DOI:10.1145/1409360.1409374

Want to make your Web site fly? Focus on frontend performance.

BY STEVE SOUDERS

High-Performance Web Sites

Google Maps, Yahoo! Mail, Facebook, MySpace, YouTube, and Amazon are examples of Web sites built to scale. They access petabytes of data sending terabits per second to millions of users worldwide. The magnitude is awe-inspiring.

Users view these large-scale Web sites from a narrower perspective. The typical user has megabytes of data that they download at a few hundred kilobits per second. Users are less interested in the massive number of requests per second being served, caring more about their individual requests. As they use these Web applications they inevitably ask the same question: "Why is this site so slow?"

The answer hinges on where development teams focus their performance improvements. Performance for the sake of scalability is rightly focused on the backend. Database tuning, replicating architectures, customized data caching, and so on, allow Web servers to handle a greater number of requests. This gain in efficiency translates into reductions in hardware costs, data center rack space, and power consumption. But how much does the backend affect the user experience in terms of latency?

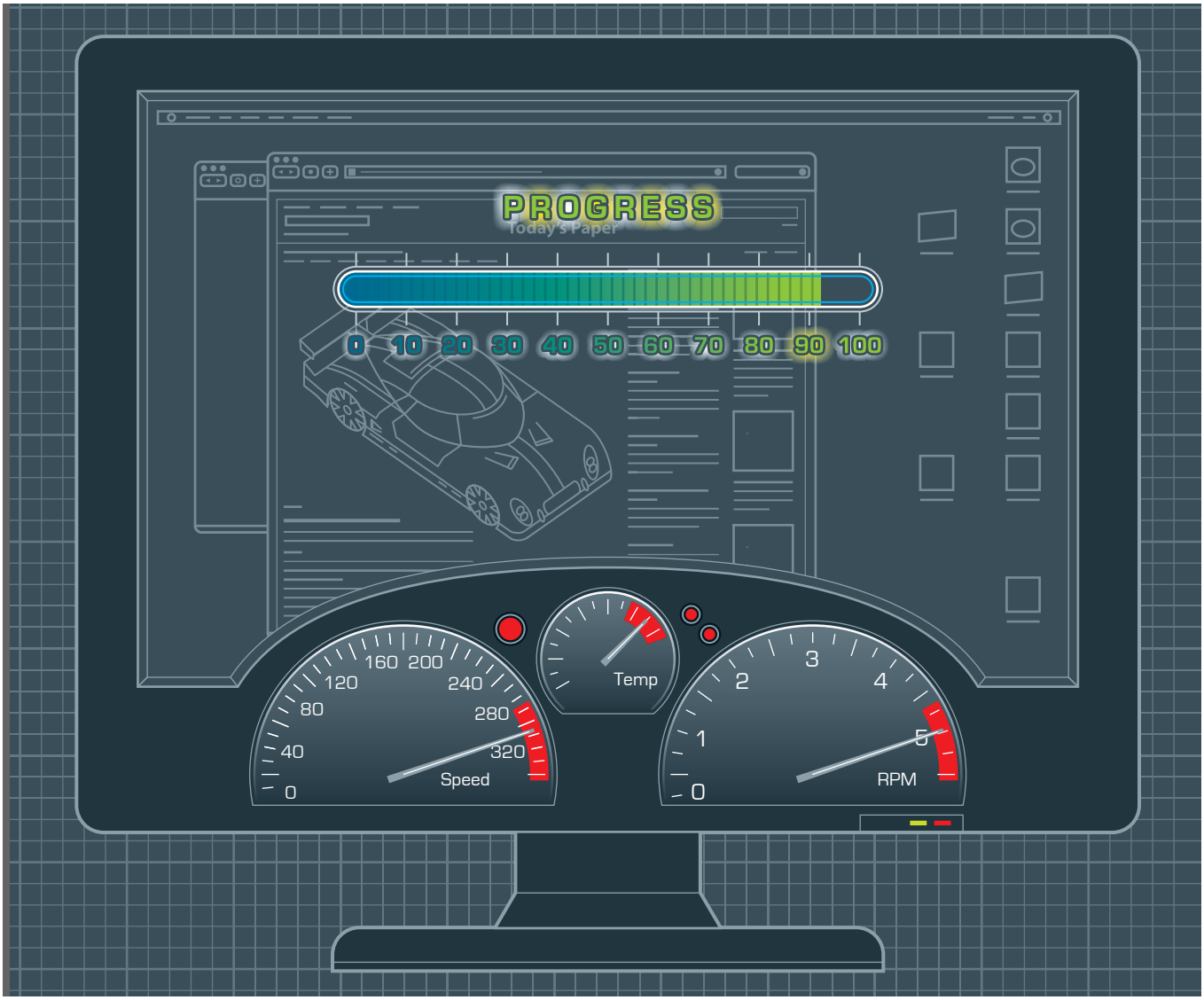
The Web applications listed here are some of the most highly tuned in the world, and yet they still take longer to load than we'd like. It almost seems as if the high-speed storage and optimized application code on the backend have little impact on the end user's response time. Therefore, to account for these slowly loading pages we must focus on something other than the backend: we must focus on the *frontend*.

The Importance of Frontend Performance

Figure 1 illustrates the HTTP traffic sent when your browser visits iGoogle with an empty cache. Each HTTP request is represented by a horizontal bar whose position and size are based on when the request began and how long it took. The first HTTP request is for the HTML document (http://www.google.com/ig). As noted in Figure 1, the request for the HTML document took only 9% of the overall page load time. This includes the time for the request to be sent from the browser to the server, for the server to gather all the necessary information on the backend and stitch that together as HTML, and for that HTML to be sent back to the browser.

The other 91% percent is spent on the *frontend*, which includes everything that the HTML document commands the browser to do. A large part of this is fetching resources. For this iGoogle page there are 22 additional HTTP requests: two scripts, one stylesheet, one iframe, and 18 images. Gaps in the HTTP profile (places with no network traffic) are where the browser is parsing CSS, and parsing and executing JavaScript.

The primed cache situation for iGoogle is shown in Figure 2. Here there are only two HTTP requests: one for the HTML document and one for a dynamic script. The gap is even larger because it includes the time to read the cached resources from disk. Even in the primed cache situation, the HTML document accounts for only 17% of the



overall page load time.

This situation, in which a large percentage of page load time is spent on the frontend, applies to most Web sites. Table 1 shows that eight of the top 10 Web sites in the U.S. (as listed on Alexa.com) spend less than 20% of the end user's response time fetching the HTML document from the backend. The two exceptions are Google Search and Live Search, which are highly tuned. These two sites download four or fewer resources in the empty cache situation, and only one request with a primed cache.

The time spent generating the HTML document affects overall latency, but for most Web sites this backend time is dwarfed by the amount of time spent on the frontend. If the goal is to make the user experience faster, the place to focus is on the frontend. Given this new focus, the next step is to identify best practices for improving frontend performance.

Frontend Performance Best Practices

Through research and consulting with development teams, I've developed a set of performance improvements that have been proven to speed up Web pages. A big fan of *Harvey Penick's Little Red Book*[1] with advice like "Take Dead Aim," I set out to capture these best practices in a simple list that is easy to remember. The list has evolved to contain the following 14 prioritized rules:

1. Make fewer HTTP requests
2. Use a content delivery network
3. Add an Expires header
4. Gzip components
5. Put stylesheets at the top
6. Put scripts at the bottom
7. Avoid CSS expressions
8. Make JavaScript and CSS external
9. Reduce DNS lookups
10. Minify JavaScript
11. Avoid redirects
12. Remove duplicate scripts
13. Configure ETags
14. Make Ajax cacheable

A detailed explanation of each rule is the basis of my book, *High Performance Web Sites*.[2] What follows is a brief summary of each rule.

Rule 1: Make Fewer HTTP Requests

As the number of resources in the page grows, so does the overall page load time. This is exacerbated by the fact that most browsers only download two resources at a time from a given hostname, as suggested in the HTTP/1.1 specification (http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html#sec8.1.4).[a] Several techniques exist for reducing the num-

a Newer browsers open more than two connections per hostname including Internet Explorer 8 (six), Firefox 3 (six), Safari 3 (four), and Opera 9 (four).

ILLUSTRATION BY NIK SCHULZ

Figure 1. iGoogle HTTP traffic with an empty cache.

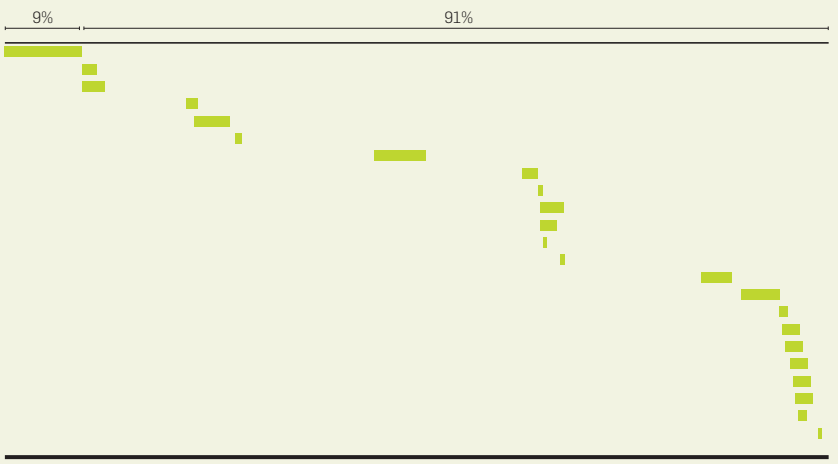


Figure 2. iGoogle HTTP traffic with a primed cache.

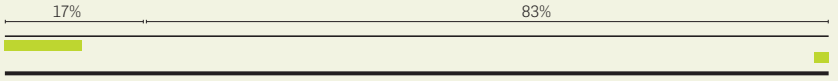


Table 1. Percentage of time spent on the backend.

Web Site	Empty Cache	Primed Cache
http://www.aol.com/	3%	3%
http://www.ebay.com/	5%	19%
http://www.facebook.com/	5%	19%
http://www.google.com/search?q=flowers	53%	100%
http://search.live.com/results.aspx?q=flowers	33%	100%
http://www.msn.com/	2%	6%
http://www.myspace.com/	2%	2%
http://en.wikipedia.org/wiki/Flowers	6%	9%
http://www.yahoo.com/	3%	4%
http://www.youtube.com/	2%	3%

ber of HTTP requests without reducing page content:

- Combine multiple scripts into a single script.
- Combine multiple stylesheets into a single stylesheet.
- Combine multiple CSS background images into a single image called a CSS sprite (see http://alistapart.com/articles/sprites).

Rule 2: Use a Content Delivery Network

A content delivery network (CDN) is a collection of distributed Web servers used to deliver content to users more efficiently. Examples include Akamai Technologies, Limelight Networks, SAVVIS, and Panther Express. The main performance advantage provided by a CDN is delivering static resources from a server that is geographically closer to the end user. Other benefits include backups, caching, and the ability to better absorb traffic spikes.

Rule 3: Add an Expires Header

When a user visits a Web page, the browser downloads and caches the page's resources. The next time the user visits the page, the browser checks to see if any of the resources can be served from its cache, avoiding time-consuming HTTP requests. The browser bases its decision on the resource's expiration date. If there is an expiration date, and that date is in the future, then the resource is read from disk. If there is no expiration date, or that date is in the past, the browser issues a costly HTTP request. Web developers can attain this performance gain by specifying an explicit expiration date in the future. This is done with the Expires HTTP response header, such as the following:

Expires: Thu, 1 Jan 2015 20:00:00 GMT

Rule 4: Gzip Components

The amount of data transferred over the network affects response times, especially for users with slow network connections. For decades developers have used compression to reduce the size of files. This same technique can be used for reducing the size of data sent over the Internet. Many Web servers and Web hosting services enable compression of HTML documents by default, but compression shouldn't stop there. Developers should also compress other types of text responses, such as scripts, stylesheets, XML, JSON, among others. Gzip is the most popular compression technique. It typically reduces data sizes by 70%.

Rule 5: Put Stylesheets at the Top

Stylesheets inform the browser how to format elements in the page. If stylesheets are included lower in the page, the question arises: What should the browser do with elements that it can render before the stylesheet has been downloaded?

One answer, used by Internet Explorer, is to delay rendering elements in the page until all stylesheets are downloaded. But this causes the page to appear blank for a longer period of time, giving users the impression that the page is slow. Another answer, used by Firefox, is to render page elements and redraw them later if the stylesheet changes the initial formatting. This causes elements in the page to "flash" when they're redrawn, which is dis-

ruptive to the user. The best answer is to avoid including stylesheets lower in the page, and instead load them in the HEAD of the document.

Rule 6: Put Scripts at the Bottom

External scripts (typically, ".js" files) have a bigger impact on performance than other resources for two reasons. First, once a browser starts downloading a script it won't start any other parallel downloads. Second, the browser won't render any elements below a script until the script has finished downloading. Both of these impacts are felt when scripts are placed near the top of the page, such as in the HEAD section. Other resources in the page (such as images) are delayed from being downloaded and elements in the page that already exist (such as the HTML text in the document itself) aren't displayed until the earlier scripts are done. Moving scripts lower in the page avoids these problems.

Rule 7: Avoid CSS Expressions

CSS expressions are a way to set CSS properties dynamically in Internet Explorer. They enable setting a style's property based on the result of executing JavaScript code embedded within the style declaration. The issue with CSS expressions is that they are evaluated more frequently than one might expect—potentially thousands of times during a single page load. If the JavaScript code is inefficient it can cause the page to load more slowly.

Rule 8: Make JavaScript and CSS External

JavaScript can be added to a page as an inline script:

```
<script type="text/javascript">
var foo="bar";
</script>
```

or as an external script:

```
<script src="foo.js" type="text/javascript"></script>
```

Similarly, CSS is included as either an inline style block or an external stylesheet. Which is better from a performance perspective?

HTML documents typically are not cached because their content is constantly changing. JavaScript and CSS are less dynamic, often not changing for weeks or months. Inlining JavaScript and CSS results in the same bytes (that haven't changed) being downloaded on every page view. This has a negative impact on response times and increases the bandwidth used from your data center. For most Web sites, it's better to serve JavaScript and CSS via external files, while making them cacheable with a far future Expires header as explained in Rule 3.

Rule 9: Reduce DNS Lookups

The Domain Name System (DNS) is like a phone book: it maps a hostname to an IP address. Hostnames are easier for humans to understand, but the IP address is what browsers need to establish a connection to the Web server. Every hostname that's used in a Web page must be resolved using DNS. These DNS lookups carry a cost; they can take 20–100 milliseconds each. Therefore, it's best to reduce the number of unique hostnames used in a Web page.

Rule 10: Minify JavaScript

As described in Rule 4, compression is the best way to reduce the size of text files transferred over the Internet. The size of JavaScript can be further reduced by minifying the code. Minification is the process of stripping unneeded characters (comments, tabs, new lines, extra white space, and so on) from the code. Minification typically reduces the size of JavaScript by 20%. External scripts should be minified, but inline scripts also benefit from this size reduction.

Rule 11: Avoid Redirects

Redirects are used to map users from one URL to another. They're easy to implement and useful when the true URL is too long or complicated for users to remember, or if a URL has changed. The downside is that redirects insert an extra HTTP roundtrip between the user and her content. In many cases, redirects can be avoided with some additional work. If a redirect is truly necessary, make sure to issue it with a far future Expires header (see Rule 3), so that on future visits the user can avoid this delay. [b]

Rule 12: Remove Duplicate Scripts

If an external script is included multiple times in a page, the browser has to parse and execute the same code multiple times. In some cases the browser will request the file multiple times. This is inefficient and causes the page to load more slowly. This obvious mistake would seem uncommon, but in a review of U.S. Web sites it could be found in two of the top 10 sites. Web sites that have a large number of scripts and a large number of developers are most likely to suffer from this problem.

Rule 13: Configure ETags

Entity tags (ETags) are a mechanism used by Web clients and servers to verify that a cached resource is valid. In other words, does the resource (image, script, stylesheet, among others) in the browser's cache match the one on the server? If so, rather than transmitting the entire file (again), the server simply returns a 304 Not Modified status telling the browser to use its locally cached copy. In HTTP/1.0, validity checks were based on a resource's Last-Modified date: if the date of the cached file matched the file on the server, then the validation succeeded. ETags were introduced in HTTP/1.1 to allow for validation schemes based on other information, such as version number and checksum.

ETags don't come without a cost. They add extra headers to HTTP requests and responses. The default ETag syntax used in Apache and IIS makes it likely that the validation will erroneously fail if the Web site is hosted on multiple servers. These costs impact performance, making pages slower and increasing the load on Web servers. This is an unnecessary loss of performance, because most Web sites don't take advantage of the advanced features of ETags, relying instead on the Last-Modified date as the means of validation. By default, ETags are enabled in popular Web servers (including Apache and IIS). If your Web site doesn't utilize ETags, it's best to turn them off in your Web server. In Apache, this is done by simply adding "FileETag none" to your configuration file.

Rule 14: Make Ajax Cacheable

Many popular Web sites are moving to Web 2.0 and have begun incorporating Ajax. Ajax requests involve fetching data that is often dynamic, personalized, or both. In the Web 1.0 world, this data is served by the user going to a specified URL and getting back an HTML docu-

b Caching redirects is not supported in some browsers.

Table 2. Percentage of unused JavaScript functions.

Web Site	JavaScript Size	Unused Functions
http://www.aol.com/	115K	70%
http://www.ebay.com/	183K	56%
http://www.facebook.com/	1088K	81%
http://www.google.com/search?q=flowers	15K	55%
http://search.live.com/results.aspx?q=flowers	17K	76%
http://www.msn.com/	131K	69%
http://www.myspace.com/	297K	82%
http://en.wikipedia.org/wiki/Flowers	114K	68%
http://www.yahoo.com/	321K	87%
http://www.youtube.com/	240%	82%
average	**252K**	**74%**

ment. Because the HTML document's URL is fixed (bookmarked, linked to, and so on), it's necessary to ensure the response is not cached by the browser.

This is not the case for Ajax responses. The URL of the Ajax request is included inside the HTML document; it's not bookmarked or linked to. Developers have the freedom to change the Ajax request's URL when they generate the page. This allows developers to make Ajax responses cacheable. If an updated version of the Ajax data is available, the cached version is avoided by adding a dynamic variable to the Ajax URL. For example, an Ajax request for the user's address book could include the time it was last edited as a parameter in the URL, "&edit=1218050433." As long as the user hasn't edited their address book, the previously cached Ajax response can continue to be used, making for a faster page.

Performance Analysis with YSlow

Evangelizing these performance best practices is a challenge. I was able to share this information through conferences, training classes, consulting, and documentation. Even with the knowledge in hand, it would still take hours of loading pages in a packet sniffer and reading HTML to identify the appropriate set of performance improvements. A better alternative would be to codify this expertise in a tool that anyone could run, reducing the learning curve and increasing adoption of these performance best practices. This was the inspiration for YSlow.

YSlow (http://developer.yahoo.com/yslow/) is a performance analysis tool that answers the question posed in the introduction: "Why is this site so slow?" I created YSlow so that any Web developer could quickly and easily apply the performance rules to their site, and find out specifically what needed to be improved. It runs inside Firefox as an extension to Firebug (http://getfirebug.com/), the tool of choice for many Web developers.

The screenshot in Figure 3 shows Firefox with iGoogle loaded. Firebug is open in the lower portion of the window, with tabs for Console, HTML, CSS, Script, DOM, and Net. When YSlow is installed, the YSlow tab is added. Clicking YSlow's Performance button initiates an analysis of the page against the set of rules, resulting in a weighted score for the page.

As shown in Figure 3, YSlow explains each rule's results with details about what to fix. Each rule in the YSlow screen is a link to the companion Web site, where additional information about the rule is available.

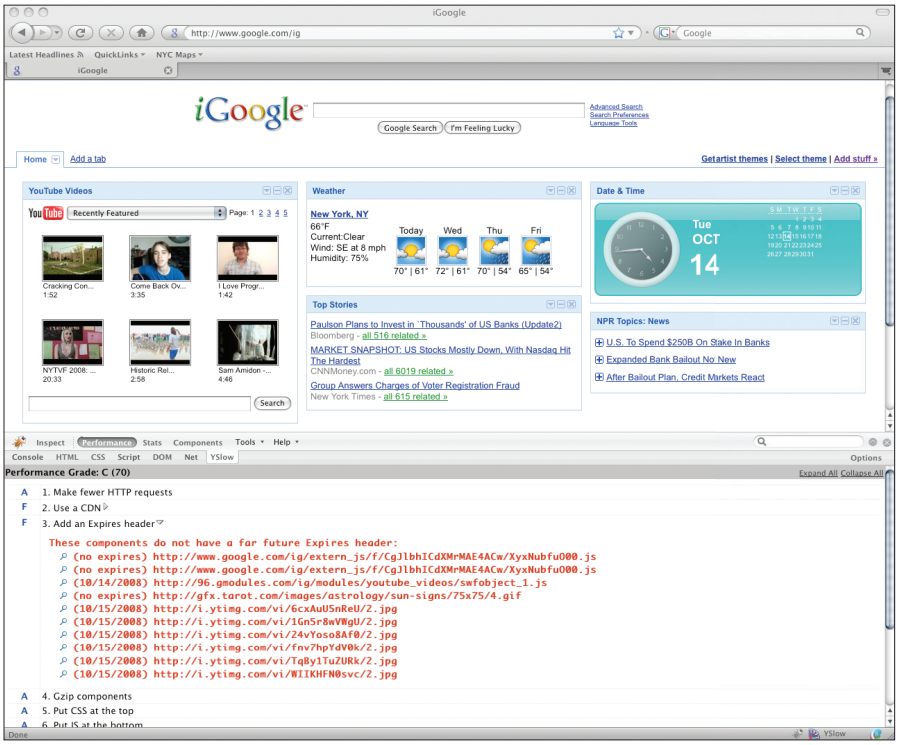


Figure 3: YSlow.

The Next Performance Challenge: JavaScript

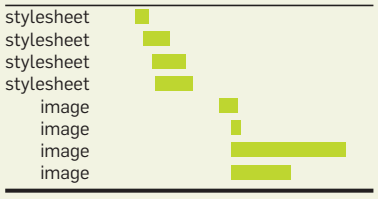
Web 2.0 promises a future where developers can build Web applications that provide an experience similar to desktop apps. Web 2.0 apps are built using JavaScript, which presents significant performance challenges because JavaScript blocks downloads and rendering in the browser. To build faster Web 2.0 apps, developers should address these performance issues using the following guidelines:

- Split the initial payload
- Load scripts without blocking
- Don't scatter scripts

Split the Initial Payload

Web 2.0 apps involve just a single

Figure 4. Stylesheet followed by inline script in www.msn.com/.



page load. Instead of loading more pages for each action or piece of information requested by the user, as was done in Web 1.0, Web 2.0 apps use Ajax to make HTTP requests behind the scenes and update the user interface appropriately. This means that some of the JavaScript that is downloaded is not used immediately, but instead is there to provide functionality that the user might need in the future. The problem is that this subset of JavaScript blocks other content that *is* used immediately, delaying immediate content for the sake of future functionality that may never be used.

Table 2 shows that for the top 10 U.S. Web sites, an average of 74% of the functionality downloaded is not used immediately. To take advantage of this opportunity, Web developers should split their JavaScript payload into two scripts: the code that's used immediately (~26%) and the code for additional functionality (~74%). The first script should be downloaded just as it is today, but given its reduced size the initial page will load more quickly. The second script should be *lazy-loaded*, which means that after the initial page is completely rendered this second script is downloaded dynamically, using one of the techniques listed in the next section.

Load Scripts without Blocking

As described in "Rule 6: Put Scripts at the Bottom," external scripts block the download and rendering of other content in the page. This is true when the script is loaded in the typical way:

```
<script src="foo.js" type="text/javascript"></script>
```

But there are several techniques for downloading scripts that avoid this blocking behavior:

- XHR eval
- XHR injection
- script in iframe
- script DOM element
- script defer
- document.write script tag

You can see these techniques illustrated in Cuzillion (http://stevesouders.com/cuzillion/), but as an example let's look at the *script DOM element approach*:

```
<script type="text/javascript">
  var se = document.createElement
('script');
  se.src = 'http://anydomain.
com/foo.js';
  document.getElementsByTagName
('head')[0].appendChild(se);
  </script>
```

A new DOM element is created that is a script. The src attribute is set to the URL of the script. Appending it to the head of the document causes the script to be downloaded, parsed, and executed. When scripts are loaded this way, they don't block the downloading and rendering of other content in the page.

Don't Scatter Inline Scripts

These first two best practices about JavaScript performance have to do with external scripts. Inline scripts also impact performance, occasionally in significant and unexpected ways. The most important guideline with regard to inline scripts is to avoid a stylesheet followed by an inline script.

Figure 4 shows some of the HTTP traffic for http://www.msn.com/. We see that four stylesheet requests are downloaded in parallel, then there is a white gap, after which four images are downloaded, also in parallel with each other. But why aren't all eight downloaded in parallel?[c]

This page contains an inline script after the fourth stylesheet. Moving this inline script to either above the stylesheets or after the images would result in all eight requests taking place in parallel, cutting the overall download time in half. Instead, the images are blocked from downloading until the inline script is executed, and the inline script is blocked from executing until the stylesheets finish downloading. This seems like it would be a rare problem, but it afflicts four of the top ten sites in the U.S: eBay, MSN, MySpace, and Wikipedia.

Life's Too Short, Write Fast Code

At this point, I hope you're hooked on building high-performance Web sites. I've explained why fast sites are important, where to focus your performance efforts, specific best practices to follow for making your site faster, and a tool you can use to find out what to fix. But what happens tomorrow, when you're back at work facing a long task list and being pushed to add more features instead of improving performance? It's important to take a step back and see how performance fits into the bigger picture.

Speed is a factor that can be used for competitive advantage. Better features and a more appealing user interface are also distinguishing factors. It doesn't have to be one or the other. The point of sharing these performance best practices is so we can all build Web sites to be as fast as they possibly can—whether they're barebones or feature rich.

I tell developers "Life's too short, write fast code!" This can be interpreted two ways. Writing code that executes quickly saves time for our users. For large-scale Web sites, the savings add up to lifetimes of user activity. The other interpretation appeals to the sense of pride we have in our work. Fast code is a badge of honor for developers.

Performance must be a consideration intrinsic to Web development. The performance best practices described here are proven to work. If you want to make your Web site faster, focus on the frontend, run YSlow, and apply these rules. Who knows, fast might become your site's most popular feature. C

c Note that these requests are made on different hostnames and thus are not constrained by the two-connections-per-server restriction of some browsers, as described in "Rule 1: Make Fewer HTTP Requests."

References

1. Penick, H. *Harvey Penick's Little Red Book: Lessons and Teachings From A Lifetime In Golf.* Simon and Schuster, 1992.
2. Souders, S. *High Performance Web Sites: Essential Knowledge for Front-End Engineers.* O'Reilly, 2006.

Steve Souders (http://stevesouders.com) works at Google on Web performance and open source initiatives. He is the author of *High Performance Web Sites* and the creator of YSlow, Cuzillion, and Hammerhead. He teaches at Stanford and is the co-founder of the Firebug Working Group.