The Ajax Papers

Part I: The Intro and the Basics

Ajax. We've all heard of it and most of us have already started to use it. Many of us ("us" being ASP.NET developers) probably decided to use Ajax because Telerik's radAjax component made it very easy to add Ajax to our existing projects. We figured what the heck? Telerik makes it easy to add Ajax to my site and the boss will love to see the Ajax buzz word in my list of accomplishments.

As radAjax developers, though, we often take for granted what's actually happening under the proverbial Ajax hood. How *does* radAjax "auto-magically" make our pages do this Ajax thing? In fact, what really makes this Ajax thing work?

These are questions that we'll answer in my multipart series on Ajax. We'll first establish firmly what Ajax is and how it manages to update a page without a refresh. Once we understand Ajax, we'll look at radAjax and ASP.NET AJAX and discuss how they automatically ajaxify ASP.NET pages. With a solid foundation of Ajax and radAjax knowledge, we'll move on to radAjax tips, tricks, and optimization techniques. Finally, we'll wrap the series by looking ways to measure your Ajax page performance and compare radAjax enabled pages with pages ajaxified by the ASP.NET AJAX framework (including a preview of the new Manager for ASP.NET AJAX). When we're done, you'll be Ajax (and radAjax) experts capable of maximizing the benefits Ajax can provide in your application.

What is Ajax?

There are many books and articles out there explaining the 5Ws (Who, What, Where, When, Why) of Ajax, so I won't spend much time on the history of Ajax in this series. I encourage you to read the Wikipedia article on Ajax or pick-up any book on Ajax from your local Barnes & Noble to get the full story. AJAX (Asynchronous JavaScript and XML) the term has been around for just two years (can you believe it?!), created by Jesse James Garrett in 2005. The technologies that make Ajax work, however, have been around for almost a decade.

Trivia: So is it AJAX or Ajax?

In Jesse Garrett's *original* article that coined the term, it was AJAX. The "X" in AJAX really stands for XMLHttpRequest, though, and not XML. Jesse later conceded that Ajax should be a word and not an acronym and updated his article to reflect his change in heart. So "Ajax" is the correct casing.

As its name implies, Ajax relies primarily on two technologies to work: JavaScript and the XMLHttpRequest. Standardization of the browser DOM (Document Object Model) and DHTML also play an important part in Ajax's success, but for the purposes of our discussion we won't examine these technologies in depth.

At the heart of Ajax is the ability to communicate with a web server asynchronously without taking away the user's ability to interact with the page. The XMLHttpRequest is what makes this possible. This technology was created by Microsoft as an IE ActiveX control to support their (then) groundbreaking Outlook Web Access, but it has since been built-in natively to all modern web browsers (including IE7). In fact, if Mozilla had not had a rare "Microsoft moment" and decided *not* to add support for the non-standard XMLHttpRequest to their Firefox browser, it is doubtful that Ajax would be nearly as popular as it is today.

How does Ajax work?

So now you know that the XMLHttpRequest built-in to the browser makes Ajax possible, but how do web pages use this object and what does it really do? Glad you asked. This is where we first see JavaScript added to the Ajax equation, because to use the XMLHttpRequest on a web page we must write some JavaScript. The XMLHttpRequest, like any other JavaScript object, works because the browser's JavaScript parser (or engine) recognizes the object in the code and knows how to process it.

Trivia: Why is JavaScript (and in turn, Ajax) hard to write?

In part, it is because every browser handles JavaScript just a little differently. Any web browser that wants to support JavaScript must provide its own engine to parse JavaScript commands and perform the correct browser actions. Firefox uses the open source JavaScript engine called SpiderMonkey, Safari uses an engine called JavaScriptCore, and Opera uses its own proprietary engine. IE's engine actually processes "JScript", Microsoft's own brand (literaly) of JavaScript created (in part) to avoid copyright issues with JavaScript trademark holder Sun. Even though "JavaScript" is standardized by Ecma, each engine does things a little differently. That means solid JavaScript or Ajax programming must be done in a way that accounts for these differences. Aren't you glad you use the Telerik magic now?

Let's start to put these ideas together in some code examples. The basic implementation of the XMLHttpRequest in JavaScript looks like this:

```
        //Get a reference to the XMLHttpRequest object
        var xmlRequest = (window.XMLHttpRequest) ? new XMLHttpRequest() : new
        ActiveXObject("Msxml2.XMLHTTP");

        //If the browser doesn't support Ajax, exit now
        if (xmlRequest == null)
               return;

        //Initiate the XMLHttpRequest object
        xmlRequest.open("POST", url, true);

        //Since this is a POST, set the request Content-Type header
        xmlRequest.setRequestHeader("Content-Type", "application/x-www-form-
urlencoded");

        //Setup the callback function
        xmlRequest.onreadystatechange =
function(){HandleAjaxResponse(xmlRequest);};

        //Send the Ajax request to the server with the POST data
       xmlRequest.send(args);
```

That's it. That's Ajax. Really. Any additional code you see is written to handle the response that the Ajax request returns (no small feat as we'll see later).

Trivia: How does ASP.NET AJAX do it?

To examine the ASP.NET AJAX JavaScript in a readable format, simply open the MicrosoftAjax.debug.js file located in in your ASP.NET AJAX installation directory. Since Microsoft built a complete client-side programming model with the ASP.NET AJAX Extensions, the "Ajax" code in MicrosoftAjax.js is actually a small portion of its overal functionality. Nonetheless, we can find the Ajax code in the Sys.Net.XMLHttpExecutor client-side class (lines 3989 – 4267). The XMLHttpExecutor is actually executed by another ASP.NET AJAX class, Sys.Net.WebRequestManager. The XMLHttpExecutor is just one "Executor" that the WebRequestManager can handle. We'll examine the ASP.NET AJAX code in more detail later in this series.

In this example, we begin by creating a new instance of the XMLHttpRequest object:

```
var xmlRequest = (window.XMLHttpRequest) ? new XMLHttpRequest() : new
ActiveXObject("Msxml2.XMLHTTP");
```

We do this by first checking to see if the XMLHttpRequest object is natively built-in to the browser; if it's not we assume we're in IE and we create our XMLHttpRequest object using IE's ActiveX component. If a browser does not support Ajax, the "xmlRequest" object will be null and our Ajax function will exit on the following line.

Trivia: Msxml2.XMLHTTP vs. Microsoft.XMLHTTP

If you look at different implementations of Ajax, you'll notice that some use "Microsoft.XMLHTTP" and some use "Msxml2.XMLHTTP" when targeting IE's ActiveX control. So what's the difference? As it turns out, very little when written like that. While the "Microsoft" namespace is older than the "Msxml2" namespace, written like this both statements will target MSXML 3.0 (the most widely distributed version of MSXML). The *latest* version of MSXML, though, is version 6.0 (released July 2006). Vista ships with version 6.0 installed and it is available for download for XP, Win2k, and Win2k3. To target the latest and most secure version of MSXML, you must use "Msxml2.XMLHTTP.6.0" to create your XMLHttpRequest. Leaving the version number off on a system with 3.0 installed will always target MSXML 3.0 (even if 6.0 is installed). Go figure.

Next we open our connection to the server with our newly created XMLHttpRequest object:

```
xmlRequest.open("POST", url, true);
```

You must supply three parameters to the "open" method:

1. **Request Method:** either "POST" or "GET" (case sensitive). The "POST" and "GET" values should be familiar as they are normal Http concepts. Like normal Http requests, GETs are processed slightly faster by the server than POSTs, but they do not allow any data to be sent to the server (other than what you can fit into the URL's querystring)
2. **URL**: the request URL (must be in your domain)
3. **Async Flag:** a Boolean value indicating if the request should asynchronous. That's right. You can actually use the XMLHttpRequest to do SJAX (*Synchronous* JavaScript blah blah). This is rarely done for obvious reasons, so the third parameter in the "open" method will almost always be true.

Important note: even though we've opened our connection at this point we have not yet sent our request to the server.

Trivia: radAjax OnRequestStart and OnRequestSent

The XMLHttpRequest's "open" method fires just after the radAjax OnRequestStart client event. The OnRequestSent client event fires right after the XMLHttpRequest's "send" method fires. Measuring the time between OnRequestSent and OnResponseReceived tells you exactly how long it took for your server to process and send your Ajax response.

Since this is a POST, we then set the Content-Type header for the request:

```
xmlRequest.setRequestHeader("Content-Type", "application/x-www-form-
urlencoded");
```

If this were a GET request, we would not need to set the Content-Type here.

You often hear the term "callback" replace the term "postback" when you work with Ajax. That's because Ajax uses a "callback" function to catch the server's response when it is done processing your Ajax request. We establish a reference to that callback function like this:

```
xmlRequest.onreadystatechange = function(){HandleAjaxResponse(xmlRequest);};
```

The HandleAjaxReponse receieves a reference to our XMLHttpRequest object so that it can process the server's response. OnReadyStateChange will fire multiple times during an Ajax request, so we must evaluate the XMLHttpRequest's "readyState" property to determine when the server response is complete. A simple callback function may look something like this:

```
function HandleAjaxResponse(xmlhttp){
        //If readyState = 4, the server response is complete
        if (xmlhttp.readyState === 4) {
                //We can evaluate the status property to see what HTTP response
                code was returned
                //Status' in the 200's are okay; "200" is the best
                 var statusCode = xmlhttp.status;
                 if (statusCode < 200) || (statusCode >= 300){
                        //Process the server response
                        processResponse(xmlhttp.responseXML);
                    }else{
                        //Some error handling
                        processError(xmlhttp)
                    }
            }
    }
```

Trivia: HTTP Status Code 304

Http Status code 304 is technically a valid response code that could be returned from the server when performing a GET. It indicates that the page has not been changed and the page in the browser's cache should be used. In Firefox, the XMLHttpRequest status property will return "200" if the server responds with "200" or "304". IE will also return status code 200 in the XMLHttpRequest GET response, so a solid implementation of your callback function does not need to check for both codes. radAjax currently throws an error for any response that does not return code 200.

The XMLHttpRequest has several properties that we're interested in during our response callback function:

- **readyState**: indicates if the server is done processing our Ajax request. A value of "4" indicates the request is complete (true for all browsers). The number and values of the codes returned before "4" vary by browser, but the possible values are:
    - **0**: uninitialized
    - **1**: loading
    - **2**: loaded
    - **3**: interactive
    - **4**: complete
- **status**: returns the HTTP response code sent by server for our Ajax request. A value of "200" means "OK" and that no errors occurred. Any other value indicates a situation that should be handled by our JavaScript Ajax error handler.
- **responseXML**: contains the XML formatted response from the server. This is actually an XML Document Object that can be parsed using XPath or regular DOM node tree methods (like getElementByTag, etc.). RadAjax primarily uses responseXML to process Ajax responses.
- **responseText**: contains the raw text response from the server

If there is an error, we are also interested in this property:

- **statusText**: contains the text describing our response status code. If we receive a HTTP response code of "404", for instance, the statusText would contain "Not Found".

Trivia: JavaScript's little know '===' operator

If you examine the ASP.NET AJAX JavaScript source code, you'll see lots of "===" compare operators where you'd expect to find the normal "==" operator. Both will evaluate if an object is equal, but the "===" takes it another step further and validates that the objects being compared share the same identity. That means, in order for "===" to return true, the objects must be equal *without* JavaScript performing any data type conversions. This provides strict equality tests in JavaScript where loosely typed objects can often cause problems. And yes, "!==" exists, too.

At this point we've sent our request to the server, received a response, and now we have an XML document object sitting in JavaScript memory. We've completed our asynchronous communication with the server, but now we need to update the page's DOM. After all, the point of Ajax is to update the page without doing a full PostBack (and thus a *full* refresh) of the page. What should now be obvious is that the harder part of creating an Ajax application is implementing the code that parses the server response and updates the page; the communication is actually fairly straight forward and easy.

What's next?

In the next part of our multi-part Ajax series, we'll look at how JavaScript is used to parse our Ajax server response and update our page. We'll see how DOM node tree methods and JavaScript are used to determine which parts of the page need updating and we'll learn why it's important to optimize our updated controls to get the most out of Ajax. So stick around, the best is yet to come.

Todd Anglin, Technical Evangelist
Telerik
www.telerik.com

References

1. Wikipedia, multiple articles, www.wikipedia.com
2. w3schools, The XMLHttpRequest Object, http://www.w3schools.com/xml/xml_http.asp
3. Sitepoint, Build Your Own AJAX Web Applications, http://www.sitepoint.com/article/build-your-own-ajax-web-apps/
4. Adaptive Path, Ajax: A New Approach to Web Applications, http://www.adaptivepath.com/publications/essays/archives/000385.php
5. Web Master World, JavaScript Jumpstart – Operator Basics, http://www.webmasterworld.com/forum91/513.htm
6. Microsoft XML Team Weblog, Using the right version of MSXML in Internet Explorer, http://blogs.msdn.com/xmlteam/archive/2006/10/23/using-the-right-version-of-msxml-in-internet-explorer.aspx