

ColdFusion and AJAX

A primer



By Jeffrey Houser

One of the most annoying things about the Web page paradigm is that you have to reload the page whenever you want to do something on the server side. If you want to do a site search, it's sprinkled

across two pages.

On the first page, the user will enter a search term, then clicks a submit button to get to the second page where the search results are returned. If the user wants to filter or sort the search results it's usually refreshed again and each result is redisplayed in its filtered state.

Lately there's been a movement to help improve the Web by eliminating these constant page refreshes. Macromedia's Rich Internet Application (RIA) push with Flash and Flex has been at the forefront of this effort, however Flex' entry price has made it prohibitive for many ColdFusion Developers. That's where AJAX comes in. It's an alternate way to create a RIA and relies on technologies built into most browsers, making it completely free.

Dissecting AJAX

AJAX stands for Asynchronous JavaScript and XML. It's not a single technology, but rather a combination of different technologies that are used to create a Rich Internet Application feel. You've probably seen examples of this kind of application. Google uses AJAX a lot on such services as maps.google.com. After bringing up a location, you can click and drag the map, zoom in or out, or even bring up directions to and from the location. All this is done without any screen refreshes. You have to admit that the interface is pretty sweet. (Note: Yahoo has a similar mapping application, built entirely in Flex at <http://maps.yahoo.com/beta>.)

Since AJAX is a combination of technologies you're going to want to know what you're getting into before going any farther, right? I thought so. Here's the list:

- **XHTML:** To start with you need a display language. Most resources on AJAX say you have to use XHTML, but regular HTML will work too.
 - **XMLHttpRequest:** The XMLHttpRequest object lets you retrieve data from the server without refreshing the page. It's a JavaScript object implemented in most major browsers. As the XMLHttpRequest object is being called, the user can do other things on the page. This is where the "asynchronous" part of the acronym comes in.
 - **iFrames:** iFrames are in-line frames and are sometimes used as an alternative to the XMLHttpRequest object. A regular frame will divide the browser window into multiple separate pages, but an in-line frame embeds one page inside another. I won't get into any iFrames details in this article.
 - **XML:** You all know what XML is, right? As a refresher, it's a way to describe data. If you need more details, read the article in this column from August 05 at <http://coldfusion.sys-con.com/read/117667.htm>.
 - **Document Object Model:** The document object model defines your HTML page through a programmer API.
 - **JavaScript:** I'm sure you all know what JavaScript is too. It's the most common language used for providing client-side functionality to HTML pages.
- How do all these different technologies work together using AJAX? Well, it generally works something like this:
1. First, the user comes to your page and the page loads. Something (depending on what your page does) is loaded and displayed to the user.
 2. The user does something such as clicking a link. Instead of reloading the page (as would happen in a normal Web app), that link fires off a JavaScript function.
 3. The JavaScript function uses XMLHttpRequest object to retrieve XML data from the server.
 4. Then the JavaScript parses the XML Data and through the Document Object Model will change the page without a refresh.
- So the user gets to see more data quicker without the annoying page refreshes. That's fantastic, right? Why isn't everyone using this? Well, as with all things, AJAX isn't perfect. There are drawbacks:
- **Development Time:** Working with AJAX will bring you back to the old days of Web development, especially if you want any kind of cross-compatibility. There are different syntaxes for creating the XMLHttpRequest object in IE vs Netscape/Firefox, which can lead to extended development times.
 - **User Experience:** When used properly, an AJAX can enhance the user experience. However, when used improperly it can

make it horrendous. Since the page isn't reloading, what happens when the user hits the back button? He'll leave the application altogether. I bet he expected to go to the app's "previous state." What happens when the user bookmarks a page? He'll go the "initial" state of the app. There are work-arounds, of course, but they're not trivial to implement.

- **Searches:** The single-page nature of the app makes indexing by search engines a tetchy proposition at best.
- **Response Times:** The very thing that works for you (no page reloads) can also work against you. What happens if you're retrieving a lot of data from the server? And the user is given no indication that something is happening.
- **JavaScript:** JavaScript must be enabled for an AJAX app to work. If JavaScript is disabled, your user isn't going to have a lot of fun with an AJAX app.
- **Accessibility:** AJAX is, most likely, not going to meet accessibility guidelines. If this is a goal of your project you're going to have to provide a fallback option. That means developing two paths to the same goal.

There are a few real simple examples of AJAX that you've probably used or seen that existed long before the term AJAX was conceived. Have you ever rolled your mouse over a graphic navigation button only to have that graphic change? JavaScript rollovers can be considered a precursor to AJAX. They do use JavaScript and they do change the state of the page. Have you turned on or off a DHTML layer on a page based on some selection by the user? This is another example of AJAX. If an application has a select box with an "other" option in it I'll often use DHTML to display an input text box if "other" is selected. If other isn't selected the input box is hidden. Both of these are simple examples of changing the page without a screen refresh. They don't go out to the server to get data, but they still fit the RIA paradigm.

Putting This All Together

Let me show you how this all comes together, so you can actually do something useful. The first step in the process is to initialize the XMLHttpRequest object. Unfortunately, there's not consistent way to do this across browsers:

```
<script language="javascript" type="text/javascript">
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;
    }
  }
}
```



```
}
</script>
```

The first line initializes the request variable, giving it a Boolean value of false. You can check against this later to make sure that your request object was properly initialized. The next code creates an instance of the XMLHttpRequest object using 'new XMLHttpRequest.' This works fine for most non-Microsoft browsers, but won't work in IE. If it doesn't initialize, then an error is thrown, which is where the first catch statement comes into play. It attempts to re-initialize using the method available in the current version of Internet Explorer: new ActiveXObject("Msxml2.XMLHTTP").

If that fails, it tries the version from older versions of Internet Explorer new ActiveXObject("Microsoft.XMLHTTP"). That's kind of complicated right? I borrowed this initialization code from the article <http://www-128.ibm.com/developerworks/Web/library/wa-ajaxintro2/?ca=dgr-lnxw07AJAX-Request> (which you should all read because it goes into a lot more details about error checking that are outside the scope of this article).

So, now the code has created the object, but what can you do with it? As you can probably see, it doesn't do much yet. You want to assign it a URL, load the data, and then process the data in some way. This function will do that:

```
function getRSS() {
  var url = "/htdocs/jeffhousercom/wwwroot/rss.cfm";
  request.open("GET", url, true);
  request.onreadystatechange = updatePage;
  request.send(null);
}
```

This is function to grab an RSS feed. The first line defines the URL. This is standard JavaScript for defining a variable. You should bear in mind that you can only request URLs from the current Web site. In this case, I'm grabbing a RSS feed from a local copy of my blog. The next line calls a method on the request object entitled open. Open prepares the request to be sent, but doesn't actual send it. There are five arguments to the open function (although we only use three in this example):

- **Request-type:** This specifies the request type, either get or post, for the URL that you want to send.
- **url:** The URL specifies the URL that you're getting (or posting) to.
- **Asynchronous:** The async value is a Boolean value that specifies whether this request is asynchronous or not. For AJAX use, you'll want to specify true. If set to false, the application will hold until this request is finished, which defeats the purpose of an RIA in the first place.
- **username:** The username specifies the username required for accessing the URL, if applicable.
- **password:** The password specifies the password required for accessing the URL, if applicable.

Okay, you've set up the request to be ready to send. The next line specifies the onreadystatechange property. This property speci-

fies what function to call once the request is complete. If you don't specify a method to be called then nothing will happen once the request is complete. The final line calls the send method. The send method is the one that will execute the request. Its parameter is the data that you need to send along with the request. In our example, we aren't sending parameters, so we pass a null value.

Once the request complete, the updatePage function will execute, so lets look at that next:

```
function updatePage() {  
    if (request.readyState == 4){  
        Form1.XMLValues.value = request.responseText;  
    }  
}
```

This function first checks the readyState of the XMLHttpRequest object. Exploring that is beyond the scope of this article, but there are a lot more details in some of the references provided at the end. For now, all you need to know is that if the readyState is 4, then the request is complete. This takes the resulting value from the server response and assigns it to a form textbox. The form is below:

```
<form name="Form1">  
    <textarea name="XMLValues" rows="20" cols="50"></textarea>  
</form>
```

The text box will contain the XML from the RSS feed. There's

another property associated with the XMLHttpRequest object called responseXML. This will contain the results as an XML Document object, which JavaScript can access natively. Unfortunately specific details are beyond the scope of this article, but I found this good resource to get you started on that: <http://www.peachpit.com/articles/article.asp?p=29307&seqNum=4&rl=1> .

Where To Go from Here

If you want to learn more about AJAX, there are some great resources you can read. The article by Jesse James Garrett that started it all is at <http://www.adaptivepath.com/publications/essays/archives/000385.php>. Garrett is credited with coining the term AJAX. You can follow an AJAX blog at <http://www.ajax-powered.net>, which talks about all things AJAX. Finally, there are a few people who are working on ways to integrate ColdFusion and AJAX together. One is the CFAJAX project <http://www.indiankey.com/cfajax>. Another is the AJAXCFC project at <http://www.robgonda.com/blog/projects/ajaxcfc>.

About the Author

Jeff Houser has been working with computers for over 20 years. He owns a DotComIt, a web consulting company, manages the CT Macromedia User Group, and routinely speaks and writes about development issues. You can find out what he's up to by checking his Blog at www.jeffryhouser.com.

jeff@instantcoldfusion.com

Efficient Web Content Management

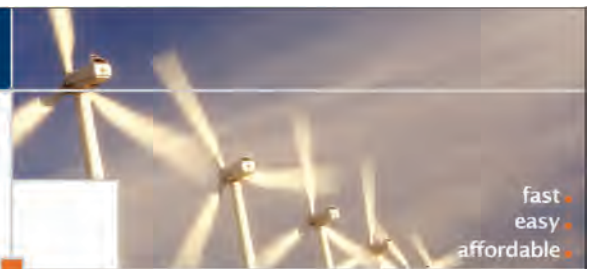
CommonSpot™ Content Server is the ColdFusion developer's leading choice for efficient Web content management.

Our rich Web publishing framework empowers developers with out-of-the-box features such as template-driven pages, custom content objects, granular security, and powerful ColdFusion integration hooks (just to name a few), allowing you to rapidly build and efficiently deploy dynamic, high performance Web sites.

CommonSpot's open architecture and extensive APIs enable easy customization and integration of external applications. With CommonSpot, you can design and build exactly what you need. Best of all, CommonSpot puts content management in the hands of content owners. With non-technical users responsible for creating and managing Web content, developers are freed to focus on strategic application development.

Evaluate CommonSpot today.

To see your site *running* under CommonSpot, call us at 1.800.940.3087.



fast
easy
affordable

features .

- 100 % browser-based
- Content object architecture
- Template driven pages
- 50+ standard elements
- Content reuse
- Content scheduling
- Personalization
- Flexible workflow
- Granular security
- Mac-based authoring
- 508 compliance
- Full CSS support
- Custom metadata
- Taxonomy module
- Extensible via ColdFusion
- Web Services content API
- Custom authentication
- Replication
- Static site generation
- Multilanguage support

1.800.940.3087
www.paperthin.com

Paper | Thin

© Copyright 2005 PaperThin, Inc. All rights reserved.