At the Forge

Beginning Ajax

Reuven M. Lerner

Abstract

How to put the A (asynchronous) in Ajax.

Many programmers, myself included, have long seen JavaScript as a way to change the appearance of a page of HTML dynamically or to perform relatively minor tasks, such as checking the validity of a form. In the past year, however, JavaScript has emerged as a major force for application developers, providing the infrastructure for so-called Ajax applications.

Before JavaScript, there was a one-to-one correspondence between user actions and the display of HTML pages. If a user clicked on a link, the currently displayed page disappeared and was replaced with another page of HTML. If a user submitted an HTML form, the contents of that form were submitted to a program on the Web server, and the content of the server's response was then displayed in the browser, replacing its predecessor. In traditional Web applications, server-side programs handle the bulk of user input and also build any dynamically generated Web pages the user might see.

Ajax applications redistribute the load, putting a greater emphasis on client-side JavaScript. In an Ajax application, many server-side programs do indeed produce complete pages of HTML, which are then displayed in their entirety in a Web browser. But many other server-side programs produce small snippets of XML-formatted data. This data is both requested and used by client-side JavaScript to modify and update the current HTML page without having to refresh or replace it. Using Web standards, such as the DOM (document object model) and CSS (cascading stylesheets), Ajax applications can approach the usability, friendliness and instant feedback that people expect from desktop applications.

This month, we continue exploring client-side JavaScript and Ajax, which we began during the past few months. Last month's column looked at a user-registration application for a Web site. Although the actual registration took place in a server-side program, we looked at ways in which we could provide an Ajax-style warning for registering users who wanted a user name that was already taken. Sure, we could have the server-side registration program check to see whether the user name had been taken already, but that would require refreshing the page, which also requires a delay.

The solution we implemented last month was fine from the user's perspective (especially if the user has somewhat Spartan tastes in design), but it solved the problem in a very non-Ajax way—by hard-coding the user names in a JavaScript array and then looking for the desired new user name in that array. This approach has a number of large problems associated with it, starting with the fact that the full list of user names is available to anyone looking at the HTML source and ending with the fact that the array will become unwieldy and cumbersome over time, taking an increasingly long time to download and search through as the number of registered users grows.
We can avoid these problems by using an Ajax-style solution. Rather than hard-code the list of user names in the JavaScript, and instead of having the server-side program produce a full list of user names, perhaps we could simply send a request to the server, checking to see if the requested user name is already taken. This will result in relatively fast download and reaction times, in a cleaner application design and in an extensible application.

This month, we take the Ajax plunge, modifying the server- and client-side programs we wrote last month to retrieve user names via an asynchronous request from the server. In producing this application, we will see how relatively straightforward it can be to create an Ajax application or to integrate Ajax functionality into a traditional Web application. By the end of this article, you should understand how to create the client and server sides of an Ajax application.

**Making an Ajax Call**

The technology that makes much of Ajax possible is JavaScript's XMLHttpRequest object. Using this object, a JavaScript function can make HTTP requests to a server and act on the results. (For security reasons, HTTP requests made by XMLHttpRequest must be sent to the server from which the current Web page was loaded.) The HTTP request may use either the GET or POST method, the latter of which allows us to send arbitrarily long, complex content to the server.

Most interesting, and at the core of many Ajax paradigms, is the fact that XMLHttpRequest may make its HTTP requests synchronously (forcing the browser to wait until the response has been completely received) or asynchronously (allowing the user to continue to use the browser window as it downloads additional information). Ajax applications typically use asynchronous calls. This allows different parts of the Web page to be updated and modified independently of one another, potentially responding simultaneously to multiple user inputs.

Ideally, we would be able to create an instance of XMLHttpRequest with the following JavaScript code:

```javascript
var xhr = new XMLHttpRequest();
```

Unfortunately, life isn't that simple. This is because many people use Internet Explorer as their primary browser. IE does not have a native XMLHttpRequest object, and thus it cannot be instantiated in this way. Rather, it must be instantiated as:

```javascript
var xhr = new ActiveXObject("Msxml2.XMLHTTP");
```

But wait! There are also some IE versions that require a slightly different syntax:

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

How are we going to handle these three different ways of instantiating XMLHttpRequest? One way is to use server-side browser detection. It is also possible to use client-side browser detection. But the most elegant method I have seen to date comes from *Ajax Design Patterns*, a new book by Michael Mahemoff (published by O'Reilly Media). Mahemoff uses JavaScript's exception-handling system to try each of these in turn until it works. By wrapping our three different instantiation methods in a function, and then assigning the value of our xhr variable to whatever the function returns, we can give our application cross-platform compatibility:

```javascript
function getXMLHttpRequest () {
```
try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch(e) {};
try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch(e) {};
try { return new XMLHttpRequest(); } catch(e) {};
return null;
}

var xhr = getXMLHttpRequest();

After executing the above code, we can be sure that xhr is either null (indicating that all attempts to instantiate XMLHttpRequest failed) or contains a valid instance of XMLHttpRequest. Once instantiated, XMLHttpRequest is compatible across browsers and platforms. The same methods thus will apply for all systems.

The most common method to call on xhr is open, which tells the object to send an HTTP request to a particular URL on the originating server. A call to xhr.open looks like this:

xhr.open("GET", "foo.html", true);

The first parameter (GET) tells xhr.open that we want to use the HTTP GET method. The second parameter names the URL that we want to retrieve; notice that because we must connect to the originating server, the initial protocol and hostname part of the URL is missing. The third parameter indicates whether the call is asynchronous (true) or synchronous (false). Almost all Ajax applications pass true, as this means that the browser doesn't freeze up while it is waiting for the HTTP response. This ability to make asynchronous HTTP requests is central to the magic of Ajax. Because the HTTP request doesn't affect the user interface and is handled in the background, the Web application feels more like a desktop application.

The call to xhr.open() does not actually send the HTTP request. Rather, it sets up the object so that when the request is sent, it uses the specified request method and parameters. To send the request to the server, we use:

xhr.send(null);

XMLHttpRequest does not return the HTTP response whoever calls xhr.send(). This is because we are using XMLHttpRequest asynchronously, as specified with the true value to xhr.open(). We cannot predict whether we will get results in half a second, five seconds, one minute or ten hours.

Instead, we tell JavaScript to invoke a function when it receives the HTTP response. This function will be responsible for reading and parsing the response and then taking appropriate action. One simple version of the function, which I have called parseHttpResponse, is as follows:

function parseHttpResponse() {
    alert("entered parseHttpResponse");
    if (xhr.readyState == 4) {
        alert("readystate == 4");
        if (xhr.status == 200) {
            alert(xhr.responseText);
        } else {
            alert("xhr.status == " + xhr.status);
        }
    }
}


parseHttpResponse is called when the HTTP response to our Ajax request comes in. However, we have to make sure that the response contents have completely arrived, which we do by monitoring xhr.readyState. When that equals 4, we know that xhr has received the complete response. Our next step is then to check that the response had an HTTP “OK” (200) code. After all, it is always possible that we got a 404 (“file missing”) error from the server, or that we failed to connect to the server at all.

To tell JavaScript we want to invoke parseHttpResponse when our HTTP request returns, we set the onreadystatechange attribute in our XMLHttpRequest object:

xhr.onreadystatechange = parseHttpResponse;

Finally, after we can be sure that we have received the response and that all is well, we can grab the text of the response with the xhr.responseText method. Our XMLHttpRequest can return its response either as a text string (as here) or as an XML document. In the latter case, we then can use the DOM to navigate through it, much as we would do with a Web page.

Of course, an actual Ajax application would not issue an alert at every step of its execution and would probably do something more useful—perhaps changing some text, adding or removing some nodes from the document tree or changing part of the document’s stylesheet. Nevertheless, you can see this code in action in Listing 1 (ajax-test.html).

Listing 1. ajax-test.html925611.qrk

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>Ajax test</title>

<script type="text/javascript">
function getXMLHttpRequest () {
try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch(e) {};
try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch(e) {};
try { return new XMLHttpRequest(); } catch(e) {};
return null;
}

function parseHttpResponse() {
    alert("entered parseHttpResponse");
    if (xhr.readyState == 4) {
        alert("readystate == 4");
        if (xhr.status == 200) {
            alert(xhr.responseText);
        } else {
            alert("xhr.status == " + xhr.status);
        }
    }
}
var xhr = getXMLHttpRequest();
alert("xhr = " + xhr);
xhr.open("GET", "atf.html", true);
xhr.onreadystatechange = parseHttpResponse;
xhr.send(null);
</script>
</head>
```
Note that ajax-test.html, although simple, is a fully working Ajax program. In order for it to work, you need to have a file named atf.html in the DocumentRoot directory of your Web site. (Otherwise, you will get an HTTP response code of 404.) If you've ever wondered how hard it is to perform an Ajax call, you now can see that it's relatively simple.

Adding Ajax to Registration

Now that we have seen how an Ajax program works, let's use this knowledge to modify the registration program that we built last month. Our old registration page defined a list of user names in the JavaScript. If the user's requested user name was a member of that list, we alert the user to the error and forbid the user from actually registering.

I won't describe all of the problems with this approach, as there are many. As a simple alternative, what if we were to use Ajax to retrieve the list of user names? That way, we could be sure that the list was up to date.

What if, instead of having the array contents hard-coded, we were to download them from a Web page on the server? (This is admittedly not as sophisticated as getting a yes or no answer to a specific user name; we will get to that functionality in next month's column.) If the Ajax-retrieved list of user names was generated dynamically, we could have it grab appropriate data from the database and then return an XML document that easily could be turned into an array. To make the example easier in this month's column, we don't use a dynamic page, but rather a static one. However, if you have done any server-side Web programming in the past, you probably will understand how to take our file, usernames.txt (Listing 2), and turn it into a dynamic page.

Listing 2. usernames.txt

```
abc
def
ghi
jkl
mno
pqr
stu
vwx
yzz
```

A registration page that follows this principle is shown in Listing 3. That file, ajax-register.html, is similar to the registration form we created last month. In last month's non-Ajax version, we defined an array (usernames). We then defined a checkUsername function that is invoked by the onchange handler for the username text field. This had the effect of invoking checkUsername when the user completed the user name. If the requested user name was in the usernames array, the user was given a warning, and the submit button was disabled. Otherwise, the user was able to submit the form to the server-side registration program, presumably as a first step to participating in the site.
Listing 3. ajax-register.html9256l3.qrk

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>Register</title>
<script type="text/javascript">
function getXMLHttpRequest () {
    try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch(e) {};
    try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch(e) {};
    try { return new XMLHttpRequest(); } catch(e) {};
    return null;
}

function removeText(node) {
    if (node != null)
    {
        if (node.childNodes)
        {
            for (var i=0 ; i < node.childNodes.length ; i++)
            {
                var oldTextNode = node.childNodes[i];
                if (oldTextNode.nodeValue != null)
                {
                    node.removeChild(oldTextNode);
                }
            }
        }
    }
}

function appendText(node, text) {
    var newTextNode = document.createTextNode(text);
    node.appendChild(newTextNode);
}

function setText(node, text) {
    removeText(node);
    appendText(node, text);
}

var xhr = getXMLHttpRequest();

function parseUsernames() {
    // Set up empty array of usernames
    var usernames = [ ];

    // Wait for the HTTP response
    if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            usernames = xhr.responseText.split("\n");
        }
    else
    {
        alert("problem: xhr.status = " + xhr.status);
    }

    // Get the username that the person wants
    var new_username = document.forms[0].username.value;
```
var found = false;
var warning = document.getElementById("warning");
var submit_button = document.getElementById("submit-button");

// Is this new username already taken? Iterate over
// the list of usernames to be sure.
for (i=0 ; i<usernames.length; i++)
{
    if (usernames[i] == new_username)
    {
        found = true;
    }
}

// If we find the username, issue a warning and stop
// the user from submitting the form.
if (found)
{
    setText(warning, "Warning: username '" + new_username +"' was taken!");
    submit_button.disabled = true;
}
else
{
    removeText(warning);
    submit_button.disabled = false;
}

function checkUsername()
{
    // Send the HTTP request
    xhr.open("GET", "usernames.txt", true);
    xhr.onreadystatechange = parseUsernames;
    xhr.send(null);
}

To turn last month's registration page into an Ajax-style one, we modify the checkUsername function, which is invoked when the user finishes entering his or her requested user name. Instead of defining the usernames array, we instead have checkUsername fire off an Ajax request to the server. Unlike last month's non-Ajax version, this is all that checkUsername will do. The updated function looks like this:

function checkUsername()
{
    xhr.open("GET", "usernames.txt", true);
    xhr.onreadystatechange = parseUsernames;
    xhr.send(null);
}
As you can see, our function is requesting the file usernames.txt from the server. When xhr's state changes, we ask to invoke the parseUsernames function. It is in this function that we have put the serious logic, first turning the retrieved file contents into an array:

```javascript
var usernames = [ ];
if (xhr.readyState == 4) {
  if (xhr.status == 200) {
    usernames = xhr.responseText.split("\n");
  }
}
```

Here, we see the standard Ajax pattern repeated from the previous example: wait for xhr.readyState to be 4, and then check that xhr.status (the HTTP response status code) is 200. At that point, we know we have received the contents of usernames.txt, which (as you can see from Listing 2) contains the existing user names, one user name per line. We use JavaScript's split function to turn this into an array, which we assign to usernames.

From this point on, we can reuse the logic from last month's non-Ajax version, first grabbing the various node IDs from the page, using DOM methods:

```javascript
var new_username = document.forms[0].username.value;
var found = false;
var warning = document.getElementById("warning");
var submit_button = document.getElementById("submit-button");
```

Then, we check to see if the requested user name is in our array:

```javascript
for (i=0 ; i<usernames.length; i++)
{
  if (usernames[i] == new_username)
  {
    found = true;
  }
}
```

If the user name is found in the list, we issue a warning at the top of the page. Otherwise, we clear out any warning that might be there:

```javascript
if (found)
{
  setText(warning, "Warning: username '" + new_username +"' was taken!");
  submit_button.disabled = true;
}
else
{
  removeText(warning);
  submit_button.disabled = false;
}
```

Now, is this a good way to handle the checking of user names? Not really—although now that we have the basic Ajax logic in place, we can modify it slightly to be more efficient and secure.

One problem is that the list of user names is in a static file. Perhaps our server is running a cron job that
creates usernames.txt on a regular basis, but that seems a bit silly when we can instead use a server-side program to query the database dynamically. Switching from a static file to a dynamic page thus seems like a good idea, if only for performance reasons.

There are security reasons as well. As with last month's version, we are downloading the entire list of user names to the user's browser. This means that a potentially malicious user would have access to all of the user names and would be able to poke through them, either with the intention of trying to break into the site or spam the users.

One potential downside of using Ajax for this type of check is the speed issue. As I indicated previously, the core of Ajax is its asynchronous nature, which means that we cannot know how long it will take for the server to respond to our query. In my simple tests, the round trip from my browser to my server and back was nearly instantaneous, and it provided me with useful feedback right away. On a more heavily loaded server, or with a more sophisticated database query, or if users have slow Internet connections, asynchronous calls might begin to feel sluggish. That said, even the worst Ajax function will likely be faster than a page refresh, because of the reduced overhead that is involved.

**Conclusion**

This month, we finally begin to use Ajax in an application. We see here how it is possible to take some existing JavaScript code and break it apart into two functions: one that invokes the Ajax call and the other that handles the parsing of data when the call receives a response.

However, we also see that there are security and efficiency problems with this approach. A better technique would be to send only the requested user name in the Ajax call and get a simple yes or no answer from the server, indicating whether the user name had been taken already. Next month, we will do just that, using an Ajax POST query instead of our GET query from this month, and replacing usernames.txt with a server-side program that works in conjunction with our Ajax call.

**Further Reading**

There has been an explosion of books and articles about Ajax programming in the last year, and I am slowly making my way through many of them. Two of the best that I've read are both published by O'Reilly. *Head Rush Ajax* is aimed at beginners and teaches the introductory material in a fun, effective way. *Ajax Design Patterns*, which I mentioned earlier in this article, is probably my favorite Ajax book so far (despite its design and editing, which aren't up to the usual O'Reilly standards). This latter book is a good introduction to the subject for experienced Web developers.

The Ajaxian.com Web site has a large number of links, tutorials and articles having to do with Ajax development on a variety of different platforms. If you're interested in Ajax development, it's worth keeping this site in your RSS reader or bookmarks.