HTTP Requests

By Dave Gash
Dave is a Tech Writer

Everything a web page needs to be a web page – text, graphics, styles, scripts, everything – must be downloaded from a server via an HTTP request. It's no stretch to say that the vast majority of a page's total display time is spent in downloading its components, not in actually displaying them. So far, we've talked about reducing the size of those downloads by compressing images, minifying CSS and JavaScript, zipping the files, and so on. But there's a more fundamental approach: in addition to just reducing download *size*, let's also consider reducing download *frequency*.

Reducing the number of components that a page requires proportionally reduces the number of HTTP requests it has to make. This doesn't mean omitting content, it just means structuring it more efficiently.

Combine Text Resources

Many web pages use multiple stylesheets and script files. As we develop pages and add useful formatting and behavior code to them, we often put pieces of the code into separate files for clarity and ease of maintenance. Or we might keep our own stylesheet separate from the one the Marketing Department requires us to use. Or we might put experimental rules or scripts in separate files during testing. Those are all valid reasons to have multiple resource files.

But because each file requires its own HTTP request, and each request takes time, we can speed up page load by combining files; one request instead of three or four will certainly save time. At first blush, this seems like a no-brainer – just put all the CSS (for example) into a master stylesheet and then remove all but one from the page. Every extra file you eliminate in this way removes one HTTP request and saves round-trip time. But there are caveats.

For Cascading Style Sheets, beware the "C". Cascade precedence allows later rules to override earlier ones without warning – literally. CSS doesn't throw an error when a previously-defined property is reset by a more recent rule, so just tossing stylesheets together is asking for trouble. Instead, look for conflicting rules and determine whether one should always supersede the other, or if one should use more specific selectors to be applied

properly. For example, consider these two simple rules, the first from a master stylesheet, the second imported from a stylesheet provided by Marketing.

```
h2 { font-size: 1em; color: #000080; } /* master stylesheet */

. . .

h2 { font-size: 2em; color: #ff0000; } /* Marketing stylesheet */
```

Marketing wants their `h2`s to be prominent, while yours are meant to be more subdued. But due to the cascading order, their rule takes precedence and every `h2` in the page will be big and red. And clearly, you can't just flip the order of the stylesheets or you'll have the same problem in reverse.

A little research might show that Marketing's `h2`s always appear inside a specific class of `section`, so a tweak to the second rule's selector resolves the conflict. Marketing's splashy `h2`s will still look exactly as they want, but without affecting `h2`s elsewhere in the page.

```
h2 { font-size: 1em; color: #000080; }

. . .

section.product h2 { font-size: 2em; color: #ff0000; }
```

You may run into similar situations when combining JavaScript files. Completely different functions might have the same names, or identically-named variables might have different scopes and uses. These are not insurmountable obstacles if you actively look for them.

Combining text resources to reduce HTTP requests is worth doing, but take care when doing so. See **A Caveat** below.

Combine Graphical Resources

On its face, this technique sounds a bit nonsensical. Sure, it's logical to combine multiple CSS or JavaScript resources into one file, but images? Actually, it is fairly simple, and it has the same effect of reducing the number of HTTP requests as combining text resources – sometimes even more dramatically.

Although this technique can be applied to any group of images, it is most frequently used with small images such as icons, where extra HTTP requests to fetch multiple small graphics are particularly wasteful.

The basic idea is to combine small images into one physical image file and then use CSS background positioning to display only the right part of the image – commonly called a sprite – at the right place on the page. The CSS repositioning is fast and seamless, works on an already-downloaded resource, and makes an excellent tradeoff for the otherwise-required multiple HTTP requests and image downloads.

For example, you might have a series of social media icons with links to their respective sites or apps. Rather than downloading three (or perhaps many more) individual images, you might combine them into one image file, like this.

Then, instead of using different images for the links, just retrieve the entire image once and use CSS background positioning ("spriting") for each link to display the correct part of the image for the link.

Here's some sample CSS.

```
a.facebook {
   display: inline-block;
   width: 64px; height: 64px;
   background-image: url("socialmediaicons.png");
   background-position: 0px 0px;
   }
a.twitter {
   display: inline-block;
   width: 64px; height: 64px;
   background-image: url("socialmediaicons.png");
   background-position: -64px 0px;
   }
a.pinterest {
   display: inline-block;
   width: 64px; height: 64px;
   background-image: url("socialmediaicons.png");
   background-position: -128px 0px;
   }
```

Note the extraneous **background-position** property in the **facebook** class. It's not necessary, as the default position is 0,0 but is included here for consistency. The other two classes simply shift the image left horizontally relative to its container by 64px and 128px, respectively, leaving the appropriate image section visible in the 64-by-64 pixel link "window".

Here's some HTML to use with that CSS.

```
<p>Find us on:</p>
<p><a class="facebook" href="https://facebook.com"></a></p>
<p><a class="twitter" href="https://twitter.com"></a></p>
<p><a class="pinterest" href="https://pinterest.com"></a></p>
```

Instead of including separate images in the links themselves, just apply the CSS classes and leave the link content empty. This simple technique saves you two HTTP requests by letting CSS do the image shifting behind the scenes. If you have lots of small images – navigation icons or function buttons, for example – it could save you many, many trips to the server.

You can find a brief but excellent article about this technique, including working examples, at WellStyled.

A Caveat

In our discussion of combining text and graphics, we should note that the newer HTTP/2 protocol may change how you consider combining resources. For example, common and valuable techniques like minification, server compression, and image optimization should be continued on HTTP/2. However, physically combining files as discussed above might not achieve the desired result on HTTP/2.

This is primarily because server requests are faster on HTTP/2, so combining files to eliminate a request may not be substantially productive. Also, if you combine a fairly static resource with a fairly dynamic one to save a request, you may adversely affect your caching effectiveness by forcing the static portion of the resource to be reloaded just to fetch the dynamic portion.

The features and benefits of HTTP/2 are worth exploring in this context.

JavaScript Position and Inline Push

We're assuming so far that all CSS and JavaScript resources exist in external files, and that's generally the best way to deliver them. Bear in mind that script loading is a large and complex issue – see this great HTML5Rocks article, Deep Dive Into the Murky Waters of Script Loading, for a full treatment. There are, however, two fairly straightforward positional factors about JavaScript that are worth considering.

Script Location

Common convention is to put script blocks in the page head. The problem with this positioning is that, typically, little to none of the script is really meant to execute until the page is displayed but, while it is loading, it unnecessarily blocks page rendering. Identifying render-blocking script is one of the reporting rules of PageSpeed Insights.

A simple and effective solution is to reposition the deferred script block at the end of the page. That is, put the script reference last, just before the closing body tag. This allows the browser to load and render the page content, and then lets it download the script while the user perceives the initial content. For example:

```
<html>
  <head>
  </head>
  <body>
    [Body content goes here.]
  <script src="mainscript.js"></script>
  </body>
</html>
```

An exception to this technique is any script that manipulates the initial content or DOM, or provides required page functionality prior to or during rendering. Critical scripts such as these can be put into a separate file and loaded in the page head as usual, and the rest can still be placed last thing in the page for loading only after the page is rendered.

The order in which resources must be loaded for maximum efficiency is called the *Critical Rendering Path*; you can find a thorough article about it at Bits of Code.

Code Location

Of course, the technique described above splits your JavaScript into two files on the server and thus requires two HTTP requests instead of one, exactly the situation we're trying to avoid. A better solution for relocating critical, pre-render scripts might be to place them directly inside the page itself, referred to as an "inline push".

Here, instead of putting the critical script in a separate file and referencing it in the page head, add a **`<script>...</script>`** block, either in the head or in the body, and insert the script itself (not a file reference, but the actual script code) at the point at which it's needed. Assuming the script isn't too big, this method gets the script loaded along with the HTML and executed immediately, and avoids the extra HTTP request overhead of putting it in the page head.

For example, if a returning user's name is already available, you might want to display it in the page as soon as possible by calling a JavaScript function, rather than wait until after all the content is loaded.

```
<p>Welcome back, <script>insertText(username)</script>!</p>
```

Or you might need an entire function to execute in place as the page loads, in order to render certain content correctly.

```
<h1>Our Site</h1>

<h2 id="greethead">, and welcome to Our Site!</h2>

<script>
//insert time of day greeting from computer time
var hr = new Date().getHours();
var greeting = "Good morning";
if (hr > 11) {
    greeting = "Good afternoon";
}
if (hr > 17) {
    greeting = "Good evening";
}
h2 = document.getElementById("greethead");
h2.innerHTML = greeting + h2.innerHTML;
</script>

<p>Blah blah blah</p>
```

This simple technique avoids a separate HTTP request to retrieve a small amount of code and allows the script to run immediately at its appropriate place in the page, at the minor cost of a few dozen extra bytes in the HTML page.

Summary

In this section, we covered ways to reduce the number of HTTP requests our pages make, and considered techniques for both text and graphical resources. Every round-trip to the server we can avoid saves time, speeds up the page load, and gets its content to our users sooner.

registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.