compliant platform for hosting functionality that serves the diverse portal needs of an enterprise. Its primary strength lies in its ability to provide robust support for custom implementation of functionality using the JSR-168 portlet API.

This book is a practical guide to installing, configuring, and using JBoss Portal Server. It explains, with examples, how to easily build feature-rich portals using JBoss. As you move further on, you will learn to personalize your portals and add new features to them. This book will equip you with everything you need to know about JBoss Portal Server to build a fully-functional portal. It will help you to quickly understand and build enterprise portals with rich features, such as personalization, AJAX, single sign-on, Google widget integration, remote portlet integration, content management, and more. Along with feature implementation, the book will also provide developers with enough detail to be able to tune and customize the portal environment to best suit the platform needs.

What This Book Covers

*Chapter 1* reviews portals, their functions, and their values. It talks about portal servers and the specifications that govern the creation and management of portals on the J2EE platform. Portal servers go beyond serving custom content and provide a feature-rich set of robust pre-built functions that take away the need to create certain fundamental sets of features from scratch each time. This chapter shows that, by removing the development efforts behind creating such features, portal developers can now spend their time and money on developing business functions.

*Chapter 2* talks in detail about the installation process of JBoss portal, with an emphasis on the differences and caveats for the various installation types offered, depending on the usage scenarios. A simplified installation and deployment process facilitates faster implementation and fewer problems, as demonstrated by almost immediate creation and management of pages on the platform.

*Chapter 3* goes a bit deeper into JBoss portal server and explains portlets better by creating a simple portlet application. It goes through the complete life cycle from code creation to deployment. This overview tour gives you a good idea of the major components that are required to create a functional portlet.

For More Information:
**Chapter 4** reviews the various options that are available to effectively manage the presentation of portlets using technologies such as JSP, JSF, and so on. It shows a few examples of each one of them. To understand the concepts better, a portal application is created from scratch and a custom portlet, created with JSP-based view is added to this new application.

**Chapter 5** reviews how the power of portals can be extended by facilitating features such as customization and personalization. It further extendeds our example portal to include custom layouts, themes, and other personalization features. It also shows how we can personalize a page and offer the users options for controlling the contents on the page.

**Chapter 6** shows how JBoss portal blends the dynamism and rich functionality offered by AJAX with its strong portal architecture, to provide users with choices for developing highly-functional portal applications. It also discusses the limitations of the current specification and walks through an example that shows how easy it is to develop and deploy AJAX-based portlets.

**Chapter 7** talks about how Hibernate, a very popular ORM tool, is used internally by JBoss applications, and how applications can integrate database support into portlet applications by using Hibernate.

**Chapter 8** elaborates upon a simple but robust content management system provided by the JBoss portal that is sufficient for most of the needs for a portal application. Using interceptors, CMSAdmin, and CMS Portlets, the user can develop a functionality that helps to effectively manage and deliver content. This chapter extends our example further, to add some new content, and then edit it. It also shows how easy it is to add, edit, and manage content in the portal.

**Chapter 9** discusses the various aspects of security as they relate to JBoss portal server and its functional components—the portal objects. JBoss portal allows a fine-grained level of control over portal objects such as portal instances, pages, and portlets. Security is an important function of an application. JBoss portal offers a varied set of options that allow the building of highly secure enterprise applications on the portal server.

**Chapter 10** discusses the basics of remoting portlets before it goes into a few implementations using some real-world examples. It talks about how easily the portlets can be exposed as remotely available services, and how remote services can be consumed relatively effortlessly.

**Chapter 11** talks about some of the features specified by the new portlet specification, such as portlet co-ordination, and filters introduction. It tells us how Portlet 2.0 provides a comprehensive set of options for performing robust portlet coordination by using events, as well as public parameters that tremendously increases the capabilities of portals and portlets by opening up possibilities for integrating not only within the application, but also with other applications within the enterprise.

---

**For More Information:**
Portals and AJAX

Dynamic, attractive, and rich interfaces have always helped in creating compelling web applications. However, in recent years, the popularity and support for asynchronous browser-to-server communication, combined with the strong client-side capabilities of JavaScript and XML, have resulted in exciting user interfaces with unprecedented functionality. **AJAX (Asynchronous JavaScript and XML)**, as the rich user interface design is commonly called, has gained wide acceptability in traditional web applications, and portals are no exception to this.

JBoss portal blends the dynamism and rich functionality offered by AJAX with its strong portal architecture to provide users with choices for developing highly-functional portal applications. The latest portlet specification, JSR-286, provides an easy and intuitive approach to implementing asynchronous calls that was not possible in JSR-168. In this chapter, we will also discuss the limitations of the current specification and walk through an example portlet to understand how AJAX can enhance portlets.

**Rich user interfaces and AJAX**

Rich user interfaces can be achieved by using a combination of dynamic HTML elements such as HTML and JavaScript. However, the scope of such an interface is limited to client-side behavior and has minimal functional implications due to the lack of server-side interactions. The power of **AJAX** is in its capability to provide even richer interface by supplementing its dynamic user interface with powerful functionality through seamless server-side invocation power. AJAX allows individual user interface components to communicate with the server and exchange data without the need for refreshing the whole screen. This is achieved using a process called **Web Remoting**. Web remoting, or the process of communicating between a browser and a server, can be performed in multiple ways. The popular approaches that are supported by today’s browsers are **IFrames** and **XMLHttpRequest**. Dynamic HTML can be complemented with either of these methods to generate AJAX functionality.
Asynchronous JavaScript and XML or AJAX

Asynchronous communication between the client and the server forms the backbone of AJAX. Although an asynchronous request-response method can provide significant value in the development of rich functionality by itself, the results are lot more pronounced when used in conjunction with other functional standards such as CSS, DOM, JavaScript, and so on. The predominant popularity of AJAX stems from such usage.

Client-server communication can be achieved either by using IFrames, or by using the supported JavaScript function call XMLHttpRequest(). Due to certain limitations of IFrames, XMLHttpRequest has gained a lot more acceptance. While IFrame can also be an effective option for implementing AJAX-based solutions, in this chapter, we will focus largely on an XMLHttpRequest-based implementation. The primary advantage of using AJAX-based interfaces is that the update of content occurs without page refreshes. A typical AJAX implementation using XMLHttpRequest happens as described in the following steps:

1. An action on the client side, whether this is a mouse click or a timed refresh, triggers a client event
2. An XMLHttpRequest object is created and configured
3. The XMLHttpRequest object makes a call
4. The request is processed by a server-side component
5. The component returns an XML (or an equivalent) document containing the result
6. The XMLHttpRequest object calls the callback() function and processes the result
7. The HTML DOM is updated with any resulting values

The following simplified image illustrates the high-level steps involved in an AJAX request flow. The portal client page gets served to the client browser, where the execution of JavaScript functions takes place.
The following example illustrates the initialization of the request object and its basic use:

```javascript
if (window.XMLHttpRequest) { // Object of the current window
    // for non-IE browsers
    request = new XMLHttpRequest();
} else if (window.ActiveXObject){
    // For IE
    request = new ActiveXObject("Microsoft.XMLHTTP");
}
request.onreadystatechange = function()
{
    // do something to process response
};
if (request.readyState == 4){
    // everything received, OK. Do something now..
} else {
    // wait for the response to come to ready state
}
```
In subsequent sections, we will modify our sample portal application by adding AJAX functionality to one of the portlets.

**AJAX in JBoss portal**

AJAX has gained tremendous popularity in the traditional web application development world due to the richness and agility that it brings to user interfaces. Portals, such as JBoss portal, can also gain significantly from AJAX, in terms of implementation of both behavior and functionality.

Refreshing the page content tends to be a time-consuming and resource-intensive process. Every request that a user makes to the server, either by clicking on submissions or links, results in the portal calling `doView()` and a series of methods for each portlet on the page, one at a time, before aggregating the results and sending the response back to the browser. Using AJAX allows for simultaneous submissions of request in their own independent threads of execution, resulting in an asynchronous and parallel execution. The portal page refresh overhead is now only as long as the time consumed by the slowest portlet.

The response times observed by the user improve dramatically, while at the same time allowing more functionality on pages. Architecturally, vertical independent stacks of execution facilitate cleaner and more modular designs and implementations.

AJAX can be implemented in JBoss portal in the following two ways:

1. Using in-built support for asynchronous portal behavior by using configurations.
2. Writing custom behavior in portlets and page content by using AJAX libraries.

The in-built support for asynchronous behavior comprises of support for both markup and content. The markup support is in layouts and renderers, while the content is supported through configurable drag-drop and partial page refresh behavior. Almost all of the AJAX behavior supported by JBoss portal relates to asynchronous communication between the client and the portal servers. The only exception is the drag-drop behavior, which is largely view functionality.

As far as the custom development within a portlet is concerned, the options are innumerable. A portlet can be developed using many advanced frameworks that are available as either commercial or open source products. For example, user interface features such as drag-and-drops, grids, accordion selects, pull-down menus, content refresh, and so on can be implemented by using third-party libraries including Scriptaculous, JQuery, and DOJO, which have gained a strong following among developers, even on traditional applications and non-portal platforms.
In the next few sections, we will walk through an example of AJAX-enabled portlets using one of these libraries, developed on the JBoss portal platform. However, before we go into the implementation, let's step back and understand the limitations that the current portlet specification—JSR-286—addresses, facilitating easy development of AJAX portlets.

**JSR-168 AJAX limitations**

Before we look at the features and options provided by the new specification, let's look at how traditional JSR-168 portlets functioned. As shown in the following figure, the "Action" request invoked the `processAction` method on the server, which implemented controller logic to route it to the correct view. The "Render" request then invokes the `render` method to serve the content page to the browser.

However, when the portlet uses AJAX and needs to makes an asynchronous call, it has to use `ActionURL`. This in turn follows the standard processing when `processAction` processes the request and the `render` method creates the user interface. However, now when the user interface is sent back, the portal injects some other markup and recreates the entire portal page. Hence, instead of refreshing a snippet of user interface, we end up refreshing the whole page.
The issues with JSR-168 and AJAX can be broadly summarized as follows:

- **ActionURL** and **RenderURL** point to a portal, and not to a portlet. When we point to a portal, the result is a complete portal page, even if the portlet generates only a snippet.

- As per the specification, the user interface rendered by the portlet is supposed to be aggregated with some other markup and served back to the browser. When more than only the necessary data and markup is sent back, the JavaScript code on the client side that makes the asynchronous call cannot process the request.

- Asynchronous calls are made through XMLHttpRequest, which is designed to consume and process the complete response from the portlet. With the portal processing the request in between, XMLHttpRequest cannot consume the original response for processing.

This defeats the purpose and value of using asynchronous calls to the server, and we end up with traditional full page refreshes.

There were obviously a few workarounds to this. The most common practice was to serve the request from outside of the portal container into the web container. The idea is that the AJAX call can still be made to **ActionURL**, but the **render** function copies or shares its context with a traditional Java servlet in the web container of the application server. The AJAX call can now make a direct request to the servlet and get an asynchronous response from the servlet with no interference from the portal.
However, this is, at best, a temporary solution with limited options. Shared contexts and sessions can be invalid or stale, and it is not always possible to expose the servlets from the web container.

There was a need for a better solution, and one that was incorporated as part of the specification. JSR-286, the latest portlet specification, addresses these problems.

**JSR-286 and AJAX**

Among the set of features that the new portlet specification JSR-286 introduces to make things easier for AJAX implementations, are the options to directly communicate with the portlet through shared render parameters and resource serving.

Referring back to the workaround option in JSR-168, the new specification provides a standard for setting and receiving render parameters, and we no longer have to deal with manual copying of context and parameters.

To set the render parameter directly to the response, all we have to do is this:

```java
public class AssetPortlet extends GenericPortlet {

    public void processAction(ActionRequest request, 
                           ActionResponse response) {

        response.setRenderParameter("rate", "4.5");

    }

    ...

}
```

To receive render parameters in the portlet directly from the view, we can say:

```java
public class AssetPortlet extends GenericPortlet {

    public void render(RenderRequest request, RenderResponse response) {

        String rate = request.getParameter("rate");

        ...

    }

    ...

}
```

Similarly, the resource servicing option allows us to directly invoke the portlet from the portlet user interface, as follows:

```html
ResourceURL resourceURL = renderResponse.createResourceURL()
<form action="<%= resourceURL %>
method="POST">
    ...
</form>
```
Using ResourceURL invokes a new life cycle method called serveResource() on the server, as follows:

```java
public void serveResource(ResourceRequest req, ResourceResponse resp)
    throws PortletException, IOException
{
    resp.setContentType("text/html");
    PrintWriter writer = resp.getWriter();

    writer.print("<p> <b>Response Markup </b> ");
    writer.print("<b>Response Markup </b>");
    writer.close();
}
```

So, now when we use ResourceURL instead of ActionURL, the implementation of the asynchronous requests appears as follows:

Hence, we can conclude that the following approaches are available for implementing a portlet user interface:

- For a traditional, JSR-168 based implementation, ActionURL and RenderURL are still options, although very limited ones, for asynchronous call implementations
- For an AJAX implementation and asynchronous request-response, the ResourceURL option offered by JSR-286, is the best.

JBoss 2.7.0 offers full support for the JSR-286 Portlet 2.0 specification, allowing us to implement robust AJAX-based applications without any workarounds.

For More Information:
With this background, let’s dive right into developing a portlet for our "MyCompany" portal page using AJAX support. In subsequent sections, we will discuss the configuration and other in-built AJAX support provided by the JBoss portal server.

**Developing an AJAX portlet**

Let us enhance our sample application, the "MyCompany" portal, and implement AJAX functionality in one of the portlets, using asynchronous server calls and dynamic HTML.

We will build a simple asset allocation calculator which asks for the user's age and returns an asset allocation mix of stocks and bonds based on a simple age-based logic on the server side. The user interface will submit the request asynchronously to a portlet on the server, which will then process it and return a markup that will only refresh a segment of the page.

We will also use some special user interface DHTML effects to demonstrate the effective combination of asynchronous calls with a rich user interface.

**The front-end**

We will create a new portlet called "AJAXPortlet" in the "MyPortal" project, and position it in the center column below our current preferences portlet that we created in the last chapter.

The myportal-object.xml shows the following entry:

```xml
<deployments>
  ...
  <deployment>
    <if-exists>overwrite</if-exists>
    <parent-ref>mycompany</parent-ref>
    <page>
      <page-name>default</page-name>
      <window>
        <window-name>MyCompany AJAX Portlet</window-name>
        <instance-ref>AJAXPortletInstance</instance-ref>
        <region>center</region>
        <height>2</height>
      </window>
    </page>
  </deployment>
  ...
</deployments>
```
Let's add our form to the JSP:

```html
<!--AJAX submission via form
Actual post takes place in Javascript function -->
<form method='post'
    id="assetform"
    name="assetcalc"
    action=""
    onsubmit="ajaxSubmit('<%= renderResponse.createResourceURL() %>', 'testdiv',
    Form.serialize(this), true); return false;">

    <p>Your Age: <input type="text" name="age" />
    <br><br>
    <input id="page" type="hidden" name="page" value="asset" />
    <input id="submit" type="submit" name="Submit" />
</form>
```

As we can see, the form has an `onSubmit` option to delegate submission to a JavaScript function. We are also using `ResourceURL`, generated from `renderResponse.createResourceURL()`, to submit our form. This allows us to talk directly to the portlet instead of to the portal. `testdiv` demarcates the location of the result.

Although we are using the direct API here for illustrative purposes, we can use the Portlet 2.0 tag library provided by JBoss portal in our JSP, in which case the following `taglib` declaration needs to be added:

```html
<%@ taglib uri="http://java.sun.com/portlet_2_0"
    prefix="portlet" %>
```

The `ResourceURL` looks like this:

```
/portal/portal/mycompany/default/MyCompany+AJAX+Portlet?action=b&cacheability=PAGE
```

The `cacheability=PAGE` parameter indicates that we want the URL to contain the state of the complete page, including values such as PortletURLs, and so on.

Now, let's add some JavaScript code to process the form. It is useful to note that for an AJAX implementation, we rely on JavaScript functions to carry out our submissions and then track responses from the server asynchronously. Although we can write the JavaScript code ourselves, using existing popular libraries gives us a lot more efficiency and a broader feature set. We will use `scriptaculous.js` for the effects and `prototype.js` for asynchronous calls. These are available as standard in the headers of JBoss portal. Hence, we can use them without worrying about adding the library files along with the code. Our AJAX code includes asynchronous calls, and the special effects appear as follows:
Chapter 6

Here, the `Ajax.Updater` function takes the `ResourceURL`, which is the target div for the results, and the parameters entered in the form, to make an asynchronous call to the server. If the effect is enabled, which it is in our example, when the server responds back with a markup, the JavaScript code will slide down, create space, and display the output, while highlighting the information in three colors for emphasis.

We are all set on the client side. Now let's go back and create our portlet.

**The server-side portlet**

Our JSP invokes an instance of `AJAXPortlet` which appears as follows:

```java
public class AJAXPortlet extends GenericPortlet {
    // async requests and responses are processed here
    public void serveResource(ResourceRequest req, ResourceResponse resp) throws PortletException, IOException {
        resp.setContentType("text/html");
        PrintWriter writer = resp.getWriter();
        String age = req.getParameter("age");
        if (age == null || age.equals("")){
            writer.print("<P> Sorry, Please enter valid age between 10 and 100! ");
        }
        else {
            String bond = age;
            int stock = 100 - Integer.parseInt(age);
            writer.print("<P> The recommended asset allocation for your age is: ");
            writer.print("<b>Bonds: \"" + bond + ", Stock: \"" + stock + ");
        }
    }
}
```
Portals and AJAX

writer.close();
}

// parent page delivery
public void render(RenderRequest renderRequest, RenderResponse renderResponse) throws PortletException, IOException {
    PortletContext context = getPortletContext();
    PortletRequestDispatcher rd = context.getRequestDispatcher("/WEB-INF/jsp/ajax_calc.jsp");
    rd.include(renderRequest, renderResponse);
}
}

Let's look at this a bit more closely. The first method, serveResource, gets invoked when the JSP JavaScript performs a submission. We make a quick calculation, open a PrintWriter and write our output markup to it. The output of the PrintWriter goes directly back to the JavaScript call which then parses the information and displays it in the requested format. We also perform a quick validation of the input data to demonstrate how efficient it is to validate data by using AJAX.

Hence, using this single method, we have achieved asynchronous communication between a window on a client browser and a portlet.

The render method follows the traditional approach of serving the portlet window when the page first loads. All we do here is to point it to the JSP page that houses our AJAX calls.

Deployment

Once we build and deploy the portlet, and the home page is loaded, it looks like this:

For More Information:
Let's enter some age as data in the field, and then submit the form.

![Form Example]

Once we enter the data and click on **Submit Query**, we should see the result showing up in the same page in a newly opened area below the portlet, and with some visual effects.

![Form Example]

We can also see validation in action by entering either nothing or a number greater than 100. The server reinterprets the data and responds back with an error message, as shown below:

![Form Example]

We just saw how simple and efficient it is to create AJAX-enabled portlets. Although, this was a simple example, fundamentally, as far as asynchronous call processing is concerned, most scenarios are likely to be very similar. However, the dynamic nature and rich content of the portlet user interface can be dramatically enhanced by adopting the latest DHTML/AJAX libraries.

Up until now, we have been talking about custom development. Let us now look at how JBoss portal leverages AJAX.

**AJAX support for markup**

JBoss portal supports markup through tags on layouts and renderers of the pages. Special tags are added to layout JSPs that facilitate the placement of AJAX features on a page. Similarly, renderers are used to interpret the tags and to render AJAX-driven content. The obvious advantage is the in-built support for the auto-creation and control of AJAX components on portal pages.

**Layout markup**

As discussed earlier, layouts provide a structure for the creation and serving of portal pages. Layouts aggregate all of the content generated by the portlet, based on region and order, merge them with some additional content provided by the portal, and serve a response back to the user. By providing support for AJAX in the layout, helps facilitate the easy development and implementation of dynamic functionality in pages, with minimal effort.

Layout markup is implemented using JSP tags. The JBoss JSP tag library, portlet-layout.tld, offers tags that facilitate the implementation of AJAX features in layouts. A JSP layout can be changed to an AJAX-supported page simply by adding references to the tags. Hence, using tags also helps with the easy implementation of features.

The following is the layout page from the default portal generic layout ${JBOSS_PORTAL_HOME}/server/default/deploy/jboss-portal.sar/portal-core.war/layouts\generic\index.jsp, and shows AJAX support implemented as tags:

```jsp
<%@ page import="org.jboss.portal.server.PortalConstants" %>
<%@ taglib uri="/WEB-INF/theme/portal-layout.tld" prefix="p" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>${PortalConstants.VERSION.toString()}</title>
<meta http-equiv="Content-Type" content="text/html;"/>
<!---- to correct the unsightly Flash of Unstyled Content. -->
```

For More Information:  
<script type="text/javascript"></script>
<!--[if !JavaScript]-->injected the theme, default to the Renaissance theme if
nothing is selected for the portal or the page -->
<p:theme themeName="renaissance"/>
<!--[if !JavaScript]-->insert header content that was possibly set by portlets
on the page -->
<p:headerContent/>
<%@include file="/layouts/common/modal_head.jsp"%>
</head>
<body id="body">
<p:region regionName='AJAXScripts' regionID='AJAXScripts'/>
<%@include file="/layouts/common/modal_body.jsp"%>
<div id="portal-container">
<div id="sizer">
<div id="expander">
<div id="logoName"></div>
<table border="0" cellpadding="0" cellspacing="0"
id="header-container">
<tr>
<td align="center" valign="top" id="header">
<!--[if !JavaScript]-->Utility controls -->
<p:region regionName='dashboardnav' regionID="dashboardnav"/>
<!--[if !JavaScript]-->navigation tabs and such -->
<p:region regionName='navigation' regionID='navigation'/>
<div id="spacer"></div>
</td>
</tr>
</table>
</div>
</div>
</div>
<div id="footer-container" class="portal-copyright">Powered by
Renderer markup

The portal combines the renderers and layouts to generate the final content. Enabling support for AJAX in the renderer just requires adding the statement `<ajax-enabled>true</ajax-enabled>` to the renderer descriptor.

The following example, at `{JBOSS_PORTAL_HOME}/server/default/deploy/jboss-portal.sar/portal-core.war/WEB-INF/layout/portal-renderSet.xml`, shows the renderer configuration of the emptyRenderer RenderSet for AJAX support:

```xml
<renderSet name="emptyRenderer">
  <set content-type="text/html">
    <ajax-enabled>true</ajax-enabled>
    <region-renderer>
      org.jboss.portal.theme.impl.render.empty.EmptyRegionRenderer
    </region-renderer>
    <window-renderer>
      org.jboss.portal.theme.impl.render.empty.EmptyWindowRenderer
    </window-renderer>
    <portlet-renderer>
      org.jboss.portal.theme.impl.render.empty.EmptyPortletRenderer
    </portlet-renderer>
    <decoration-renderer>
      org.jboss.portal.theme.impl.render.empty.EmptyDecorationRenderer
    </decoration-renderer>
  </set>
</renderSet>
```

AJAX support for content

Whereas the layout and renderer contribute to AJAX behavior at the markup level, JBoss portal's support for object-level configuration can be leveraged to provide AJAX support at the page level. The object property inherits a configured behavior from its parent. Currently, two features are offered for AJAX-driven content:
1. **Drag and drop**: Facilitates easy movement of portlets to various locations on screen using the mouse.

2. **Screen Refresh**: Allows sub-components of pages or individual portlets to refresh themselves without refreshing the entire page.

**Drag-and-Drop**

As the name suggests, this feature is triggered by a user action, and allows a portlet to detach itself from a specific location on the page and move to a different location on the page. This allows for the customization of the user interface to a form that is most convenient to the user. The dynamic view behavior comes from a combination of DHTML and asynchronous server-side communication.

Due to the nature of the behavior, drag-and-drop capability is available and effective only in dashboard pages where there are multiple portlets and the page layout can be personalized. The feature is allowed by default on the dashboard, but can be turned off by setting the value in the configuration file to `false`.

The following is a snippet of the default object configuration file (`jboss-portal.sar/conf/data/default-object.xml`), which illustrates the enabling of the feature. Please note that this can also be configured using the administration console user interface of the JBoss server.

```xml
<deployment>
  <if-exists>keep</if-exists>
  <context>
    <context-name>dashboard</context-name>
    <properties>
      ...
      <!-- Set the dnd property -->
      <property>
        <name>theme.dyna.dnd_enabled</name>
        <value>true</value>
      </property>
      ...
    </properties>
  </context>
</deployment>
```

The `<name>theme.dyna.dnd_enabled</name>` value enables or disables the drag-and-drop behavior.
Partial content refresh

One of the most expensive processes in a portal is the refresh of portlets when the page is generated. For every user action on a page, the portal calls all of the portlet methods in a serial, but non-specific order, which involves a significant amount of time and server-side processing. Partial content refresh support mitigates these issues to a large extent with an effective use of client-server asynchronous communication. When the state of a single portlet changes, a partial content refresh facilitates the update and refresh of only that portlet, instead of all of the portlets on the page. This prevents the regeneration of the whole page and the initialization of all of the portlets on the page.

The following image illustrates the partial content refresh flow:

The partial refresh capability is compatible with the JSR-168 portlet API, which allows for programmatic update of portlet states during runtime.

Partial refreshes can be enabled through portal object configuration or through configuration at the default server level.
Portal object configuration

To implement partial page refreshes successfully, the properties of the portal objects need to be updated to include the property, `theme.dyna.partial_refresh_enabled`, which takes the values `true` or `false`. The deployment descriptor appears as follows:

```xml
<deployment>
  <if-exists>keep</if-exists>
  <context>
    <context-name>dashboard</context-name>
    <properties>
      ...
      <!-- Set the dnd property -->
      <property>
        <name>theme.dyna.partial_refresh_enabled</name>
        <value>true</value>
      </property>
      ...
    </properties>
  </context>
</deployment>
```

For changing the behavior during runtime, the property editor of the management portal can be used to set the values at the default server level. Once set, all of the pages in the portal automatically inherit the property. The default portal configuration is shown in the following screenshot:

![Properties](image)

Portlet configuration

Although the default portal configuration enables partial refreshing for all portlets, it is also important to understand when a portlet would actually use a partial refresh. Having this knowledge helps us to control the behavior at the portlet level.
The server, by default, enables partial refresh for action and render links. However, there are the following exceptions for which the whole page is refreshed:

- Forms with GET requests. Using a GET request is discouraged by the portal specification. Hence, this should not be of significant concern.
- Form uploads.
- A window in a MAXIMIZED state. Partial page refreshes will not function when the window is entering or exiting a MAXIMIZED windows state.

There are, however, a few scenarios where we don't want to use a partial refresh at all. In such cases, partial refreshes can be disabled by using the following configuration in the portlet descriptor file, jboss-portlet.xml.

```xml
<portlet>
    <portlet-name>TestPortletBlockPartialRef</portlet-name>
    <ajax>
        <partial-refresh>false</partial-refresh>
    </ajax>
</portlet>
```

Setting the `<partial-refresh>` value to `false` prevents partial refreshes of portlets when they are invoked.

**Constraints in implementing partial refresh**

Although partial refresh is a powerful option, it has certain constraints on both the client and server sides.

**Inconsistent session attributes**

When a page is partially refreshed, the state of the page and its relationship to the portlets on the page become inconsistent. The direct impact of this is the session variables. If a partially-refreshed variable updates a session variable, a different portlet using this same variable will continue to use old data and might not be able to update to the latest session data.

In such cases, selective enabling of partial refreshes for portlets should address the problem.

**Non-AJAX interaction**

The AJAX implementation for partial refreshes is triggered by DOM events generated by the user in the browser. However, the user doesn't trigger these events. The portal JavaScript has no way of knowing the request, and hence might refresh the whole page.
A good example of this is the direct programmatic submission of forms, instead of relying on browser events for submission.

For issues like these, care should be taken when coding the portlets, and using constructs that do not generate DOM events should be avoided.

**Considerations for AJAX implementations**

Although AJAX is an exciting technology and provides significant advantages in terms of performance, usability, and implementation, there are certain scenarios where using AJAX is not a good fit. This is especially true when a lot of custom development is involved in using AJAX libraries.

Some of these are discussed in the following sections.

**Global variables**

One of the important aspects of portal-specific development is the obvious collision of variables declared at the global page level. Due to the nature of portals, a page typically consists of multiple portlets with their own local variables. Creating global variables almost invariably exposes the application to conflicts with local variables.

One solution might be to assign name spaces along with the variable declaration.

**State management**

With constant partial page refreshes happening, and content being refreshed through AJAX implementation, it is very difficult to keep the state of the page consistent. Page refreshes and content refreshes can occur at any given time, triggered by any portlet on a page. Hence, building a portlet with dependencies on a page can cause serious problems. Similar issues can crop up if a user uses the Back button on the browser.

All calls need to be atomic and should use techniques such as cookies to store states.
Visual cues

Users of web applications prefer to continuously receive the status on their actions, without which they tend to get impatient or nervous. Most AJAX functions and calls execute in the background with no obvious sign in the browser that the user needs to wait and that the browser is busy. Portals with multiple components on pages sometimes operate independently, further exacerbating the issue.

Hence, AJAX implementation should always use techniques that provide continuous visual cues to the user.

Summary

AJAX is a powerful tool. Used appropriately, it can provide tremendous value in terms of features, usability, performance, and implementation. The implementation is relatively native and simple with no need for special skills or plug-ins, other than the ones that web developers normally have. With the adoption of the Portlet 2.0 specification, AJAX development has become significantly easier and more reliable. Our example in this chapter showed how easy it is to develop and deploy AJAX-based portlets.

JBoss portal supplements custom development of portlets enriched with AJAX, with its own set of pre-packaged baked-in support for features such as AJAX-enabled layouts, drag-and-drops, partial refreshes, and so on. Portlets can be configured individually to provide or block AJAX features. Although it is powerful, it also comes with certain limitations. Developers can build a more robust portal solution using AJAX if they bear in mind these limitations.

In the next chapter, we will discuss how portlets can leverage persistent data stores and databases to build enterprise portal applications.
Where to buy this book

You can buy JBoss Portal Server Development from the Packt Publishing website:

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.