CHAPTER 1

The State of JavaScript

This chapter takes a brief walk down memory lane so you can get a sense of how the industry has changed over the last decade, including the rise of Ajax and its influence on the popularity of JavaScript. It then explains how JavaScript gets evaluated in the browser and how to plan for that. You'll learn ways to debug applications and the tools you can use to do so. It's important to understand how your code is working to fix those pesky bugs that haunt you.

JavaScript Is One of the Good Guys Again, but Why Now?

JavaScript has come a long way since its inception back in 1995. Initially used for basic image and form interactions, its uses have expanded to include all manner of user interface manipulation. Web sites are no longer static. From form validation, to animation effects, to sites that rival the flexibility and responsiveness traditionally found in desktop applications, JavaScript has come into its own as a respected language. Traditional (and expensive) desktop applications such as word processors, calendars, and e-mail are being replicated in cheaper (and often easier-to-use) Web-based versions such as Writely, 30 Boxes, and Google Mail.

Over the course of 10 years, the popularity of JavaScript has increased and waned; fortunately, it is now making its triumphant return. But why now? One word: *ubiquity* ("the state of being everywhere at once"). The goal of most developers has been to have the work they produce be available and accessible to everyone. HTML accomplished this goal early on. Much of the format matured before the Internet really took off in the late 1990s. The HTML you produced for one browser would appear mostly the same in all other browsers: Mac, PC, or Linux.

JavaScript was still quite immature, however. Its capability to interact with the HTML document was inconsistent across browsers. Its two main facilitators, Netscape and Internet Explorer (IE), implemented very different approaches, which meant that two completely different implementations were required to complete the same task. People often tried to create helper scripts, or sometimes even full-blown JavaScript libraries, to bridge the gap. Keep in mind that JavaScript libraries weren't that popular back in the day. Most saw them as bloated and unnecessary to achieve what they needed. The libraries certainly eased development, but they were large in comparison with the problems people were trying to solve with JavaScript. Remember that broadband certainly wasn't what it is today. Tack bandwidth concerns onto security concerns and entire companies disabling JavaScript outright, and you have a situation in which JavaScript seemed like a toy language. You had something that seemed the Web could do without.

With IE a clear victor of the "browser wars," Netscape languished. You might have concluded that developers would develop only for IE after it garnered more than 90 percent of the market. And many did (including me). But that ubiquity still didn't exist. Corporate environments and home users continued to use Netscape as a default browser. Clients I worked with still demanded Netscape 4 compliance, even heading into the new millennium. Building any sort of cross-browser functionality was still a hassle except for processes such as form validation.

The World Wide Web Consortium (W3C), which included partners from many of the browser developers, continued to update and finalize much of the technologies in use today, including HTML/XHTML, Cascading Style Sheets (CSS), and the document object model (DOM).

With standards in place and maturing, browser developers had a solid baseline from which to develop against. Things began to change. When Mozilla Firefox finally came out in 2004, there was finally a browser that worked across multiple operating systems and had fantastic support for the latest HTML/XHTML, CSS, and DOM standards. It even had support for nonstandard technologies such as its own native version of the `XMLHttpRequest` object (a key ingredient in enabling Ajax, which is covered in Chapter 5). Firefox quickly soared in popularity, especially among the developer crowd. The W3Schools web site, for example, shows recent Firefox usage at almost 34 percent (see `http://w3schools.com`, May, 2007).

■ **Note** Take browser statistics with a grain of salt. As the saying goes, there are lies, damned lies, and statistics. Every site is different and attracts a certain demographic, so you can expect your stats to differ from everybody else's. For example, 60 percent of those who visit my site, with its heavy skew toward developers, use Firefox. This speaks heavily to the need to build sites that work on all browsers because you never know what your users will have or how the market might shift.

Apple released Safari for the Mac, which filled the gap when Microsoft decided to discontinue developing a browser for the Mac platform. Safari, along with Firefox and Camino (based on the Gecko engine that Firefox uses), had solid support for HTML and CSS standards. Early versions of Safari had limited DOM support, but recent versions are much easier to work with and also include support for `XMLHttpRequest`. Most importantly, they all support the same set of standards.

The differences between the current versions of the browsers on the market became minimal, so you have that ubiquity you've been looking for. The reduced set of differences between browsers meant that smaller code libraries could be developed to reduce the complexity of cross-browser development. Smart programmers also took advantage of JavaScript in ways that few had done before. JavaScript's resurgence is here!

Google demonstrated that JavaScript-powered applications were ready for the mainstream. Google Maps (`http://maps.google.com/`) and Google Suggest (`www.google.com/webhp?complete=1`) were just two of many applications that showed the power, speed, and interactivity that could be achieved.

JavaScript Meets HTML with the DOM

Although this discussion is about JavaScript and its evolution, it's the DOM (which has evolved immensely from its early days) that takes center stage in the browser. Netscape, back in version 2 when JavaScript was invented, enabled you to access form and image elements. When IE version 3 was released, it mimicked how Netscape did things to compete and not have pages appear broken.

As the version 4 browsers were released, both browsers tried to expand their capabilities by enabling ways to interact with more of the page; in particular, to position and move elements around the page. Each browser approached things in different and proprietary ways, causing plenty of headaches.

The W3C developed its first DOM recommendation as a way to standardize the approach that all browsers took, making it easier for developers to create functionality that worked across all browsers—just like the HTML recommendations. The W3C DOM offered the hope of interactivity with the full HTML (and XML) documents with the capability to add and remove elements via JavaScript. The DOM Level 1 recommendation is fairly well supported across Mozilla and IE 5+.

The W3C has subsequently come out with versions 2 and 3 of the DOM recommendations, which continue to build on the functionality defined in level 1. (Differences between the DOM versions are covered in Chapter 2.)

The Rise of Ajax

The term *Ajax*, which originally stood for Asynchronous JavaScript and XML, was coined by Jesse James Garrett of Adaptive Path (`www.adaptivepath.com/publications/essays/archives/000385.php`). It was meant to encapsulate the use of a set of technologies under an umbrella term. At the heart of it is the use of the `XMLHttpRequest` object, along with DOM scripting, CSS, and XML.

`XMLHttpRequest` is a proprietary technology that Microsoft developed in 1998 for its Outlook Web Access. It is an ActiveX object that enables JavaScript to communicate with the server without a page refresh. However, it wasn't until the rise of Mozilla Firefox and its inclusion of a native version of `XMLHttpRequest` that it was used on a large scale. With applications such as Google Mail starting to take off, other browser developers quickly moved to include it. Now IE, Firefox, Opera, and Safari all support a native `XMLHttpRequest` object. With that kind of ubiquity, it was only inevitable to see the technology take off. The W3C has now moved to try and establish a standard for Ajax (see `www.w3.org/TR/XMLHttpRequest`).

■ **Note** ActiveX is a Microsoft technology that enables components within the operating system to communicate with each other. Using JavaScript with ActiveX, you can actually interact with many applications stored on the client's machine (if installed). For example, given a loose security setting, you can open Microsoft Office applications, interact with them, and even copy data out of them—all from a web page. The same can actually be done with any application that offers a component object model (COM) interface.

I mentioned XML as being one of the core tenets of Ajax, and you might wonder how XML fits into all this. As Jesse James Garrett originally describes, Ajax incorporates XML as a data interchange format, XSLT as a manipulation format, and XHTML as a presentation format. While XML was originally described as a major component of Ajax, that strict description has loosened and now describes the process of communicating with the server via JavaScript using the `XMLHttpRequest` object and the many technologies that are involved in implementing a site or an application using Ajax (such as HTML and JSON).

Ajax enables communication with the server without requiring a page refresh. But what does that mean to you? It gives you the ability to perform asynchronous actions (hence the first *A* in Ajax). You can perform form validation before the form has even been submitted. For example, have you ever tried signing up for a service only to find that the user ID was already taken? You'd hit the Back button, try a different name (and retype your password because it is never retained), and resubmit. This cycle would annoyingly repeat itself until you found an unused name. With Ajax, you can check the user ID while the user is completing the rest of the form. If the name is taken, an error message displays to the user, who can fix it before submitting the form.

With this new power, developers have been pulling out all the stops to build some dazzling applications. Alas, many are more glitz than guts; more pizzazz than power. While you might find yourself wanting to add the latest trick, it will always be important to think about usability and accessibility in all you put together. This topic will be discussed throughout the book.

Managing JavaScript

These days, JavaScript-based applications can get large and unwieldy. Before you get into any JavaScript, I want to talk about where to place code in an HTML page and the best approaches for long-term maintenance. There are some nuances that are important to remember when testing and evaluating your own code.

Code Loading

The first process to understand is the loading process. When an HTML page loads, it loads and evaluates any JavaScript that it comes across in the process. Script tags can appear in either the `<head>` or the `<body>` of the document. If there's a link to an external JavaScript file, it loads that link before continuing to evaluate the page. Embedding third-party scripts can lead to apparent slow page load times if the remote server is overburdened and can't return the file quickly enough. It's usually best to load those scripts as close to the bottom of the HTML page as possible.

```
<head>
<title>My Page</title>
<script type="text/javascript" src="myscript.js"></script>
</head>
```

Scripts that you build should appear at the head of the document and need to be loaded as soon as possible because they'll probably include functionality that the rest of the page relies on.

Code Evaluation

Code evaluation is the process by which the browser takes the code you've written and turns it into executable code. The first thing it will do is test to see whether the code is syntactically correct. If it isn't, it will fail right off the bat. If you try and run a function that has a syntax error (for example, a missing bracket somewhere), you'll likely receive an error message saying that the function is undefined.

After the browser has ensured that the code is valid, it evaluates all the variables and functions within the script block. If you have to call a function that's in another script block or in another file, be sure that it has loaded before the current script element is loaded. In the following code example, the `loadGallery` function still runs, even though the function is declared after the function call:

```
<script type="text/javascript">
loadGallery();

function loadGallery()
{
   /* gallery code */
}
</script>
```

In the following example, you'll get an error message because the first script element is evaluated and executed before the second one:

```
<script type="text/javascript">
loadGallery();
</script>

<script type="text/javascript">
function loadGallery()
{
   /* gallery code */
}
</script>
```

My general approach is to include as much of my code in functions and load them in from external files first; then I run some code to start the whole thing up.

Embedding Code Properly into an XHTML Page

Embedding JavaScript on an HTML page is easy enough, as you saw in the previous examples. Many online examples usually include HTML comment tags to hide the JavaScript from browsers that don't recognize JavaScript.

```
<script type="text/javascript">
<!--
/* run my code */
loadGallery();
//-->
</script>
```

However, the days of someone using a browser that doesn't recognize JavaScript are long gone, and HTML comments are no longer necessary.

XHTML, however, is a different beast. Because it follows the rules of XML, the script has to be enclosed into a `CDATA` block, which starts with `<![CDATA[` and ends with `]]>`.

```
<script type="text/javascript">
<![CDATA[
/* run my code */
loadGallery();
]]>
</script>
```

■ **Note** Throughout the book, I'll be using HTML almost exclusively; if you prefer to use XHTML, you'll need to keep this in mind.

Debugging Your Code

It doesn't matter how simple your code is, you are guaranteed to have errors in your code at some point. As a result, you'll need to have a way to understand what went wrong, why it went wrong, and how to fix it.

Alert

Probably the most common technique of JavaScript debugging is using `alert()`. There's no software to install and no complicated code to set up. Just pop a line into your code, place the information you're looking for into the alert and see what comes up:

```
alert(varname);
```

An alert is ineffective, however, for tracing anything that is time sensitive or any values within a loop. If something is time sensitive (for example, an animation), the alert throws things off because it has to wait for your feedback before continuing on. If it's a loop, you'll find yourself hitting the OK button countless times. If you accidentally create an infinite loop, you have to force the browser to close entirely, losing anything else that was open, to regain control of it—and that's never fun!

Alerts can also be ineffective because they show only string data. If you need to know what's contained within an array, you have to build a string out of the array and then pass it into the alert.

Page Logging

Page logging is a handy trick and a step above using an alert. Create an empty `<div>` on the page and use absolute positioning along with setting the overflow to scroll. Then, any time you want to track some information, just append (or prepend) the value into your `<div>`.

The script is as follows:

```
function logger(str){
   var el = document.getElementById('logger');
   // if the logger container isn't found, create it
   if(!el) {
     el = document.createElement('div');
     el.id = 'logger';
     var doc = document.getElementsByTagName('body')[0];
     doc.appendChild(el);
   }
   el.innerHTML += str + '<br>';
}
var value = 5;
logger('value = ' + value);
```

The CSS used to give the element a little style and to ensure that it doesn't interfere with the layout is as follows:

```
#logger {
   width:300px;
   height:300px;
   overflow:scroll;
   position:absolute;
   left:5px; top:5px;
}
```

Others have produced some elaborate and useful loggers that work in the same vein. Over at A List Apart, on online web magazine, there's an article on fvlogger (`http://alistapart.com/articles/jslogging`). Also, check out the project log4javascript at (`www.timdown.co.uk/log4javascript`). The log4javascript project uses a separate window to log messaging, which can be handier because it's not in the way of the current document.

Browser Plug-ins

Browser plug-ins are often beautifully crafted applications that can give you thorough minutiae on not only JavaScript but also on the HTML and CSS rendered on the page. They can be a lifesaver for learning what is actually happening on your page. On the downside, they're almost always browser-specific. That means that testing in some browsers might prove more difficult, especially if the problem is happening only in that browser.

DOM Inspector

When it comes to JavaScript development, Firefox is one of the best browsers to develop for. Its DOM support is certainly one of the best, if not the best, and it also has some of the best tools for troubleshooting. Built right in is the DOM Inspector, as seen in Figure 1-1.

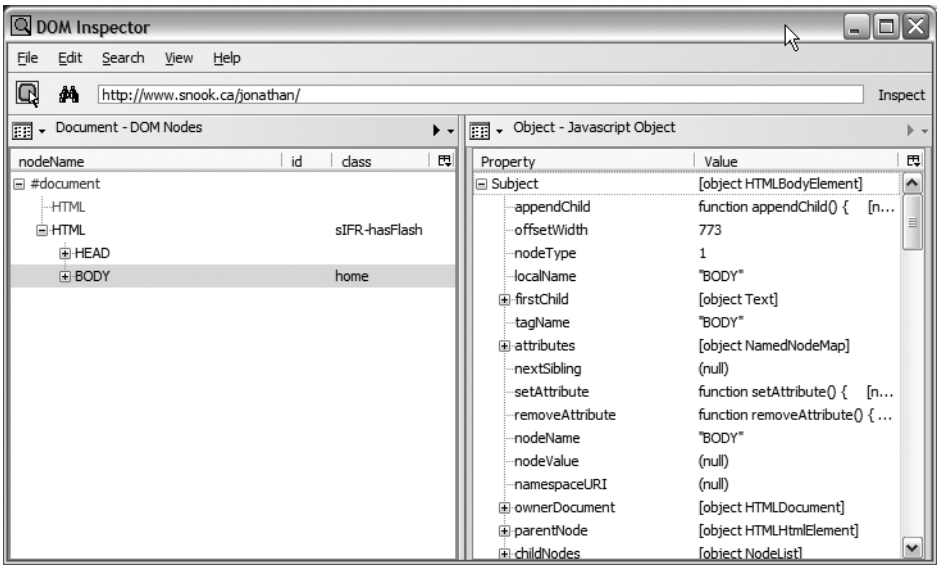

Figure 1-1. *The Firefox DOM Inspector*

With the DOM Inspector, you can navigate the document tree and view the various properties for each one. In the screenshot, you can see the properties that you can access via JavaScript. In addition, there are views for seeing which styles have been set, along with the computed values, which are handy for seeing why a layout has gone awry.

Firebug

Firebug (`www.getfirebug.com`) is currently the reigning champion of JavaScript and CSS debugging tools. It is by far the most powerful and flexible tool to have in your arsenal.

Firebug takes the DOM Inspector to a whole new level. Once installed, the interface panel is accessible from the status bar. The icon (see Figure 1-2) indicates whether you have any errors on the current page.

Figure 1-2. *The Firebug check mark icon*

Clicking the icon expands the interface. There's a lot of functionality packed into it, and while I won't go into everything, I do want to highlight some key features that will help in your debugging efforts.

In Figure 1-3, the Console tab is selected. JavaScript error messages, Ajax calls, Profile results, and command-line results appear in the console. Objects can be expanded to view properties, error messages can be clicked to view the offending line in the source, Ajax calls can be expanded to view request and response information, and profile results can be analyzed to discover where errors might be occurring.

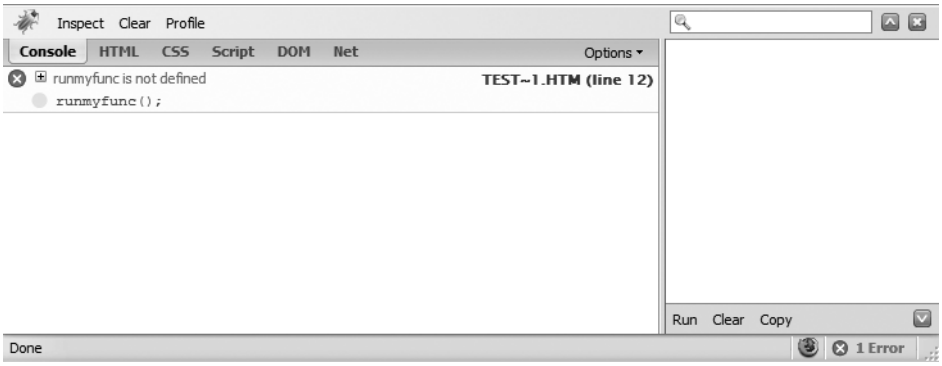

Figure 1-3. *The Firebug console*

The HTML, CSS, and Script tabs enable you to view the current state of each of those elements. You can also make changes and view them live in the Firefox window. Keep in mind that those changes are only temporary and will be lost when you refresh the page or close the window. The original files are never touched.

The DOM tab enables you to view the DOM tree and all its properties. The Net tab, as seen in Figure 1-4, shows all file requests and how long each took to load. You can use this information to determine where certain bottlenecks might be occurring.

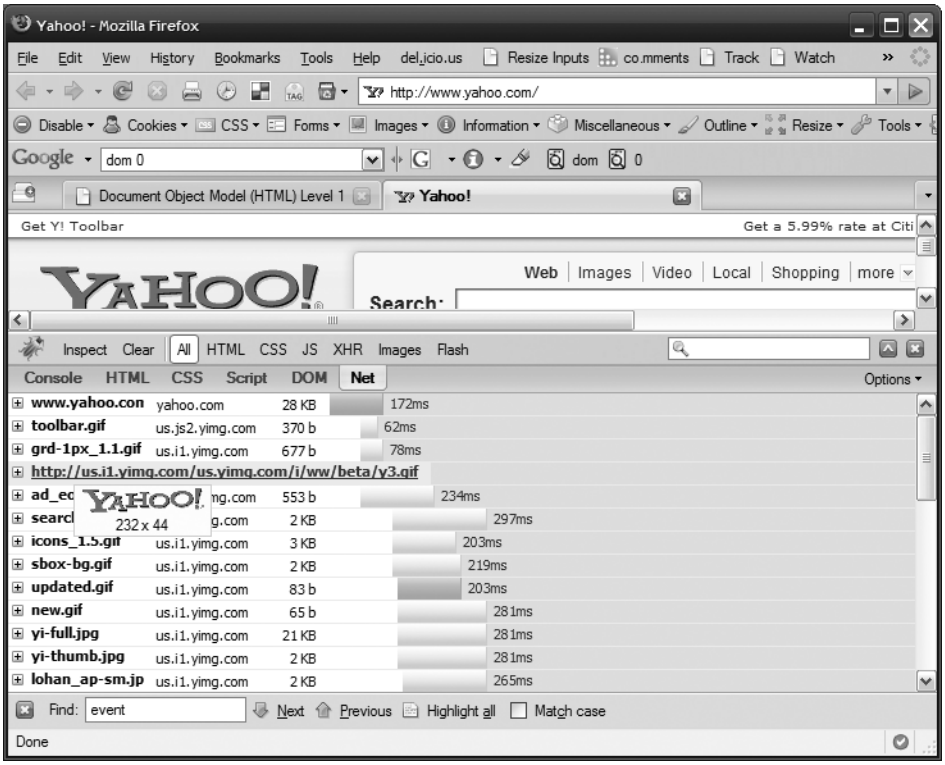

Figure 1-4. *The Net tab in Firebug*

On the main toolbar is the Inspect button, which is very useful, and you will probably use it constantly (at least, I do!). When you click the button, you can then move your mouse anywhere on the HTML page. Firebug highlights which element you are currently hovering over. It also highlights that element in the HTML tab.

With the current element selected in the HTML tab, you can see the applied style information in the panel on the right (see Figure 1-5). You can even see when certain styles have been overwritten by other styles. So as you can see, the power of Firebug extends well beyond just JavaScript.

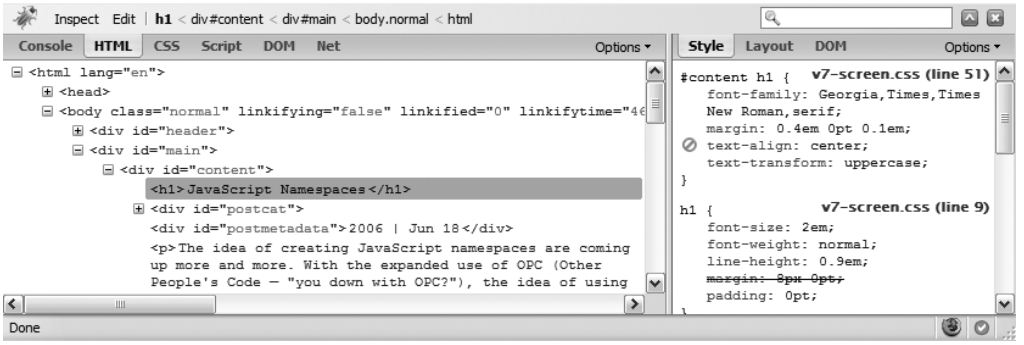

Figure 1-5. *Selected element in Firebug*

Instead of using alert statements or page logging, there are a number of hooks that Firebug adds that enable you to log information right into the Firebug console. The one I use most often is `console.log()`, which works exactly like the logger function discussed earlier, but doesn't disturb the current page—it just loads the information into the console. If you're tracing an object, you can click that object in the console and inspect all the properties of that object.

There are plenty of other features stored within Firebug, and a whole chapter could probably be written just on the gems contained within. I'll leave it up to you to discover those jewels.

HTTP Debugging

Everything you do on the Web runs over HTTP, which is the protocol that sends the packets of information back and forth.

Particularly with Ajax calls, but also useful with any server/client interaction, you'll want to see what information is actually getting sent or received. You can sometimes log this information from the back end, but that doesn't always paint a true picture of what's happening on the front end. For that, you need an HTTP debugger.

Firebug

As further evidence of its coolness, the debugger in Firebug traces Ajax calls, enabling you to view the request and the response headers, as shown in Figure 1-6. This is handy to ensure that you're receiving the correct data back. You can inspect the call to also see what data has been posted to the server and what the server returned.

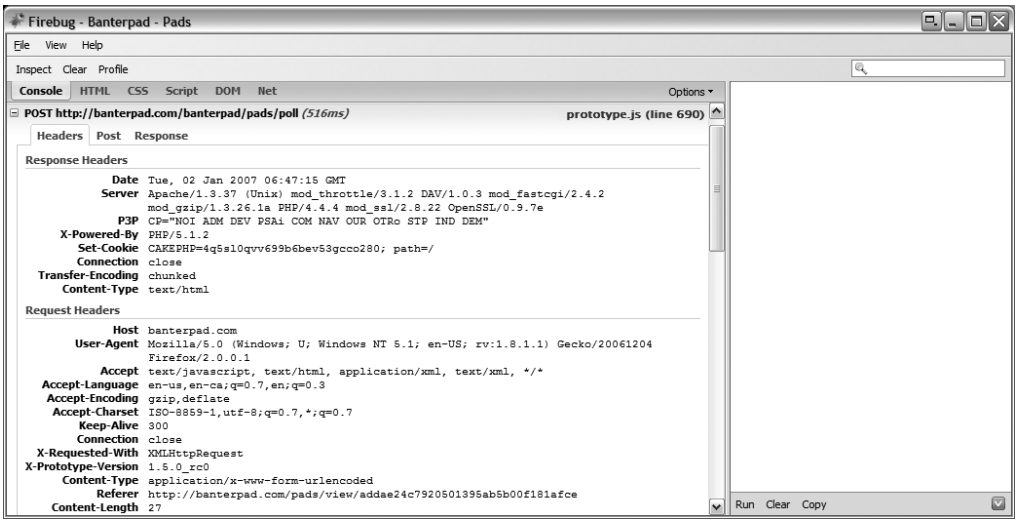

Figure 1-6. *Firebug Ajax call inspection*

Live HTTP Headers

For more fine-grained analyses of HTTP requests, I recommend that you grab Live HTTP Headers from `http://livehttpheaders.mozdev.org`. This helpful Firefox extension displays request and response info for *all* HTTP requests, which can be handy for not only Ajax calls (such as the one seen in Figure 1-7) but also monitoring page requests (including form data), redirects, and even server calls from Flash. It also enables you to replay specific requests. Before replaying data, you can even modify the headers that are being sent to test various scenarios.

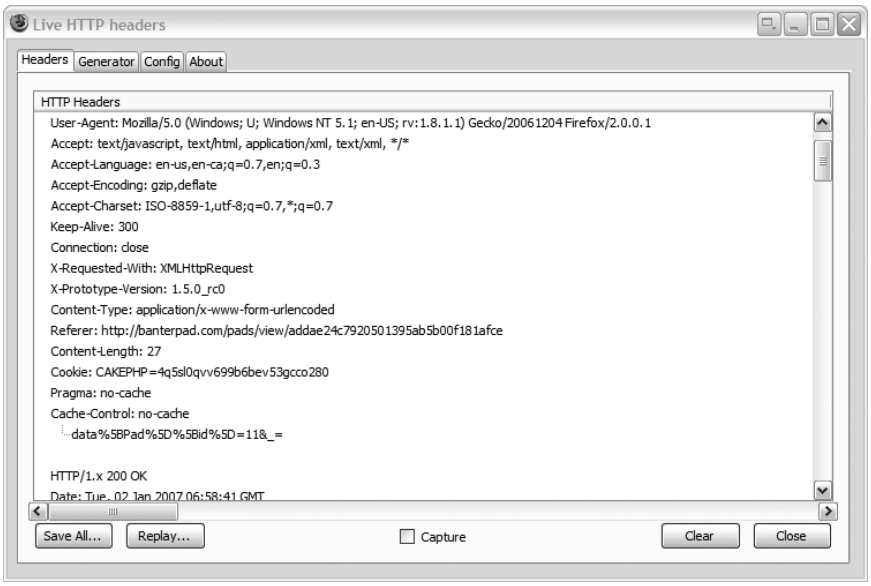

Figure 1-7. *Live HTTP Headers Ajax call inspection*

Firebug reveals more response information, so you might have to bounce between it and Live HTTP Headers to get a better picture of what's going on.

ieHTTPHeaders

IE similarly has an add-in called ieHTTPHeaders to analyze the information going back and forth. It's available at `www.blunck.info/iehttpheaders.html`.

Charles

Probably the most robust HTTP debugging tool of the bunch is Charles. This debugger is shareware, so you'll have to spend a little money to include it in your toolbox—but it's money well spent for more than just tracing Ajax calls.

Charles can provide a number of useful tasks, such as bandwidth throttling for testing slow connections and spoofing DNS information for testing under a domain name before it goes live. It can automatically parse the AMF format that Adobe Flash uses for remote calls, and can parse XML and JSON used in Ajax calls. (Data exchange using XML and JSON is discussed in Chapter 5.)

The other nice thing about Charles is that it is browser-agnostic. It works as a proxy server and tracks everything through there, so you can use it with all your browsers. It's even available for Mac OS X and Linux users. (You can grab it from `www.xk72.com/charles`.)

Summary

This chapter discussed the following topics:

- Why JavaScript is becoming more popular
- How JavaScript gets evaluated in the browser
- What tools you can use to debug your code

After the quick "how-do-you-dos," you should now have a sense of why JavaScript has become the superstar it is today. You have some understanding of the things to consider when putting code into your page and have all the tools necessary to run and test the code you'll be developing from here on out. You're all set to become a JavaScript ninja!