John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
December 2010

DISCLAIMER

This document accompanies one of a series of presentations on extending your OpenEdge application by creating a rich browser-based user interface. Other sessions in this series introduce specific control libraries that provide a rich user experience, based on AJAX. So in this session I talk a little about what is going on under the covers when you use a framework or library that uses AJAX.

Perhaps the trickiest thing to understand is that AJAX isn't a language or a new technology, but really more of a technique for combining other existing technologies to achieve a very significant goal. What the word stands for tells us something about what makes it up. The **A** is for **asynchronous**, meaning that you can send requests to the server without blocking until a response comes back. This is a key aspect of an AJAX client, but it's not absolutely required. You can have synchronous AJAX requests, as I'll show you a little later.

The **J** is for **JavaScript**. This is the key element of an AJAX user interface definition. You can have JavaScript embedded in an HTML page in a browser, or a page can reference external JavaScript files. Either way, it's the essential programming element. The second **A** is just for **and**, to make the acronym pronounceable.

The final **X** is for **XML**, as a data representation. This is not the the only choice either. You can create AJAX front ends that receive and parse data as JSON, JavaScript Object Notation, or as HTML, or in other formats. I'll use XML for the example in this session, but if you look at the ExtJS sessions in this series, for example, you'll see that that support library uses JSON. The key value of AJAX is that it runs in the browser, but it makes the UI you build to run in the browser act more like a desktop application.

So what are the key technologies used in an AJAX client?

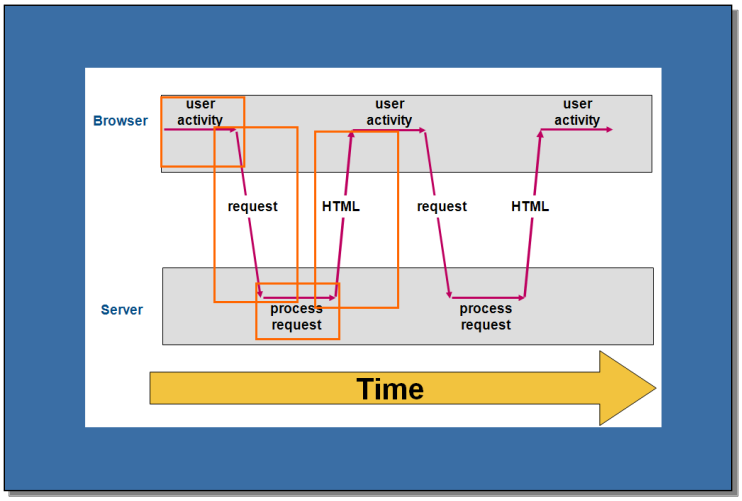
First, as mentioned, there's JavaScript, which can be embedded in the HTML for a page, or coded in separate script files.

Then there's CSS, the Cascading Style Sheet language that lets you format the UI details however you like.

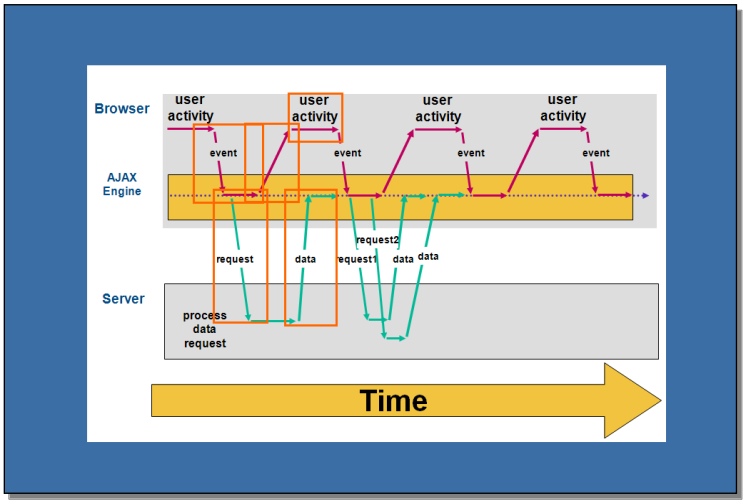
The key element for parsing retrieved data is the DOM, the Document Object Model, which lets you in effect walk the widget tree of a response containing data to locate data elements as you need them for display or client-side UI logic.

Finally, the one unique element is an object with the name **XMLHttpRequest**. This is the object that the browser needs to support in order for an AJAX client to function.

So what makes an AJAX client different from the traditional interaction between a browser client and server? In a typical HTML client, the user fills in values on an HTML form, submits a request to the server, and then blocks waiting for a response. That's the synchronous part. Then the response comes back in the form of a complete replacement for the HTML page, which is a lot of overhead and can make the user interface experience very clunky and visually broken up:



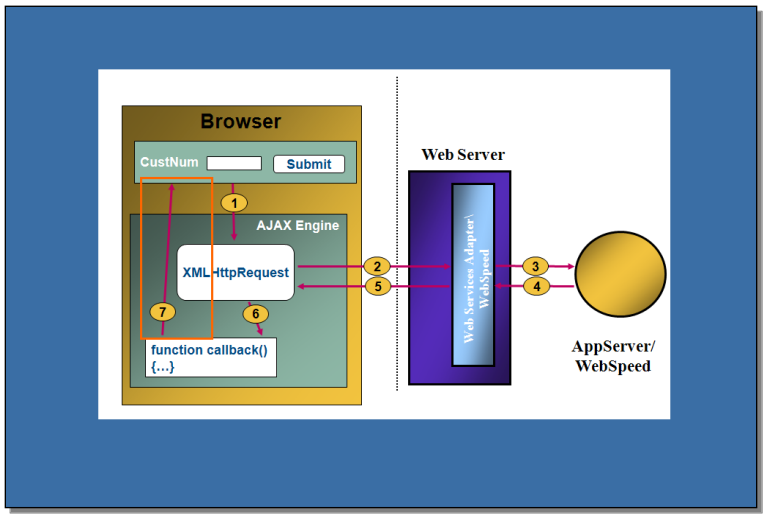
By contrast, an AJAX front end has two key advantages. When the UI needs to interact with the server, it sends an event to the part of the client that represents the AJAX engine, basically the support for the XMLHttpRequest object. This sends a request off to the server, but control is immediately returned to the UI, so that the user can keep working for the typically short period until a response comes back. That's the asynchronous part. Then when the response comes back, it is not a full replacement for the HTML page, but simply the data needed to fill in the right spots in the form. This reduces network traffic, and provides a more seamless, natural flow to the user experience.



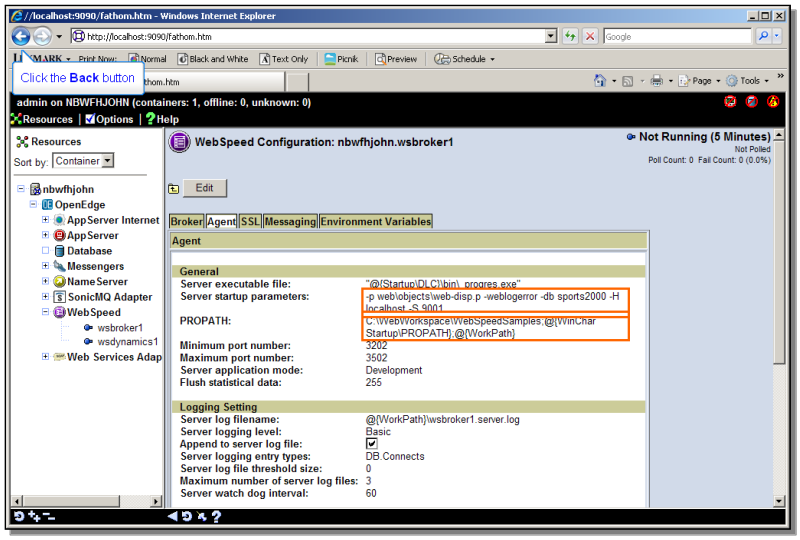
Take a look at what happens when the client needs data from the server. The UI code in the browser sends a request via the XMLHttpRequest object. It's aptly named, because its job is to send requests to the server via HTTP, although not always using XML.

The XMLHttpObject then routes the request to the Web server. This is basically the same step whether the Web server is supporting WebSpeed to run OpenEdge procedures, or using proxies to run .p's as web services. OpenEdge supports both. In either case, the Web services adapter or WebSpeed messenger routes the request to an ABL procedure, and the response data comes back -- in the form of output parameters in the case of a Web service request, or content posted to the webstream in the case of WebSpeed.

The data is then sent back to the request object on the client. The JavaScript code in the client must define a callback function to execute when a response comes back. This function interacts with the UI to display returned data values or make whatever changes are needed in the UI based on the response. That's basically the complete cycle, as illustrated here:



Take a look at a concrete example to get a better idea of what's involved. The example uses WebSpeed to communicate with ABL procedures on the server, supported by the Apache HTTP server. In OpenEdge Explorer, you can quickly review a few settings that need to be correct. Drilling down to the WebSpeed group, you can look at wsbroker1, and its configuration information, in particular a couple of agent properties that need to be set right. The example accesses the OpenEdge sports2000 database, so you need to establish a connection to the database server. In addition, the ProPath needs to hold the directory where the OpenEdge Architect project stores the procedures that get run on the server.



Once that's all checked out, you can start the WebSpeed broker.

Take a look at the ABL procedure that's going to be run on the server when the client needs data. It's a variation on a simple procedure used in the example for the ExtJS controls library. That library uses JSON as its standard data exchange format, but you can do the same thing just as easily with XML. Here is a description of what the ABL statements are doing.

```
/* getcustomers_param.p */

{src/web2/wrap-cgi.i}

OUTPUT-CONTENT-TYPE ("text/xml").

DEFINE TEMP-TABLE ttCust LIKE Sports2000.Customer.

TEMP-TABLE ttcust:SERIALIZE-NAME = "customers".

FIND Sports2000.Customer WHERE Customer.CustNum =
    INTEGER(get-value("piCustNum")).
CREATE ttCust.
BUFFER-COPY Customer TO ttCust.

TEMP-TABLE ttCust:WRITE-XML("stream", "webstream", true).
```

The **wrap-cgi** include file provides hooks to the WebSpeed support the procedure needs, and the next statement confirms that the output format to the web stream is going to be XML.

The procedure defines a temp-table and sets its **SERIALIZE-NAME**, which is in effect the header for the returned XML data, to **customers**.

Then it takes advantage of a standard WebSpeed function, **get-value**, which extracts an input parameter by name from the URL passed from the client, and uses that to find a specific customer.

It copies that one row to the temp-table, and the built-in ABL **WRITE-XML** function converts the temp-table data to XML and sends it back over the webstream.

That simple procedure represents all the business logic that would be on the backend of the application. The other half is defined in the OpenEdge Architect WebSamples project. The example is an HTML file with some embedded Javascript, which lets the user enter a customer number, pass it to the Web server, and get some customer data back for display.

Take a look at some of the elements of this example. At the top is a script tag identifying the embedded JavaScript.

```
<!--
Program AjaxCustWebParam.html
Uses WebSpeed to run getcustomers_param.p,
  passing in the selected customer number.
-->

<html>
<head>
<script type="text/javascript">
```

At the end of the file, there is a simple data entry form, defined in HTML, that defines a CustNum field with a label, and says that on change of the value in that field, it runs the function **LoadXMLdoc**.

```
<!-- This is the little data entry form where CustNum is captured: -->
<form action="javascript:void(0)">
<fieldset style="width:600px">
<legend>Please Select a Customer</legend>
<label for="custnum">Customer Number:</label>
<input type="text" name="custnum" id="custnum"
        onchange="loadXMLDoc()"  />
</fieldset>
</form>
<pre id="custinfo" style="width:400px"></pre>

</html>
```

There's also a section defined using the **pre** tag (for pre-formatted data) called **custinfo** where the data that comes back gets formatted and displayed.

Looking back at the remainder of the file, here's the definition for the JavaScript function **loadXMLdoc**, which gets run when the user enters a customer number.

```
/* This is run on leave of the custnum field in the form: */
function loadXMLDoc()
{
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
```

The first thing the function has to do is to create an instance of the **XMLHttpRequest** object. This reveals the first interesting wrinkle in the programming required when you're using AJAX without the support of one of the control libraries that are built on it. Not all browser versions support the object in the same way. AJAX has been around long enough now that pretty much all modern browsers support it, but at it happens, Microsoft did not support the object until Internet Explorer version 7. So this code checks to see if the object is defined by the browser, and if it is, creates an instance of it in the usual way. For older versions of IE, it instead creates it as an ActiveX control, which is how Microsoft supported the behavior before IE7. This is a good example of one of the key services that any controls library built on AJAX provide: allowing for various differences in browsers that would give you major headaches if you tried to write all the supporting code yourself.

The next code fragment shows an essential feature of the XMLHttpRequest object. The object defines an **onreadystatechange** property. When a request is passed through the object, there are several intermediate states that it goes through, numbered 0 through 3. You don't need to worry about these, and in fact some browsers don't even report all of them, but the important thing is that when the response from the server is complete and is ready to process, the state gets set to 4, so the function checks for that. The second key property here is the **status**. Basically there are just two status values you care about: 200 means success, and the familiar HTML error 404 means failure. If both of these properties have the appropriate value, the function proceeds with its work.

```
xmlhttp.onreadystatechange=function()
  {
    if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
```

Then the code looks at the response that comes back as an XML document, and saves that off as a string called **custXML**.

```
        /* This is the XML document that comes back
          through the webstream: */
        var custXML = xmlhttp.responseXML.documentElement;
```

Next the code constructs another string that formats the customer number, name and address that get returned in the XML response into a form with labels that we can then display in the HTML:

```
var custRec =
                '<table>' +
                        '<tr>' +
```

```
                    '<th align="left">Customer Number:  </th>' +
                    '<td>' +
                        document.getElementById( "custnum" ).value +
                    '</td>' +
            '</tr>' +
            '<tr>' +
            '<th align="left">Customer Name:</th>' +
            '<td>' +
            custXML.getElementsByTagName("Name")[0].
                                    firstChild.nodeValue +
            '</td>' +
            '</tr>'+
            '<tr>' +
            '<th align="left">Address:</th>' +
            '<td>'  +
            custXML.getElementsByTagName("Address")[0].
                                    firstChild.nodeValue +
            '</td>' +
            '</tr>';
```

Note the use of the DOM function **getElementById**, an example of the role the DOM plays in helping you parse the response that comes back as a single complex document from the server.

As the final step in processing the response, the **custinfo** slot in the display is filled in with this formatted information showing the three customer fields being displayed.

```
        document.getElementById( "custinfo" ).innerHTML = custRec;
```

That's the end of what happens when the readystate is 4 and the status of 200 indicates success. The next statement in the loadXMLdoc function is this one, which is executed when the user first enters a custnum value and tabs out of that field in the form.

```
/* This captures the custnum entered and submits it to WebSpeed: */
var custval = document.getElementById("custnum").value;
```

The **custnum** value – which was defined in this bit of the code we looked at earlier -- is retrieved from the form. Then comes the key step in using the XMLHttpRequest object, the **open** method, which sets up the HTTP request.

```
xmlhttp.open("GET",
  "cgi-bin/cgiip.exe/WService=wsbroker1/getcustomers_param.p?piCustNum="
   + custval, true);
xmlhttp.send();
```

Open takes three parameters. The first can be GET or POST, and for now I'll just say that **GET** is sufficient for most cases. The second is the URL of the service you want to run on the backend. In this case, using WebSpeed, it's the location of the WebSpeed broker relative to the cgi-bin directory for the Apache Web server, plus the name of the ABL procedure relative to WebSpeed's ProPath, plus in this case the

parameter being passed in, named **piCustNum**, attached to the **custval** value retrieved from the input form.

Now piCustNum is a little misleading as a name for the value, first because it's not actually passed as an INPUT parameter to the ABL procedure, and second, because as part of the URL, it's just passed as a string, not an integer. As a reminder, here's how that value is handled in the ABL procedure on the backend. The WebSpeed **get-value** function retrieves it from the URL, and the ABL **INTEGER** function converts it from a string to an integer to use in the **FIND** statement.

The third parameter to the **open** method is true if you want the request to be asynchronous, and false otherwise. Normally you'll set this to true to take advantage of that feature of AJAX. Once the request is set up, the **send** method makes the HTTP request. Then the embedded function that acts as the event handler for the state change event waits for the state to go to 4, to process the response. And that's the end of loadXMLdoc.

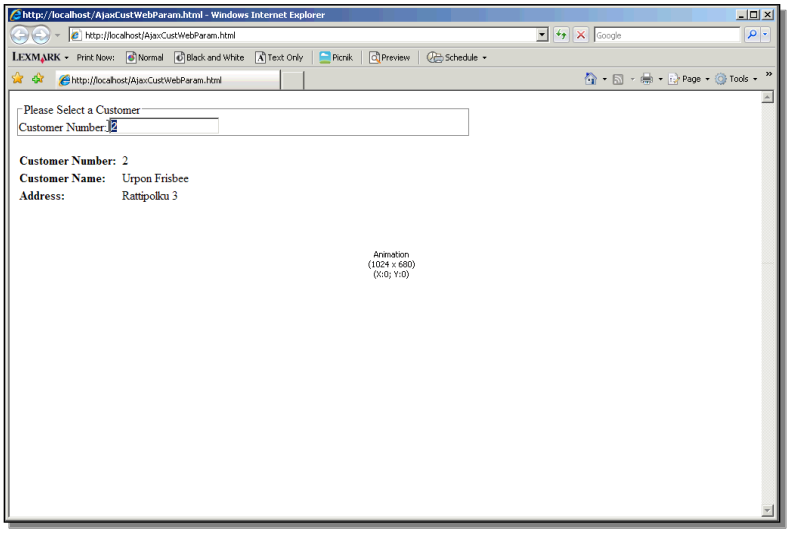
For clarity, and in caser you want to copy and paste this code as the starting point for your own example, the entire HTML file is repeated here:

```
<!--
Program AjaxCustWebParam.html
Uses WebSpeed to run getcustomers_param.p,
  passing in the selected customer number.
-->

<html>
<head>
<script type="text/javascript">

/* This is run on leave of the custnum field in the form: */
function loadXMLDoc()
{
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
xmlhttp.onreadystatechange=function()
  {
    if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {

      /* This is the XML document that comes back
         through the webstream: */
       var custXML = xmlhttp.responseXML.documentElement;
       /* And this is a little formatted display of
          custnum, name, and address: */
       var custRec =
              '<table>' +
                     '<tr>' +
                            '<th align="left">Customer Number:  </th>' +
                            '<td>' +
                                document.getElementById( "custnum" ).value +
                            '</td>' +
                     '</tr>' +
                     '<tr>' +
                     '<th align="left">Customer Name:</th>' +
                     '<td>' +
                     custXML.getElementsByTagName("Name")[0].
                                              firstChild.nodeValue +
```

```
                        '</td>' +
                        '</tr>'+
                        '<tr>' +
                        '<th align="left">Address:</th>' +
                        '<td>'  +
                        custXML.getElementsByTagName("Address")[0].
                                                firstChild.nodeValue +
                        '</td>' +
                        '</tr>';


          document.getElementById( "custinfo" ).innerHTML = custRec;
    }
  }

/* This captures the custnum entered and submits it to WebSpeed: */
var custval = document.getElementById("custnum").value;

xmlhttp.open("GET",
  "cgi-bin/cgiip.exe/WService=wsbroker1/getcustomers_param.p?piCustNum="
   + custval, true);
xmlhttp.send();
}
</script>
</head>

<!-- This is the little data entry form where CustNum is captured: -->
<form action="javascript:void(0)">
<fieldset style="width:600px">
<legend>Please Select a Customer</legend>
<label for="custnum">Customer Number:</label>
<input type="text" name="custnum" id="custnum"
        onchange="loadXMLDoc()"  />
</fieldset>
</form>
<pre id="custinfo" style="width:400px"></pre>

</html>
```

Now that the coding is complete, let's see how all this works. If I enter the HTML file name as a URL, which the HTTP server can find relative to its htdocs directory, the little data entry form defined at the end of the HTML file comes up. I enter a customer number and tab out, which runs the loadXMLdoc JavaScript function.

Let's quickly review what happens now. The **loadXMLdoc** JavaScript function in the code creates an XMLHttpRequest object, opens it with the parameters we saw, and does the send, which initiates an HTTP request to the server. The WebSpeed messenger sends that on to a running OpenEdge instance, which runs the right ABL procedure, and sends the output back on the **webstream**. The request object picks that up and fires the callback function with a state of 4 and a status of 200. And the callback, which in my example is the unnamed event handler function embedded in loadXMLdoc, takes the response data, formats it, and puts it up in the HTML form in the browser. Here's the formatted output:



If I enter another customer number and tab out, the same thing happens again. But what comes back from the server is not the whole HTML page with new customer data embedded in it. It's only the customer data itself, so there's no distracting page refresh in the UI. The new data is just plugged into the page that was already there.

In conclusion, with AJAX, your user interface can interact with the server asynchronously, so that the user isn't always blocked waiting for a response. The UI can fill in data as it comes back without having to refresh the entire form. And you can code the client using HTML, supported by Cascading Style Sheets, to define the UI, plus JavaScript to define the client-side event handling logic, including the XMLHttpRequest object, and operations support by the DOM to parse the data that comes back from the server. On the OpenEdge side, you can code any business logic you need, using either WebSpeed or Web services as a way to run procedures and pass parameters in and data back out. Add to this the support for advanced UI controls provided by the third-party frameworks and libraries built on top of AJAX, and you have a powerful environment for defining a zero footprint, browser-independent user interface that rivals the behavior of a fat client desktop UI. That's the essential value of AJAX.