Programming websites with AJAX

# FASTER, HIGHER, FARTHER

AJAX offers a fast and efficient approach for building interactive websites. We'll show you how to call upon the powers of AJAX for your own web creations.

BY OLIVER FROMMEL

Web 2.0 isn't only about the user experience. Several important new developer tools are also helping to create faster and more efficient websites. One of the most important features of the new Internet is a technology known as AJAX.

On the old Internet, if you filled out a web form with several input fields and sent it to the server, you were expected to wait while the server evaluated your input and responded with a new page (Figure 1). An AJAX-based website provides a more elegant solution. Instead of re-requesting the whole page, the web browser simply requests a small fragment of the page. The user continues to work as the request if filled by the server, and the browser goes on to merge the data with the existing page. In the user's experience, the website is almost as fast as a desktop application.

The name AJAX was coined by Jesse James Garrett [1] in his essay "AJAX: A New Approach to Web Applications." Although Garrett maintains that AJAX is not an acronym, most people take as a shortcut for Asynchronous Javascript and XML. AJAX websites are not built from static HTML files (and CSS stylesheets), but are, instead, comprised of Javascript code that runs when a user clicks a link or triggers some other kind of event. Javascript functions request data from the server, which returns the XML (this explains the X) and HTML-formatted data, along with other formats.

This article provides a hands-on look at how to put AJAX to work for your own website.

## Asynchronous

Because the AJAX request is asynchronous (this explains the A), the user can continue to interact with other parts of the HTML page without blocking the browser. An asynchronous request emancipates the request from the response. Once the browser's Javascript code has issued a request, it just goes on running. When the response reaches the browser, it calls a specific Javascript function, which in turn merges the result with the existing HTML page. The result can be as simple as a single numeric value in an HTML table, structured data in XML or JSON (Javascript Object Notation), or a form value.

This technique gives Google Mail the ability to use workspace at the center of the browser to display an editor where the user can compose a message, display a message, or display a list of messages (Figure 2).

Other good examples of AJAX are the Web 2.0 apps Flickr and Del.icio.us.

## Quick Start

To get us started, let's take a look at a very simple web application. When the user clicks, we want the browser to download a random number from the server and display the number at a predefined position on the web page.

The application comprises three components: the static HTML front page, the Javascript code, and a server-side script that returns the results (the random number). One thing you definitely need for this is a web server (typically Apache) that supports a scripting language – this will be PHP in our example. However, you can run the server and the browser on the same machine – any normal desktop will do – to try AJAX out.

If installing and configuring Apache and PHP with your distribution's standard tools is too complex, just pick one of the popular LAMP or XAMPP packages. The packages include Apache and PHP, along with the MySQL database, which, strictly speaking, you do not need, although it can come in handy for dynamic AJAX applications.

The HTML file in the AJAX sample application has a simple structure (Listing 1). The script tag in the header section points to an external Javascript file, titled *ajax.js*, which contains the code for the AJAX application. As an alternative to this, you can insert Javascript functions between the opening and closing script tags in the body of the HTML file.

Line 6 in Listing 1 links the code and the HTML. The link's *onclick* attribute stores the name of the Javascript function called by the browser (*getRandom*)

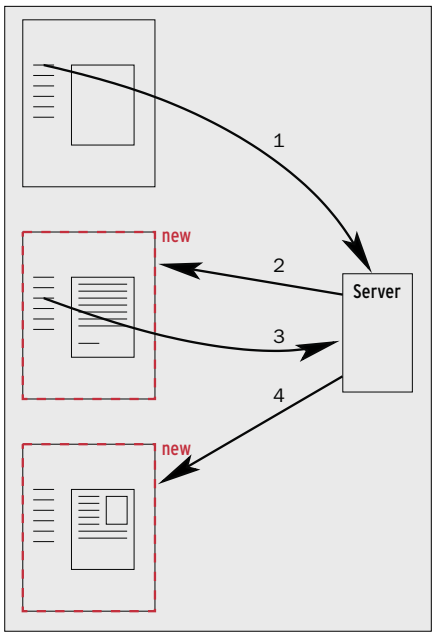


**Figure 1: The legacy approach: the browser reloads the page completely whenever a change occurs.**

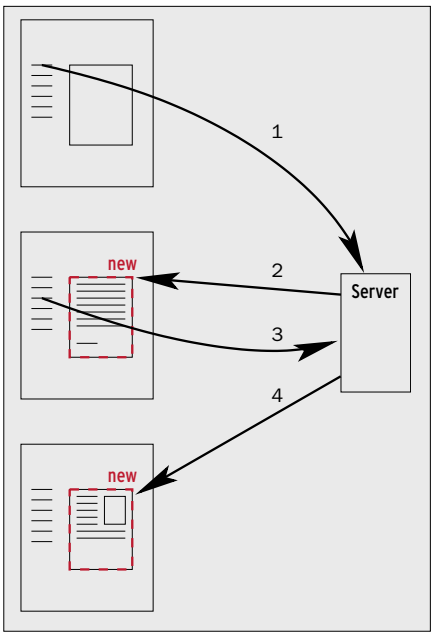


**Figure 2: The AJAX model: requests only reload the parts of a page that have actually changed.**

when the user clicks it. This is followed by an HTML *span* with an ID, which the Javascript will reference later.

If you want to test whether the HTML/Javascript connection is working, add the following code to the file:

```
function getRandom()
{
   alert("Clicked");
}
```

Assuming that *index.html* and *ajax.js* are located in the same server directory, *ajax-test* for example, the URL for the HTML page would be *http://Servername/ajax-test/*. If the server and browser are running on the same machine, use *localhost* as the server name. Now, when you click the link, the browser pops up a dialog that displays a text message of *Clicked*.

If this doesn't work, at least you can practice troubleshooting. The Firefox browser is a good choice for development work with AJAX, as it includes a collection of practical tools. For example, the *Tools* menu has a *JavaScript Console* item that pops up a small window for Javascript error messages.

A simple application like this does not have many potential sources of error. The server may not be able to locate the Javascript file, or the problem could be a syntax error.

## Connecting with the Server

Thus far, the browser has sent one request each for the HTML page and the

Javascript file to the server. This completes the communication. The two do not contact each other when a user clicks a link. To retrieve more data from the server, the Javascript code first has to create a request object. The *XMLHttpRequest()* handles this and also explains the X in AJAX. The following command assigns the new request object to the *request* variable:

```
request = ➥
new XMLHttpRequest();
```

The request object has a number of methods that are important for communicating with the server later on. First of all, *request.open()* sets the connection parameters.

The first parameter specifies the HTTP method (GET or POST); this is followed by the web address (URL) to contact. The next parameter is for asynchronous access with a value of *true*) in our case. Two optional parameters can pass in the username and password for password-protected pages. Assuming an address of *http://localhost/~oliver/ajax/test.php* in the Javascript variable *url*, the following line sets up the connection:

```
request.open("GET", url, true);
```

Before you send the request to the server, you first have to specify the function the browser calls when it receives the response. You may recall that the browser will not wait for the server response, as communications between the browser and the server are asynchronous. The request object's *onreadystatechange* field is used to specify the callback function.

As the name would suggest, the browser does not just call the callback function when it receives a response, but whenever the request object state has changed. Five states are defined for the request object, ranging from unused (0) to finished (4).

### Listing 1: HTML with AJAX

```
01 <html>
02   <head>
03     <script
   language="JavaScript"
   type="text/javascript"
   src="ajax.js"/>
04   </head>
05   <body>
06     <a href="#"
   onclick="javascript:
   getRandom()">AJAX-Test</a>
07     <span id="random"> </span>
08   </body>
09 </html>
```

### Listing 2: The Complete Listing

```
01 function checkResult()
02 {
03   alert("New state: " +
   request.readyState);
04 }
05
06 function getRandom() {
07    request = new
   XMLHttpRequest();
08    var url = "http://
   localhost/~oliver/ajax/index.
   html";
09    request.open("GET", url,
   true);
10    request.onreadystatechange
   = checkResult;
11    request.send(null);
12 }
```

## Waiting for Responses

Finally, let's use the *send* method to send the request to the server; the method can handle any payload data you may have as optional parameters, giving you the ability to insert user input into forms, for example.

Our simple example does not have any payload data; this is why we are passing in a *null* parameter to *send*.

Listing 2 shows the complete listing for the simple AJAX application. The callback function occurs in Line 10, *checkResult()*; it outputs the current request status in a dialog whenever it is called.

Because the browser doesn't actually do anything with the server response, there is no need to go to the trouble of writing a PHP script. For demonstration purposes, you can simply use the index page to issue the request, as shown in Listing 2. This solution does not influence the way the request object or the callback function are used.

If you don't see a dialog, it's back to troubleshooting: a typo in the URL variable, possibly? Copy the string (without the quotes), insert it into the address box in another browser window, and press the Enter key. If the server responds with an error message, compare the variable with the filename on the server once again.

The security settings for Javascript in your browser are another potential source of error: the request object can only contact the server that served up the original HTML page. If the server addresses differ, the browser will assume a security infringement and issue a "Permission denied" message.

A Firefox extension titled Firebug can be useful for troubleshooting AJAX applications [2]. It not only shows you the Javascript errors but can give you every single *XMLHttpRequest* with the header fields and response code.

## Payload

Of course, these simple examples are not exactly what AJAX's inventor intended. Background server requests are designed to merge dynamic content with the current website. The PHP script in Listing 3 implements a test service for this purpose, issuing a random number between 1 and 100 for each request.

If you store the script as *random.php* in the same server directory as the other files, you need to change the *url* variable in *ajax.js* to match. This gives you the following line:

```
var url = "http://localhost/➥
~oliver/ajax/random.php";
```

To tell the browser to load the modified Javascript code, just click the reload button. Now, when you click the link, the dialog windows will display the request object status.

Of course, you have to complete the transfer to access the server data – this is request object state 4 – so you might like

### Listing 3: Random Number PHP Script

```
01 <?
02   srand(time());
03   $random = (rand()%100);
04   print $random;
05 ?>
```

to have the callback handler check for the terminal state on every state change, and not process the payload until the object reaches this state.

You can use the request object's *readyState* variable to read the data. The *responseText* stores the payload data in ASCII clear text. The *checkResult()* function now looks like this, as you can see in Listing 4.

Now, when you reload the page in your browser and click the link, a dialog appears with the random number generated by the server.

## Updating the Page

This completes two of the three steps to creating an AJAX application. We have used a request object to send a request to the server, received a response, and processed the response. That just leaves updating the web page itself. We want to display the payload data at the assigned position, and to format the data in an attractive way, rather than just popping up a dialog box.

Again, Javascript code will take care of this. The page make up is represented browser-side by the Document Object Model (DOM, see Figure 3). The document tree structure gives us the ability to reference, read, and modify any HTML element in Javascript. You can supply code to insert elements into the tree; the elements are then displayed on the HTML page. The Firefox *Tools | DOM Inspector* function lets you inspect the document tree (Figure 3).

For the time being, let's just modify an existing element to display the results of the AJAX query. Modifying an element is very simple if the element we want to change has a unique ID, which you can use to reference it in Javascript. We assigned an ID of *random* to the span element in the HTML file shown in Listing 1.

**Listing 4: The checkResult() function**

```
01 function checkResult()
02 {
03   if (request.readyState == 4)
   {
04     alert("Response: " +
   request.responseText);
05   }
06 }
```

The *getElementById()* Javascript function for the complete document provides us with a reference to the HTML element. The command that follows sets the HTML elements *innerHTML* property to the required random value.

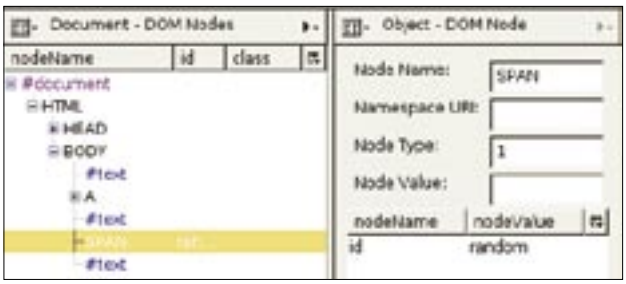


**Figure 3: Select Tools | DOM Inspector in Firefox to inspect the Document Object Model of an HTML page.**

```
var randomDiv = ➥
document.getElementById➥
("random");
randomDiv.innerHTML = ➥
request.responseText;
```

Just add these two lines to *checkResult()*, in place of the *alert* function, to complete the AJAX application: the Javascript code writes the results directly to the HTML page, instead of reloading the whole page.

## Incompatibility

At least that's what happens in theory, but as is so often the case in development work, reality is a different matter. Each browser does its own thing, and AJAX programmers just have to cope with these quirks.

Let's start with Internet Explorer – current versions don't implement the XML Request object, or at least not in a way that gives you the required results when you call *XMLHttpRequest()*. Microsoft expects you to implement an ActiveX object instead, but luckily it does at least react in the same way as a request object apart from this. In fact, another variant is actually required, as different versions of Internet Explorer use different syntax.

KDE's Konqueror and the Apple Safari browser are two more candidates, as some Javascript constructs used to manipulate the DOM tree can cause trouble. For example, the procedure we just looked at, for accessing an element you want to change via the *getElementById()* document function, will not work. To save you work, AJAX examples with various browser workarounds are available online from [3].

The *checkResult()* function in the listing also demonstrates how to make the interaction more dynamic. As shown in the last variant, it waits until the request has been completed (state 4) before setting the span element to the result. For all other state changes, and they start when the connection is established, it changes the text to *Loading...* to give the user some feedback on what is going on.

## Help on the Web

As you can see, AJAX web applications take slightly more programming effort than conventional websites, starting with the design: not every website is suitable for "ajaxifying." Good planning is even more important than for legacy web development. Finally, you need a server-side script for every AJAX section on a page. And the Javascript code for every AJAX application needs to know exactly how the pages are structured. Cascading Stylesheets (CSS) can provide an improved structure, however, this means adding even more files.

The numerous Javascript and AJAX libraries on the Internet are useful. Sajax [4] abstracts AJAX requests and response processing, removing worries about browser incompatibility. Rico [5], and the Yahoo User Interface Library (YUI) [6] add more convenience; both include functions for creating dynamic HTML interfaces. ■

## INFO

[1] AJAX: A New Approach to Web Applications: *http://adaptivepath.com/publications/essays/archives/000385.php*

[2] Firebug: *http://www.joehewitt.com/software/firebug*

[3] Listings for this article: *http://www.linux-magazine.com/Downloads/2006/12/ajax*

[4] Sajax: *http://www.modernmethod.com/sajax*

[5] Rico: *http://openrico.org*

[6] YUI: *http://sourceforge.net/projects/yui*