Oracle Application Server Portal

Technical Note

BUILDING HIGHLY INTERACTIVE PORTLETS WITH AJAX

July 2006

ORACLE APPLICATION SERVER 10g

INTRODUCTION

In today's fiercely competitive Web space, creating a rich user interface for your Web application is no longer an option; it's a requirement. Wherever you look on the Web, you see highly interactive user interfaces (UI). Photo Web sites present drag and drop upload capabilities, visual editing, as well as interactive order placement. Search sites offer popular search results as you type. Mapping applications provide smooth scrolling and allow you to overlay satellite photos and street maps.

What is the technology behind these highly interactive Web sites? How can you apply this technology to portals? How natural is it for portlets to provide rich interaction? Can you improve usability of your portlets in the same way as Web applications?

This paper introduces you to the underlying technology used by today's Web applications and portlets: Ajax (Asynchronous JavaScript and XML). It then shows how you can use Ajax with your portlets to provide a highly responsive end user experience. This paper also illustrates how you can use Ajax to contextually wire portlets, that is, pass information from one portlet to the other.

The samples in this paper use the standards-based portlet API (JSR 168), but can easily be applied to portlets built with any Web technology. We assume that you have a basic understanding of Java, JSR 168 portlets, and JavaScript.

AJAX: THE TECHNOLOGY BEHIND THE SCENES

Today's rich Web applications use a mix of JavaScript and asynchronous communication with the application server. This mechanism is also known as Ajax: Asynchronous JavaScript and XML. The intent of Ajax is to exchange small pieces of data between the browser and the application server, and in doing so, use partial page refresh instead of reloading the entire Web page.

Although Ajax does not have a strictly defined set of components, the following are the key Ajax features:

- *JavaScript:* The scripting language of the browser.
    - To detect user interactions, such as mouse click, mouse movements, and typing.
    - To dynamically modify the page by manipulating the Document Object Model (DOM) tree.
- *XMLHttpRequest object:* The means provided by browsers to submit HTTP requests to the server and handle the HTTP response, without a full-page refresh.
- *XML:* The format of the data exchanged between the client and the server. There is no restriction as to the data format used, though the format is usually HTML.

HOW DOES AJAX WORK?

A traditional Web application submits the HTTP request to the Web or the application server from the browser. An action, such as a mouse click on a hyperlink or a form submit, initiates the request. The server generates the HTTP response, and returns the requested page.

Ajax Web applications add a bit of seasoning to this flow. Similar to a classic Web application, the application submits a request as a result of a user interaction. Instead of refreshing the entire page, Ajax refreshes only part of the page.

An Ajax-driven application submits the HTTP request asynchronously. So, instead of waiting for the entire page to load, the end user can continue interacting with the page.

The Ajax engine is the key player in the Ajax model. The Ajax engine is a piece of code that is typically implemented in JavaScript, is downloaded along with the HTML page, and runs in the browser. The Ajax engine has multiple responsibilities (Figure 1):

1. *Detecting user interactions.* The engine detects and reacts to user interactions as they take place. For example, if the user hovers the mouse over a specific area, the Ajax engine recognizes the action and triggers an HTTP request.
2. *Submitting HTTP request to the server.* When the pre-defined user interaction takes place, the Ajax engine submits a request to the Web server asynchronously.
3. *Handling HTTP response returned by the server.* The engine handles markup returned by the Web or application server. For example, if the response is XML, the Ajax engine applies the XSL style sheet to it.
4. *Performing partial page refresh.* The engine makes the necessary changes to the Document Object Model (DOM), which is the internal representation of the HTML document, thus updating the rendered page. For example, the engine can display a new layer of HTML containing the data returned from the server.

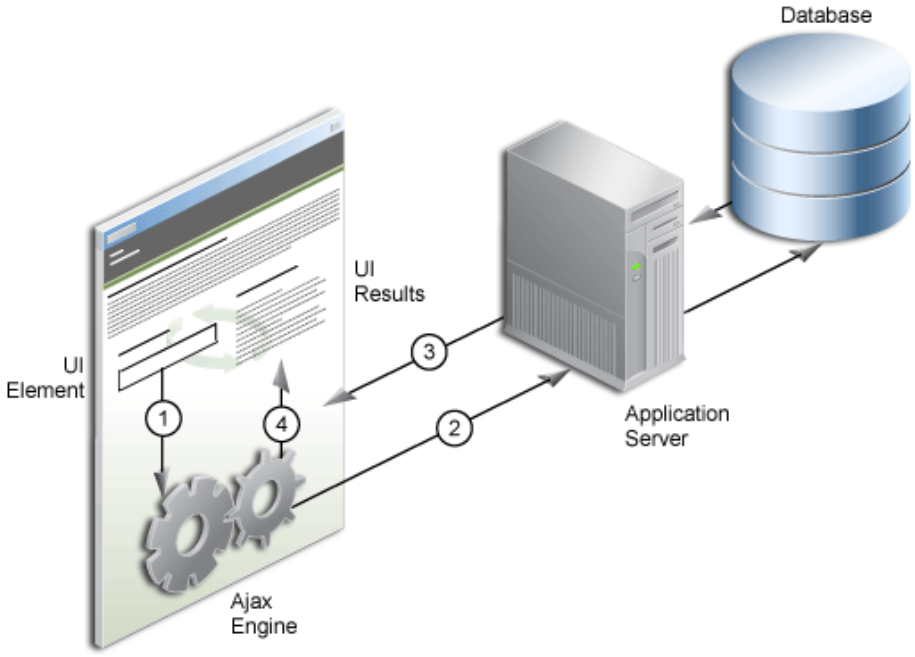

Figure 1 – Ajax Web Application Model

AJAX IN THE PORTLET WORLD

How does this model map to the world of portlets?

The Web application implementation model can be applied in the same way to portlets. Just as in classic Web applications, the key player here is the Ajax engine. Depending on the implementation, the Ajax engine can be part of the portal page containing the portlet, but often it is implemented by the portlet itself.

When the Ajax engine is part of the portlet, the portlet developer and/or the page designer need to resolve several questions, such as:

- Do the JavaScript libraries collide if multiple Ajax portlets reside on the same page?

- What happens if multiple instances of the same portlet are added to the page?

In this portlet use case, the Ajax engine has very similar responsibilities to the Web application model described in Figure 1:

1. *Detecting user interactions*. As the user interacts with the portlet UI, the Ajax engine detects each action you define. For example, if the user types a character in a text field, the Ajax engine tracks it and executes the defined action. The value entered is then used as a parameter of the XMLHttpRequest object.
2. *Submitting HTTP request to the server*. The Ajax engine creates and submits the request to the portlet application.
3. *Handling HTTP response returned by the server*. When the portlet application returns the markup, the Ajax engine must handle it. For example, if the response is XML, the Ajax engine must apply an XSL style sheet to it.
4. *Performing partial page refresh*. To avoid full-page refresh, the markup must be injected into the DOM to facilitate a partial page refresh. For example, the result set returned by the portlet is displayed right below the text field.

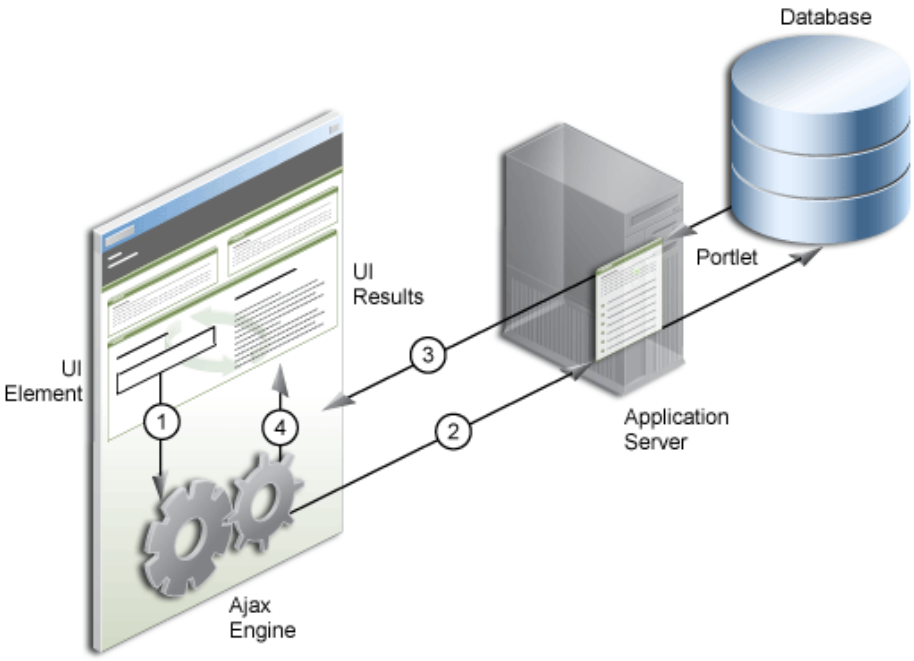

Figure 2 - Ajax Portlet Model

USE CASE: INTERACTIVE PHONE BOOK PORTLET

Suppose you are required to create a new version of a phone book portlet, where end users can enter either a full name or part of a name, and the portlet returns all matching names. The current version of your portlet contains a text field and a Submit button. To submit the name entered, the end user needs to click the Submit button. After the entire page refreshes, the portlet displays the names matching the end user's query. If the end user misspells the name, he or she must change the entry, and then click Submit again.

Using the Ajax portlet model, the new version of the portlet contains just a text field. As the user types in the name, the results immediately display in the portlet. The portlet initiates a roundtrip to the back end asynchronously, passing the string entered in the text box as a parameter. Upon return of the markup, the portlet displays the results by

injecting it into the DOM tree of the page. As the user continues to type, the result set narrows down according to the entered text.

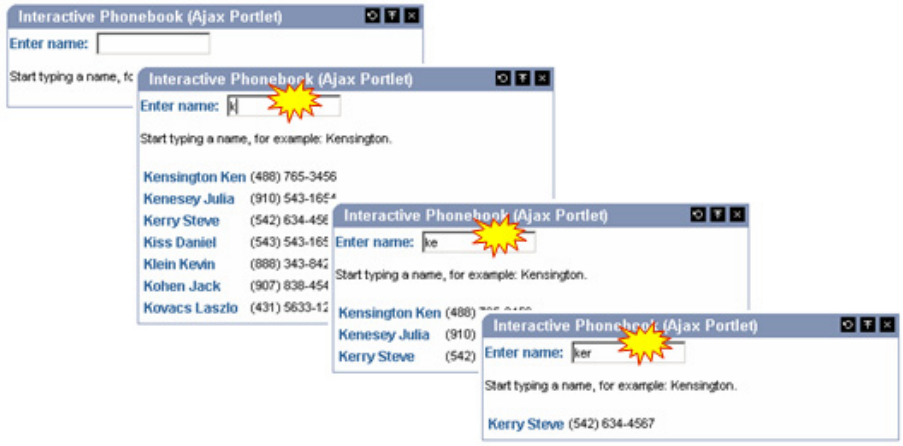

Figure 3 – Contextually Wired Ajax Portlets

BUILDING THE AJAX PHONE BOOK PORTLET

What does it take to build the interactive Ajax phone book portlet? Assuming you used JSP to generate your portlet's markup, the following steps show you how to build the portlet described in the previous section.

1. Create the input field by using the following code

```
<form name="sampleForm" id="sampleForm">
  <input class="portlet-form-input-field"
         type="text"
         size="14"
         name="custName"
         id="custName"
         onKeyUp="assembleURL(event);"
  />
  . . .
</form>
. . .
<div id="results"/>
```

2. Catch the typing event. The `onKeyUp` JavaScript event is triggered when the user hits a key on the keyboard. This part plays the "user interaction monitoring" role of the Ajax engine. Also notice the `<div>` tag, which is where the returned markup will be injected.

3. Construct a resource URL to the portlet application. The portlet application contains a servlet, `GetCustomerInfo`, which performs the lookup operation. The `encodeURL()` method creates the resource URL to the servlet.
   Note: If the portlet is accessed through Web Services for Remote Portlets (WSRP), the resource URL ensures that the request for the `GetCustomerInfo` servlet is proxied through the application server middle tier to the portlet.

```
<script language="JavaScript"
        type="text/JavaScript">
```

```
. . .
var url =
 '<%=response.encodeURL(request.getContextPath()
+
 "/GetCustomerInfo")%>';
. . .
```

4. Create the `XMLHttpRequest` object. Creating the `XMLHttpRequest` object is a single line of JavaScript code:

```
request = new XMLHttpRequest();
```

While the above code looks simple, you must write it in a significantly more complex way to accommodate browser incompatibilities:

```
var request = false;
   try {
     request = new XMLHttpRequest();
   } catch (trymicrosoft) {
     try {
       request = new
             ActiveXObject("Msxml2.XMLHTTP");
     } catch (othermicrosoft) {
       try {
         request = new
             ActiveXObject("Microsoft.XMLHTTP");
       } catch (failed) {
         request = false;
       }
     }
   }
```

5. Submit the `XMLHttpRequest`. Define the submit method: POST or GET, then specify the JavaScript function, `updatePage()`, that executes after the engine returns the response. Finally, you submit the request.

```
function getCustomerInfo(url) {
     request.open("post", url, true);
     request.onreadystatechange = updatePage;
     request.setRequestHeader("Content-Type",
     "application/x-www-form-urlencoded;
        charset=UTF-8");
     request.send
     (createQuery(document.getElementById
                          ("sampleForm")
                 )
     );
 }
```

Note: When the form is submitted, the value in the text box is passed to the servlet as a parameter. The name of the parameter is the name of the field: `custName`.

6. Populate the `<div>` tag with the response. To do this, inject the returned markup in the DOM tree, where you want to see the results. The following code performs the injection:

```
function updatePage() {
  document.getElementById("results").innerHTML =
    response[0].replace(/\n/g, "");
}

. . .

<div id="results">
```

The code above has been simplified to highlight the important function of injecting the HTML. You should add logic to handle error checking and reading the results, as in the following example:

```
function updatePage() {
  if (request.readyState == 4)
    if (request.status == 200) {
      var response =
        request.responseText.split("|");
          document.getElementById
            ("results").innerHTML=
           response[0].replace(/\n/g, "");
    }
    else if (request.status == 404)
      . . . // Request URL does not exist
    else
      . . . // Error
  }

. . .

<div id="results">
```

In addition to building the client side of the code, you must also build a server-side component, which serves the asynchronous requests. In this example, the component is a servlet. The servlet takes a String parameter, which is the contents of the input field the user has entered. For example, if the user types the letter "K," the application executes the servlet with the parameter containing the letter "K":

`http://<host>:<port>/<contextRoot>/servlet/GetCustomerInfo?`**`custName=K`**

The result returned by the servlet:

```
Kenesey Julia    (910) 543-1654
Kensington Ken   (488) 765-3456
Kerry Steve      (542) 634-4567
Kiss Daniel      (543) 543-1654
Klein Kevin      (888) 343-8425
Kohen Jack       (907) 838-4545
Kovacs Laszlo    (431) 563-1212
```

As the user continues to type, the list is progressively narrowed down:

`http://<host>:<port>/<contextRoot>/servlet/GetCustomerInfo?`**`custName=Ke`**

```
Kenesey Julia    (910) 543-1654
Kensington Ken   (488) 765-3456
Kerry Steve      (542) 634-4567
```

When the user deletes a letter from the text field, the remaining contents of the field is submitted to the servlet, which in turn updates the result set.

INTER-PORTLET COMMUNICATION USING AJAX

Although the JSR 168 specification does not address inter-portlet communication, there are well known and documented workarounds for the wiring portlets together. For example, you can use the portlet application's session context or vendor-specific extensions to implement contextual portlet wiring.

Traditional workarounds, however, lack rich interaction. For example, when information is passed from portlet to portlet, the entire portal page is refreshed. With Ajax, you can implement inter-portlet communication while maintaining partial page refresh. Let's revisit the phone book portlet example, where you can improve the functionality by separating the search results from the form. This proposed modification allows the user to see the name he has entered in one portlet, while simultaneously viewing the results in another portlet.

1. Build the phone book portlet by performing steps 1-5 as described in the previous section (Portlet A). Next, you create a second portlet.
2. *Portlet B:* Populate the `<div>` tag with the response in the newly created portlet. Make sure that the `updatePage()` JavaScript function injects the returned markup in the DOM of the Search Results portlet.
   Portlet A

```
function updatePage() {
  document.getElementById("results").innerHTML =
      response[0].replace(/\n/g, "");
}
```

   Portlet B

```
<div id="results">
```

   Because the `<div>` tag was moved to Portlet B, the results are displayed in Portlet B.

Portlet A **Portlet B**

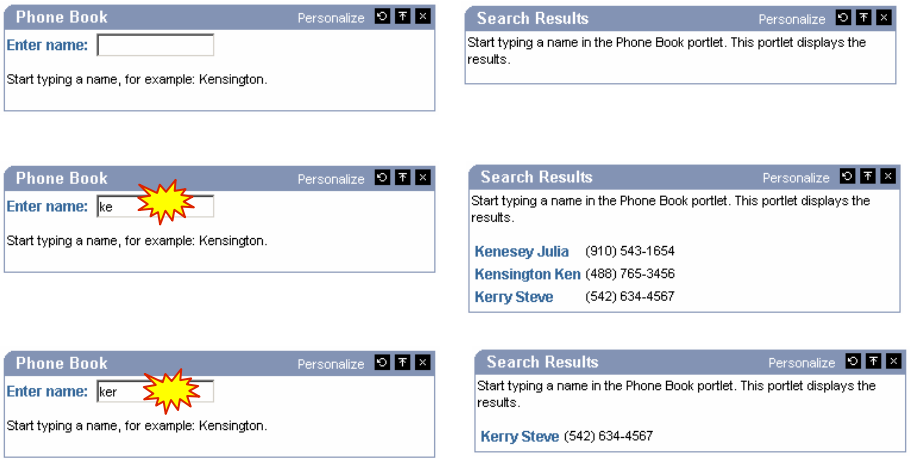

Figure 4 – Contextually Wired Ajax Portlets

WIRING PORTLETS AT RUNTIME

Traditional inter-portlet communication workarounds for JSR 168 portlets require the portlet developer to perform the wiring while developing the portlets. Although hard coding the target portlet that displays the search results may work in some cases, very often you need to perform the wiring at runtime.

To implement runtime wiring, the page designer must enable the portlets to "talk" to each other (specifically, wiring them together). The page designer can achieve this in several ways. In the simple phone book scenario, you add personalization capabilities to the portlets and store the personalization information that links the portlets together in each portlet's preference store.

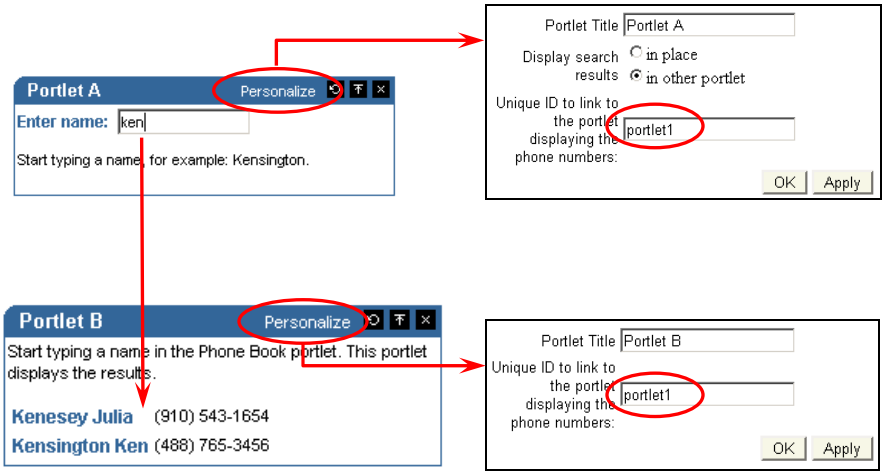

Figure 5 – Wiring Portlets Through Personalization

To support this use case, both portlets must read their own preference store as they render their markup. The first uses the value to address the ID of the `<div>` tag in the other portlet. If the personalization values do not match in the two portlets, the results returned by the asynchronous request will not display.

The portlets use the `getValue()` method of the PortletPreferences class to read the preference store. The value read is then concatenated to a fixed string, `detailAddress`.

```
Portlet A
 function updatePage() {
   document.getElementById
  ("detailAddress<%=prefs.getValue("linkID", "")%>").
     innerHTML = response[0].replace(/\n/g, "");
   }

Portlet B
<div id=
  'detailAddress<%=prefs.getValue("linkID", "")%>'>
</div>
```

The steps outlined above not only allow you to implement visually inter-connected portlets, but you can also perform the wiring at runtime.

OPEN ISSUES

While this paper has addressed a couple of typical use cases related to Ajax-enabled portlets, there are numerous scenarios that have not been covered.

Quite often, as a developer you require the portlets to maintain their own state. As the user navigates away from the page and returns, the portlet should be able to restore its state, including its asynchronous state. In our example, the text field should contain the last entered query criteria, and the results should display, even when a user clicks on another page and returns to the page containing the portlets.

A second open issue includes the possibility of conflicts between JavaScript variables and methods that may be caused by having more than one instance of the Ajax portlets on the same page. Ideally, you should reference a JavaScript library once in the portal page, and prefix the variable and method names with the portlet's unique ID.

Under some circumstances, you may also want to wire together more than two portlets. Although not covered in this paper, you can develop a fairly simple framework that allows portlets to register with other portlets and "listen" to their events. This way, portlets can refresh their content based on the events taking place in other portlets.

You also need to be aware that the technique outlined in the paper needs to be modified if the consumer renders the portlets in iframes. Since many portals, including Oracle Portal, don't use iframes, the approach described in this paper works fine in those environments.

In the long run, these issues need to be addressed by the next version of the portlet standards, JSR 286.

SUMMARY

This white paper discussed how easily you can bring the improved usability of today's Web applications to the world of portlets; it also stepped you through creating a sample standards-based Ajax portlet using the JSR 168 portlet API.

Although inter-portlet communication between JSR 168 portlets can be achieved with well-known workarounds, you can improve inter-portlet communication using Ajax. In addition to describing a rich interaction model, this paper also presented how portlets can be wired together at runtime.

Building Highly Interactive Portlets with Ajax
July 2006
Author: Peter Moskovits

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com