Ajax using XMLHttpRequest and Struts

Frank W. Zammetti

About five years ago I worked on a web app project where one of the primary requirements was that it should look, work and feel like a Windows-based fat client. Now, ignoring the question of why it was not in fact done as a Windows-based fat client in the first place, that can be a rather tough requirement in the world of web development, certainly it was five years ago when not very many examples of such a thing existed.

As a result of a number of proof-of-concepts, I stumbled upon some techniques for doing things that were at the time rather atypical approaches to web development. The end result was an application that, to this day, many people cannot even tell is web-based, aside from the fact that you access it with a browser!

Little did I know that only a few years later the basic concepts behind what I had done would re-emerge in the world as something called Ajax. Ajax is a term coined by the folks at Adaptive Path and is shorthand for Asynchronous Javascript + XML.

This just goes to show, patenting every idea you ever have is indeed a good way to riches! If only I would have thought what I did was anything special! But I digress…

Google is doing it. So are many others. But what is it? In a nutshell, the Ajax concept, which is not a concrete technology implementation but rather a way of thinking and a set of techniques that go along with the mindset, is concerned with the idea that rebuilding a web page for every user interaction is inefficient and should be avoided.

For instance, say you have a web page with two <select> elements on it. You want the contents of the second one to change when a selection is made in the first. Something like this is not uncommon and there are a number of ways you can handle it.

The Ajax approach to this is to only redraw a small portion of the web page, namely the contents of the second <select>.

The Ajax concept is based around something called the XMLHttpRequest component. All you Microsoft haters get ready to yell because this was a Microsoft creation! Yes, Microsoft got something right, and did so before anyone else! Microsoft first implemented the XMLHttpRequest object in Internet Explorer 5 for Windows as an ActiveX object (ok, so they didn't get it QUITE right!). The Mozilla project later implemented a native version with the release of Mozilla 1.0, and by extension, Netscape 7. Apple has now done the same as of Safari 1.2. Opera is now following suite with their version 7.60. All released versions of Firefox contain it as well.

Let's just cut to the chase and get to some code, shall we?

The XMLHttpRequest component, being a client-side component, must be instantiated via scripting before it can be user. Thankfully, the process is as simple as can be… For IE, use the following Javascript:

```
var req = new ActiveXObject("Microsoft.XMLHTTP");
```

…for any other browser, use:

```
var req = new XMLHttpRequest();
```

You will of course want to perform some branching logic in your code. There are a couple of ways to do it, but I prefer a simple approach, which to me is just an object existence check:

```
var req;
if (window.XMLHttpRequest) { // Non-IE browsers
  req = new XMLHttpRequest();
} else if (window.ActiveXObject) { // IE
  req = new ActiveXObject("Microsoft.XMLHTTP");
}
```

However you choose to do it, after this code executes you will find that the variable req is now a reference to an XMLHttpRequest object. This object has a number of properties and methods, which are summarized below:

Property	**Description**
onreadystatechange	Event handler for an event that fires at every state change
readyState	Status: 0 = uninitialized 1 = loading 2 = loaded 3 = interactive 4 = complete
responseText	Data returned from server in string form
responseXML	DOM-compatible document object of data returned
status	HTTP status code (i.e., 200, 404, 500, etc.)
statusText	String message associated with the status code

Method	**Description**
abort()	Stops the current request
getAllResponseHeaders()	Returns all headers (name and value) as a string
getResponseHeader("<headerName>")	Returns the value of the specified header
open("<method>", "URL"[, asyncFlag[, "<username>"[, "<password>"]]])	Opens a connection and retrieves response from the specified URL. Can also specify optional values method (GET/POST), username and password for secured sites
send(content)	Transmits request (can include postable string or DOM object data)
setRequestHeader ("<name>", "<value>")	Assigns values to the specifed header

Before we go any further I highly suggest checking out the webapp referenced at the end of this article. If you haven't already done so, download the sample webapp at the URL referenced at the end of this article. Install it into your servlet container of choice. It is supplied in exploded format, so just copying the directory created when unzipping it should work. For instance, if you are using Tomcat, simply copy the **xhrstruts** directory into <TOMCAT_ROOT>\webapps. That should be all you need to do. Start up your app server and give it a shot! Access the application using http://localhost:8080/xhrstruts (replacing 8080 with whatever port your app server is listening on). This sample shows a number of different usage situations including a sortable table, a dropdown changing the contents of another as described above, and an RSS feed parser. This example is Struts-based, as the title of this article implies! Although Ajax can be used completely independent of Struts, or any other back-end technology, I generally work in Java, and with Struts, hence my choice.

All of the examples in the webapp work with a chunk of script in the <head> of the document. Although each version has some minor differences, the all come from the same basic code. Here it is:

```
var req;
var which;

function retrieveURL(url) {
  if (window.XMLHttpRequest) { // Non-IE browsers
    req = new XMLHttpRequest();
    req.onreadystatechange = processStateChange;
    try {
      req.open("GET", url, true);
    } catch (e) {
      alert(e);
    }
    req.send(null);
  } else if (window.ActiveXObject) { // IE
    req = new ActiveXObject("Microsoft.XMLHTTP");
    if (req) {
      req.onreadystatechange = processStateChange;
      req.open("GET", url, true);
      req.send();
    }
  }
}

function processStateChange() {
  if (req.readyState == 4) { // Complete
    if (req.status == 200) { // OK response
      document.getElementById("urlContent").innerHTML = req.responseText;
    } else {
      alert("Problem: " + req.statusText);
    }
  }
}
```

The basic idea of this code is simple… You call the retrieveURL() function, passing it the URL you wish to access. It instantiates the XMLHttpRequest object appropriately depending on browser, and then opens a request to the supplied URL. Note the try…catch block in the portion of the code for non-IE browsers. This is due to the fact that some browsers (Firefox as an example) will not allow any request via XMLHttpRequest that is to a domain other than the domain the page is served from. In other words, if you access www.omnytex.com/test.htm and it tries to access www.cnn.com, the browser will not allow it. But, accessing www.omnytex.com/whatever.htm will work just fine. IE will allow this cross-site scripting, but with a user verification required.

One important line is the req.onreadystatechange = processStateChange; line. This line of code is setting up an event handler. When the state of the request changes, the processStateChange() function will be called. You can then interrogate the state of the XMLHttpRequest object and do whatever is appropriate. The previous table given lists all the possible values. All we particularly care about here is when the request is complete. The next thing we need to do is check the HTTP response code that was received. Anything other than 200 (HTTP OK) will just result in an alert with the error information displayed.

In this example, when we get a complete response back and it was OK, we insert the text we received back into the urlContent span, which appears later on in the page.

That is literally all there is to using XMLHttpRequest!

A slightly more interesting example is the second example in the webapp, dynamic sorting of a table. Here is the complete page involved:

```
<html>
<head>
<title>Example 2</title>

<script>

  var req;
  var which;

  function retrieveURL(url) {
    if (window.XMLHttpRequest) { // Non-IE browsers
      req = new XMLHttpRequest();
      req.onreadystatechange = processStateChange;
      try {
        req.open("GET", url, true);
      } catch (e) {
        alert(e);
      }
      req.send(null);
    } else if (window.ActiveXObject) { // IE
      req = new ActiveXObject("Microsoft.XMLHTTP");
      if (req) {
        req.onreadystatechange = processStateChange;
        req.open("GET", url, true);
        req.send();
      }
    }
  }

  function processStateChange() {
    if (req.readyState == 4) { // Complete
      if (req.status == 200) { // OK response
        document.getElementById("theTable").innerHTML = req.responseText;
      } else {
        alert("Problem: " + req.statusText);
      }
    }
  }

</script>

</head>
<body onLoad="retrieveURL('example2RenderTable.do');">

<h1>Example 2</h1>
Dynamic table.<hr>
<p align="right"><a href="home.do">Return home</a></p><br>
This example shows how a table can be built and displayed on-the-fly by showing
sorting of a table based on clicks on the table headers.
<br><br>

<span id="theTable"></span>
<br>

</body>
</html>
```

Notice again the nearly identical script elements in the <head>. Here we are actually making a request to a Struts Action. This action returns the actual HTML for the table, sorted appropriately. There are other ways to accomplish this that don't require you to generate HTML in the Action, but this is a quick-and-dirty approach that works quite well. We make an initial call to the Action when the page loads so that we have a table to begin with. Clicking on any of the column headers will result in the table being sorted and redrawn.

Let's take a look at another example from the webapp, this time the RSS feed parser:

```
<html>
<head>
<title>Example 6</title>
</head>

<script>

  var req;
  var which;

  function retrieveURL(url) {
    if (url != "") {
      if (window.XMLHttpRequest) { // Non-IE browsers
        req = new XMLHttpRequest();
        req.onreadystatechange = processStateChange;
        try {
          req.open("GET", url, true);
        } catch (e) {
          alert(e);
        }
        req.send(null);
      } else if (window.ActiveXObject) { // IE
        req = new ActiveXObject("Microsoft.XMLHTTP");
        if (req) {
          req.onreadystatechange = processStateChange;
          req.open("GET", url, true);
          req.send();
        }
      }
    }
  }

  function processStateChange() {
    if (req.readyState == 4) { // Complete
      if (req.status == 200) { // OK response
        // We're going to get a list of all tags in the returned XML with the
        // names title, link and description.  Everything else is ignored.
        // For each that we find, we'll constuct a simple bit of HTML for
        // it and build up the HTML to display.  When we hit a title,
        // link or description element that isn't there, we're done.
        xml = req.responseXML;
        i = 0;
        html = "";
        while (i >= 0) {
          t = xml.getElementsByTagName("title")[i];
          l = xml.getElementsByTagName("link")[i];
          d = xml.getElementsByTagName("description")[i];
          if (t != null && l != null && d != null) {
            t = t.firstChild.data;
            l = l.firstChild.data;
            d = d.firstChild.data;
            html += "<a href=\"" + l + "\">" + t + "</a><br>" + d + "<br><br>";
            i++;
          } else {
            i = -1;
          }
        }
        document.getElementById("rssData").innerHTML = html;
      } else {
        alert("Problem: " + req.statusText);
      }
    }
  }

</script>
```

```
<body>

<h1>Example 6</h1>
RSS example.<hr>
<p align="right"><a href="home.do">Return home</a></p><br>
This example is a more real-world example.  It retrieves an RSS feed from one
of three user-selected sources, parses the feed and displays the headlines
in clickable form.  This demonstrates retrieving XML from a server and
dealing with it on the client.
<br><br>
<b>Note that the RSS feed XML is actually stored as files within this
webapp.  That is because some browsers will not allow you to retrieve
content with XMLHttpRequest outside the domain of the document trying to
do the call.  Some browsers will allow it with a warning though.</b>
<br><br>
<form name="rssForm">
  <select name="rssFeed" onChange="retrieveURL(this.value);">
    <option value=""></option>
    <option value="cnn_rss.xml">CNN Top Stories</option>
    <option value="slashdot_rss.xml">Slashdot</option>
    <option value="dans_rss.xml">Dan's Data</option>
  </select>
</form>
<hr><br>
<span id="rssData"></span>
<br>

</body>
</html>
```

The first thing to note is that the RSS feed XML files are actually local files included with the webapp. A true RSS reader using XMLHttpRequest wouldn't be possible because it would be dealing with domains outside its own. However, one way around this would be to write an Action that retrieves the feed from the true URL and then returns it to the page, and that is demonstrated in example 7. Other than the Action acting as the proxy to retrieve the RSS feed, the code for the page itself is the same.

In any case, the example works very similarly to all the others except that we see an example of XML parsing here in the event handler function. This is a simplified example in that we're simply ignoring any tags except those that describe each headline. In a true example (i.e., one that was retrieving more complex XML), the parsing code would be accordingly more complex, but I leave that as an exercise to the reader.

Let's finish up by examining the script for the example that demonstrates submitting data with a request. Just the script in the <head> is:

```
  var req;
  var which;

  function submitData() {
    // Construct a CSV string from the entries.  Make sure all fields are
    // filled in first.
    f = document.theForm.firstName.value;
    m = document.theForm.middleName.value;
    l = document.theForm.lastName.value;
    a = document.theForm.age.value;
    if (f == "" || m == "" || l == "" || a == "") {
      alert("Please fill in all fields first");
      return false;
    }
    csv = f + "," + m + "," + l + "," + a;
    // Ok, so now we retrieve the response as in all the other examples,
    // except that now we append the CSV onto the URL as a query string,
    // being sure to escape it first.
    retrieveURL("example5Submit.do?csv=" + escape(csv));
  }
```

```
function retrieveURL(url) {
  if (window.XMLHttpRequest) { // Non-IE browsers
    req = new XMLHttpRequest();
    req.onreadystatechange = processStateChange;
    try {
      req.open("GET", url, true);
    } catch (e) {
      alert(e);
    }
    req.send(null);
  } else if (window.ActiveXObject) { // IE
    req = new ActiveXObject("Microsoft.XMLHTTP");
    if (req) {
      req.onreadystatechange = processStateChange;
      req.open("GET", url, true);
      req.send();
    }
  }
}

function processStateChange() {
  if (req.readyState == 4) { // Complete
    if (req.status == 200) { // OK response
      document.getElementById("theResponse").innerHTML = req.responseText;
    } else {
      alert("Problem: " + req.statusText);
    }
  }
}
```

In this example we are constructing a simple comma-separated string based on the inputs of the user. You can of course construct an XML document and send that, and in fact that is the more common example. But that is part of the reason I DIDN'T do it: I wanted to show you that you do not have to send XML when using XMLHttpRequest. In fact, this example does nothing more than append the CSV string onto the URL. There are numerous examples on the web for constructing a true XML document and submitting that using the XMLHttpRequest.send() method, and I highly recommend looking those up if you intend to use that approach.

I hope that this rather short article and accompanying webapp have given you a good starting point from which to explore what the XMLHttpRequest object provides. Before I close I want to also point out that the Ajax concept itself does NOT require you to use XMLHttpRequest. You can get very much the same basic effect in other ways, including code in a hidden frame taking the place of XMLHttpRequest. This is in fact the technique I used five years ago in the project I spoke of at the beginning. However, XMLHttpRequest does make the underlying Ajax concept far easier to implement, and more standard. Just look to Google as a great example of the power of this technique.

However, I caution anyone from thinking this is the way all web apps should be developed. I do not for a second advocate this as the One True Solution (for all you LOTR fans out there!). It is a good approach in many cases, but will not be appropriate in others. If reaching the maximum possible audience is your goal, you would want to stay away from this. If a user disabling scripting in their browser might be a concern (and your site wouldn't be any good without it), this probably isn't a good answer either. There are other reasons to stay away from it in some situations, but the bottom line is treat it like any other tool in your toolbox: it will be right for some jobs, maybe not so for others. After all, you don't drive a nail with a glue gun, right??

In any case, I hope this has given you some food for thought. Have fun!

About the author

Frank W. Zammetti is a Web Architect Specialist for a leading worldwide financial institution during the day and a PocketPC developer at night. He is the Founder and Chief Software Architect of Omnytex Technologies. He has over 10 years of professional development experience, and nearly 15 more of "amateur" experience, and he has been developing web-based applications (Intranet applications mostly) almost exclusively for nearly 7 years. Frank holds numerous certifications including SCJP, MCSD, CNA, i-Net+, A+, CIW, MCP and numerous BrainBench certifications. He is a contributor to a number of open-source projects, including DataVision, as well as having started two, including the Struts Web Services Enablement Project (search strutsws on SourceForge, or http://sourceforge.net/projects/strutsws/). Frank's resume is available online at http://www.zammetti.com/resume.

Sample webapp: http://www.omnytex.com/articles/xhrstruts/xhrstruts.zip

PDF copy of this article: http://www.omnytex.com/articles/xhrstruts/xhrstruts.pdf