Maximizing JavaScript and Ajax performance

Measure performance with Firebug, Safari Web Inspector, YUI Profiler, and YSlow

Joe Lennon

With Ajax and JavaScript becoming more prominent in modern web applications, it is becoming increasingly important to ensure that your JavaScript code and Ajax requests are finely tuned to provide optimal performance. In this article, learn the best practices for writing JavaScript code and performing Ajax requests. Learn how to measure the performance of your existing applications and how to identify common bottlenecks in your code. And finally, learn how to fix performance-related problems and how to use the various tools that are at your disposal when it comes to maximizing the speed of your application.

Introduction

In the early days of the web, maximizing the performance of a web page usually meant avoiding the use of unnecessary HTML markup, keeping the amount of JavaScript code to a minimum, and heavily reducing the file size of any images so that the typical surfer didn't have to go away and make a cup of coffee while your page loaded.

Advancements in various aspects of the web have meant that we are now faced with a whole new set of performance considerations. Despite the fact that DSL and broadband have made high-speed Internet access far more accessible to many people, our expectations of load time and responsiveness have also evolved to a state where we look for instant results when we perform an action on a page. The emergence of Asynchronous JavaScript and XML (Ajax) allowed developers to deliver a desktop-like experience in web applications, no longer requiring entire pages to load before responding to an event. The advantages of this are obvious, but it also led to the average web user expecting this type of responsiveness from all web applications. With the recent rise of the mobile web there is a new challenge of meeting a modern web user's expectations, and all of this on a target device with a smaller screen, less power, and slower connectivity.

The focus of this article is on informing you of the considerations you should take to maximize the performance of your JavaScript and Ajax web applications. This article offers a set of guidelines on how to best approach any new code you are writing, whether it be in a new application or an existing one. You will also learn about the various tools and techniques that are out there for
measuring the performance of your application. Finally, you will learn about various methods of improving performance that do not require you to change your existing code.

Best practices for JavaScript and Ajax development

One of the problems with JavaScript development is that a high percentage of the developers and web designers who write JavaScript code have never actually learned JavaScript itself from the ground up. Their knowledge of the language is usually amassed from years of adding snippets they have found on the web that perform a particular function. They will usually have an understanding of how to declare variables, write conditional statements, and perform calculations, but they have never sat down with a comprehensive guide to the JavaScript language and learned it from scratch. Today, this problem persists, as developers rush to libraries and frameworks like jQuery and YUI to make their lives easier.

While there is nothing wrong with using JavaScript libraries (I am, in fact, a big fan of them.), there is a tendency among modern developers to be experts in their JavaScript framework of choice, but not in JavaScript itself. The problem with this is that you will often find these developers using highly inefficient coding practices, sometimes even using a framework feature to do something that would be much faster using regular old JavaScript.

In this section, you will learn about some best practices for JavaScript and Ajax development, particularly in areas that tend to be overlooked by non-JavaScript developers.

Use external JavaScript files

The golden rule for maximizing the performance of JavaScript applications is to use, where possible, external JavaScript files rather than including JavaScript code in your HTML files directly. Not only does doing so mean that you won’t have to duplicate your JavaScript code across multiple pages, but it also means that your JavaScript code will be cached by the web browser and won’t need to be loaded repeatedly with each subsequent page load. The first page load will be significantly slower, as the external file requires an additional HTTP request to be sent to the server. In the majority of applications, however, the hit on performance associated with this is more than made up for by the subsequent performance savings gained from subsequent page loads.

The exceptions to this rule are where the majority of your visitors only view one page. This is where using inline JavaScript is preferred; or where you need the first page load to be as fast or faster than subsequent page loads. In his book, High Performance Web Sites (see Related topics for a link), Steve Sounders puts forward the idea of the Post-Onload Download, where you keep the JavaScript code for the first page inline in the HTML file itself, and then dynamically load the external JavaScript files required for subsequent pages after the page has completely loaded. In the majority of cases, however, simply using external JavaScript files should be sufficient.

When to use JavaScript frameworks and libraries

I'm all for using JavaScript frameworks and libraries. Not only do they help solve many cross-browser compatibility issues, but, when used properly, they can vastly speed up the development time for web applications. With that said, great care should be taken when using these tools, because the majority of them are quite large and can impair the performance of your application.
The first thing you need to ask yourself is: Do I really need to use a framework? My introduction to JavaScript frameworks came a few years ago when I needed to use Ajax in a web application I was developing. Rather than roll my own XMLHttpRequest function, I decided to employ the Prototype framework to make my life easier. The application made no use of the framework other than to perform Ajax requests and handle the responses from the server, but I used it anyway. Fortunately for me, the application I was developing was relatively small and for internal use, so performance wasn’t all that critical, but years on I know that I would probably have been better off using a lighter solution that provided Ajax functionality alone.

The latest version of the Prototype framework, unminified and uncompressed, is 141KB in size. The part of the code that was relevant to my application probably amounted to less than 2KB, leaving about 139KB of JavaScript code in my application that was completely unused. Not only would this have increased the load time of my application in terms of file size, but it would also have increased the execution time as the JavaScript code executed in the browser.

To summarize my point, modern JavaScript frameworks and libraries such as Prototype, jQuery, Dojo, MooTools, YUI, ExtJS, and so on all contain a plethora of features that you may or may not be using. If you are only using a minimal set of features, it might be worth looking into a lighter solution. The YUI library, for example, allows you to load the bare minimal framework by default and then choose which libraries you need to load on top of it.

**Script placement and loading**

If you read a book on HTML, it is likely to advise that all of your `<script>` tags should be placed inside the `<head>` element on the page. If this is your current understanding on `<script>` tag placement, erase it from your memory right now! Placing `<script>` tags near the top of your HTML page will block the page from rendering until the JavaScript code has completed loading and executing. If you put them inside the `<head>` tag, no part of the page body will actually render until after the script has loaded and executed, giving the user the impression that your page is loading slowly.

To maximize the performance of your pages, you should place your JavaScript code near the bottom of your page, just before the `</body>` tag, where possible. This way, the rest of the web page (HTML, CSS, images, Flash content, and so on) will all download and render before your scripts are loaded and executed, giving the impression of a faster load to the user.

If your web page or application requires a lot of JavaScript, putting all of this code into a single file may lead to a long wait as it is downloaded and executed. In these cases, it may make sense to break your JavaScript code into multiple files and dynamically load these as necessary when the page has finished loading. The LazyLoad JavaScript library aims to provide dynamic script loading and will take care of some cross-browser inconsistencies concerning the preservation of script execution order. For more information on the LazyLoad library, see [Related topics](#).

**Minimize Ajax requests**

Ajax requests have revolutionized the way traditional web applications look and feel, allowing JavaScript developers to create applications that are highly dynamic, interactive, and responsive,
much like you'd expect from a desktop application. As a result, it is now common to see Ajax requests all over the place in a modern web application. Sometimes it is easy to forget that, although the user does not see a new page being loaded, Ajax requests do perform an entire HTTP request, which is equivalent to the request made for a regular page load. As such, great care should be taken to minimize the number of Ajax requests that are employed.

An example of this is with pagination of search results. I have often seen applications that use an Ajax request to bring back the search result records themselves in a JSON array and a second request that returns the total number of results in the database, which is then used for the pagination logic. Listing 1 and Listing 2 show really basic examples of these two requests (using the Prototype framework).

Listing 1. First request: retrieve table records

```javascript
var url = "get_data.php";
var options = {
  method: "post",
  parameters: {"page":1,"rows":5},
  onSuccess: firstCallbackFunction,
  onFailure: firstCallbackFunction
}
new Ajax.Request(url, options);
```

Listing 2 shows a second request to retrieve the total records.

Listing 2. Second request: retrieve total records count

```javascript
var url = "get_count.php";
var options = {
  method: "post",
  parameters: {},
  onSuccess: secondCallbackFunction,
  onFailure: secondCallbackFunction
}
new Ajax.Request(url, options);
```

Listing 3 and Listing 4 show the two corresponding HTTP responses in JSON format.

Listing 3. First response: array of records

```json
{"records": [
  {"id":1,"name":"John","email":"john@example.com"},
  {"id":2,"name":"Mary","email":"mary@example.com"},
  {"id":3,"name":"Tony","email":"tony@example.com"},
  {"id":4,"name":"Emma","email":"emma@example.com"},
  {"id":5,"name":"Alan","email":"alan@example.com"}
]}
```

Listing 4 shows a second response reporting the total number of records.

Listing 4. Second response: total count of records

```json
{"total_records": 95}
```
Separating these two Ajax requests is a waste of resources, when they could easily be combined into one request that produces the following response in Listing 5.

Listing 5. Efficient response: total count and array of records

```json
{
    "total_records": 95,
    "records": [
        {"id":1,"name":"John","email":"john@example.com"},
        {"id":2,"name":"Mary","email":"mary@example.com"},
        {"id":3,"name":"Tony","email":"tony@example.com"},
        {"id":4,"name":"Emma","email":"emma@example.com"},
        {"id":5,"name":"Alan","email":"alan@example.com"}
    ]
}
```

Not only does this require one less HTTP request and response to be made, but it also means you have one less server-side script to create, which provides a response to Ajax requests.

This example is a very simple demonstration—the more complex your application gets, the more important it is to consider if you can reduce the number of Ajax requests you need to use.

**Use variables (properly)**

In an effort to minimize the number of lines of code they write, many developers often overlook the use of variables, where in many cases they could considerably speed up the execution of a particular piece of code. Take, for instance, the following code, which applies various styles to an element:

Listing 6. Applying styles to an element (inefficient)

```javascript
document.getElementById("myField").style.backgroundColor = "#CCC";
document.getElementById("myField").style.color = "#FF0000";
document.getElementById("myField").style.fontWeight = "bold";
```

In each of the lines above, the browser will need to search through the DOM for an element with the ID myField. Instead of needing to do this three times, we could improve the efficiency of this operation by assigning the result of `document.getElementById("myField")` to a variable and using that in each line instead, as shown in Listing 7.

Listing 7. Applying styles to an element (efficient)

```javascript
var myField = document.getElementById("myField");
myField.style.backgroundColor = "#CCC";
myField.style.color = "#FF0000";
myField.style.fontWeight = "bold";
```

Another example of where variables are often not used when they should be is in the case of `for` loops that iterate through an array. Take the example shown in Listing 8.

Listing 8. Looping through an array with `for` (inefficient)

```javascript
for(var i=0; i < myArray.length; i++) {
    //do something
}
```
This code is highly inefficient as it will need to calculate the length of the array `myArray` on each iteration of the loop. While this may not be noticeable with small arrays, the consequences are much greater on a larger array. Improving the performance of this loop is very simple and doesn't even require an additional line of code (see Listing 9).

**Listing 9. Looping through an array with for (efficient)**

```javascript
for(var i=0, arrayLength=myArray.length; i < arrayLength; i++) {
    //do something
}
```

In Listing 9, we have moved the reference to `myArray.length` into the initialization section of the for statement, assigning it to a variable. This will only happen on the first iteration of the loop, saving valuable execution time on each subsequent iteration.

**Working with the DOM**

Traversing and manipulating the DOM in general can have a heavy performance burden on a web application. The problem is, interacting with the DOM is absolutely essential in order to deliver a highly responsive, rich interface to your application. A particular issue with DOM manipulation is that the browser needs to re-flow and re-paint (basically re-render) the affected elements on the screen. As a result, it is important to minimize the number of times your code results in a re-flow and re-paint.

Take the previous example in Listing 7, where we took an element with the ID `myField` and applied three separate style properties to it. This would have resulted in three separate re-flows and re-paints, which is inefficient, given that we were making fairly insignificant changes. It is especially inefficient given that all of these actions could have been combined into a single manipulation action. Listing 10 shows a more efficient example.

**Listing 10. Improving DOM manipulation performance by combining changes**

```javascript
var myField = document.getElementById("myField");
myField.style.cssText = "background-color: #CCC; color: #FF0000; font-weight: bold";
```

By combining the style changes, the result is a single re-flow and re-paint, and by extension, improved responsiveness and performance in this application.

This section covered many best practices for JavaScript and Ajax development that can help deliver optimal performance in your applications. This list is by no means exhaustive. See Related topics for links to excellent articles, tutorials, and books that are dedicated to highlighting the best practices for high-performance JavaScript.

**Measuring performance**

When it comes to measuring the performance of a web application, a series of tools is available that allows you to analyze the speed and load time of your application from within the web browser itself. These tools generally provide features to profile a piece of JavaScript code and basically calculate the time it takes to execute. They also provide a means of analyzing the network traffic...
generated by a page load in terms of the number of HTTP requests it causes, the size of the static resources (images, external stylesheets, and JavaScript files) loaded, and so on.

In this section, you will learn about the various tools at your disposal for profiling and analyzing network activity in your JavaScript applications.

**Firebug**

Firebug is a browser extension for Mozilla® Firefox® that provides a wealth of useful functionality to web developers. These features include a JavaScript console, DOM manipulation and selection, script debugging, DOM source browsing, profiling, network analysis, and much more.

To perform JavaScript profiling in Firebug, you navigate to the Console tab and click the Profile button. Now, you interact with the various JavaScript/Ajax-enabled features on your page and Firebug will time them for you. Firebug also provides a console API to interact with the profiler. Figure 1 shows a screen shot of a sample report generated by the Firebug profiler.

**Figure 1. Firebug profiler sample report**

![Firebug profiler sample report](image)

To use Firebug's network analysis tools, simply click on the Net tab. It will show you all of the HTTP requests made, the response code and message received, the domain it was retrieved from, the file size, and the timeline of the page load. You can even drill into requests to see the HTTP headers sent, the response received, and the relevant cache information for the file in question. Figure 2 shows a sample of the Firebug output for network traffic.
Safari Web Inspector and Chrome Developer Tools
Safari® Web Inspector and Chrome® Developer Tools provide similar features to Firebug, albeit they look much prettier. In Safari, open the Web Inspector by going to the Develop menu in the toolbar (you may need to enable it in your Safari preferences) and selecting **Show Web Inspector**. Click on the **Profile** tab at the top of the inspector window and click the circle record icon in the bottom left to start profiling. Click the icon again to stop profiling and generate a report. **Figure 3** shows a sample of what this report looks like.
To analyze the load time and the HTTP requests made by your application, click the **Resources** button at the top of the Inspector window. You can then choose to view a graph of the network resources used by either time or size. A sample of the beautiful output can be seen in **Figure 4**.

**Figure 4. Safari Web Inspector resources sample report**

Google Chrome's Developer Tools are the same as Safari's (both browsers being based on WebKit), but can be found in the Developer menu as Developer Tools.

**Internet Explorer Developer Tools**

Internet Explorer® 8 comes with a set of Developer Tools also. Hit F12 to launch the Developer Tools window. Internet Explorer's tools offer a subset of the features included with the likes of Firebug, but it does include a decent profiler for measuring your JavaScript functions' performance. In Developer Tools, click the **Profiler** tab and click the **Start Profiling** button to begin profiling your application. Use your application to perform the functions you wish to test and then click **Stop Profiling** to generate the report. It should look similar to the one shown in **Figure 5**.
Unfortunately, Internet Explorer's Developer Tools don't include a network analyzer. If you want to analyze the network traffic in your application using Internet Explorer, you can use the Fiddler tool that works with pretty much any application that makes HTTP requests, including Internet Explorer. For more on Fiddler, see Related topics.

**YUI Profiler**

The YUI Profiler is a code profiler tool written in JavaScript. Unlike the profiler tools included with web browsers (or browser extensions like Firebug), the YUI profiler is non-visual and must be included in your web page. The benefit of using it is that you can specifically target the areas of your application that you wish to profile (and not just any function that is executed). YUI Profiler allows you to register functions, class constructors, and objects to be measured. You can then produce a report of the output produced by the profiler, optionally providing a filter function which allows you to further pinpoint the target areas you are looking to measure. The profile report is produced in JSON format, which allows you to easily export results into your own applications, where you can produce all sorts of tabular and graph-based views of the data.

One of the primary advantages of YUI Profiler is that it is not specific to any particular browser. It will even run on mobile devices with an A grade browser, such as the WebKit-based browsers found on devices like the Apple® iPhone®, Android® devices, and the Nokia® N95.

For more information about YUI Profiler, see Related topics.

**YSlow**

YSlow is a Firefox extension that plugs in to the Firebug extension, providing scoring for the loading and execution of a web page. It provides an overall grade as well as detailed analysis on
various aspects of JavaScript and web page performance in general. In my test, I ran YSlow on a page that made 91 HTTP requests, which received a grade level of C. It provided a useful tip that the page had 10 external JavaScript scripts, recommending that I should try combining them into one.

Other areas it grades include adding expires headers, using gzip, putting JavaScript at the bottom of the page, minifying JavaScript and CSS, removing duplicate JavaScript and CSS code, and much more. Figure 6 shows an example of the output produced by YSlow.

Figure 6. YSlow sample report

Improving performance without changing code

Now that you have measured the performance of your application and determined that you need to give it a performance boost, you're probably thinking that you need to dig deep and modify your application's source code in order to do so. Well, yes, this is likely necessary in some respect, but there are several ways to improve JavaScript and Ajax performance without changing your application code at all.

Combine JavaScript source files

If you have a large JavaScript application, you are likely loading many external JavaScript source files into your pages to provide various pieces of functionality. Perhaps you are using a framework such as jQuery and numerous plug-ins that add additional features that build on top of jQuery itself. It is likely that you also have your own application's underlying JavaScript code, which may itself be split over several files.

While there are advantages of separating JavaScript into separate files, this has a serious impact on performance. For each external JavaScript file you load in your web application, a
separate HTTP request is performed, which drastically increases the load and execution time of your scripts. To reduce the number of requests, it is highly recommended that you merge your JavaScript source files where possible. Granted, if a particular plug-in is only required on one page, it shouldn't be included in the single source file. In fact, there's an argument that perhaps this should be included as an inline script, seeing as it's not going to be used on other pages. It may well be that you can't merge absolutely everything into one file, but the lower the number of HTTP requests your application makes, the faster your application will load.

**Minify JavaScript source files**

Good programming practice guidelines state that you should adhere to a certain style when writing application code, and this guideline holds true for JavaScript as much as any other language. The problem is, all of the whitespace, line breaks, and comments that you add to your code increase the file size of the script and the load time of your application.

In order to reduce your JavaScript file sizes, it is highly recommended that you minify your files before putting them into a production environment. Minification basically involves stripping out the extra bits of your source code to produce a lean and smaller file. For an example of the potential savings, minification alone takes the latest version of jQuery at the time of writing down from 155KB to 70.6KB. That's a savings of almost 55%.

Of course, it's all well and good to suggest that you minify your source code, but how do you go about doing so? No sane person would be expected to minify his or her code manually, so thankfully there are a series of developer tools out there that will do the job for you. Some popular ones include Douglas Crockford's JSMIN, Dean Edwards' Packer, and the compressor included in the Dojo toolkit. My personal favorite is the YUI Compressor from Yahoo!® Inc. For more information on these tools, see Related topics.

**Compressing your JavaScript source files with gzip**

In the previous section, you learned that you can drastically reduce the size of your JavaScript source files by minifying them. In fact, you can further reduce the file size by compressing the files using gzip. The best way to do this is to tell your web server to gzip your JavaScript files before sending them to the client. If you are using Apache 2 (with mod_deflate) and have your JavaScript files stored in the same directory, you can simply create a .htaccess file in the same location as your JavaScript files. The contents of this file should be simply: SetOutputFilter DEFLATE.

The savings from compressing your JavaScript files are significant. In the previous section, you learned that minifying the jQuery source code resulted in a drop in file size from 155KB to 70.6KB (55% reduction). If we gzip the minified file, the file size is further reduced to just 24KB. Between minification and compression, that's a total savings of almost 85%. Results will vary depending on your code, but the savings can be quite significant.

**Caching JavaScript source files**

It is important to ensure that the web browser caches your content correctly. For instance, if you use Apache (with mod_expires) and you want the client to cache your JavaScript files for two days,
you could add the directive in Listing 11 to the .htaccess file in the location where you keep your JavaScript files (assuming that you have a directory dedicated to JavaScript files; if not, you may have to do this in your Apache configuration file).

**Listing 11. Directive to cache your JavaScript files for two days**

ExpiresActive on  
ExpiresDefault "access plus 2 days"

Of course, the problem with caching JavaScript files is that if you make changes, any clients with the cached version who revisit inside the cache expiry time will use the cached version of your script and not the updated version. Fortunately, you can force the client to retrieve the latest version of the script by appending a query string with a version number to the <script> tag that loads the script. This query string will have no impact on the JavaScript code, but as far as the browser is concerned, it is a completely separate file and it will download the fresh version. Of course, it is important to increase the version number each time you make a change to a file. In larger applications, a build script should be used to automate this process to prevent issues from occurring.

**Conclusion**

In this article you learned about the importance of JavaScript and Ajax performance in modern websites and applications. First, you saw how some good JavaScript programming practices can make a big difference to your application's responsiveness and load times. Next, you learned how to measure the performance of an existing application using profiling and network analyzer tools. Finally, you learned how to speed up existing applications without modifying your application source code by employing some simple but powerful techniques to reduce file sizes and the number of HTTP requests made by your application.
Related topics

• **Mastering Ajax, Part 2: Make asynchronous requests with JavaScript and Ajax** (Brett McLaughlin, developerWorks, January 2006): Learn how to use Ajax and the XMLHttpRequest object to create a request/response model that never leaves users waiting for a server to respond.

• **Create Ajax applications for the mobile Web** (Michael Galpin, developerWorks, March 2010): Learn how to build cross-browser smartphone web applications using Ajax.

• **Where and when to use Ajax in your applications** (Jesse Skinner, developerWorks, February 2008): Learn how you can use Ajax to improve your websites while avoiding bad user experiences.

• **Improve the performance of Web 2.0 applications** (Jian Qiao Sun and Hua Pin Shen, developerWorks, December 2009): Explore different browser-side cache mechanisms.

• **Ajax performance analysis** (Kristopher William Zyp, developerWorks, April 2008): Examine toolsets that find and correct performance problems within your Ajax-enriched application.

• **Compare JavaScript frameworks** (Joe Lennon, developerWorks, February 2010): Get an overview of the frameworks that greatly enhance JavaScript development.

• **High Performance Web Sites** by Steve Souders (O'Reilly Media, September 2007): See 14 specific tools you can use to speed up your website.

• **High Performance JavaScript** by Nicholas Zakas (O'Reilly Media, March 2010): Learn techniques and strategies to help you eliminate performance bottlenecks during development.

• **Best Practices for Speeding Up Your Web Site**: This article from the Yahoo! Developer Network includes 35 best practices for making web pages fast.

• Visit the Mozilla Developer Network, a network of diverse developer communities that cover the Web, Mobile, Add-ons, Applications, and Labs.

• Learn JSON syntax in this **Introducing JSON** article.

• The **LazyLoad library** is a tiny (only 1,541 bytes minified), dependency-free JavaScript library that makes it easy to load external JavaScript and CSS files on demand.

• Get the **Firebug** web development tool.

• The **YUI Profiler** is a simple, non-visual code profiler for JavaScript.

• **YSlow** analyzes web pages and suggests ways to improve their performance based on a set of rules for high-performance web pages.

• **Fiddler** is a Web Debugging Proxy which logs all HTTP(S) traffic between your computer and the Internet.

• Minify your JavaScript code with **YUI Compressor**.

• To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.

© Copyright IBM Corporation 2010

Trademark
(www.ibm.com/developerworks/ibm/trademarks/)