

11

Looking to the future with HTTP/2

This chapter covers

- Learning the history of HTTP/1 and its problems
- Exploring the evolution of HTTP/2
- Understanding request multiplexing and header compression, new in HTTP/2
- Exploring how optimization practices differ between HTTP/1 and HTTP/2
- Speeding the delivery of crucial page assets by using Server Push
- Optimizing for HTTP/1 and HTTP/2 clients on the same server

The web is changing. For years, users and developers have been vexed by the limitations of the HTTP/1 protocol. Although developers have been squeezing every last drop of performance from this aging protocol, we must accept that it's time to move on and adopt HTTP/2.

This chapter covers the problems inherent in HTTP/1 as well as the benefits of HTTP/2, such as request multiplexing and header compression. We also cover how these benefits solve the problems that exist in HTTP/1 client/server interactions. In the course of all of this, you'll write a small HTTP/2 server in Node and see these benefits in action.

HTTP/2 offers more than cheaper requests and compressed headers. It also offers an optional feature called Server Push, which can be used to send specific assets to visitors without them having to ask. When used intelligently, Server Push speeds up the loading and rendering of your website. You'll learn how this feature works and how to use it.

HTTP/2 also has an impact on *how* you optimize your website, so this chapter covers how to change your optimization practices to be better performing on HTTP/2 connections. Because many of your visitors may still be working with browsers that use HTTP/1, I'll show a proof of concept of how you can serve content optimally for both HTTP/1 and HTTP/2 users from the same web server. Let's get started!

11.1 Understanding why we need HTTP/2

The need for HTTP/2 is due to the shortcomings of HTTP/1, which is now a legacy protocol that's ill-equipped to handle the demands of modern websites. To know why we need a new protocol, we need to understand the problems inherent in HTTP/1. In this section, you'll explore those problems and how HTTP/2 solves them. Then you'll go on to write an HTTP/2 server in Node.

11.1.1 Understanding the problem with HTTP/1

HTTP originated in 1991 with the invention of HTTP/0.9. This protocol, which was capable of using only a single method (`GET`), was originally designed for a much simpler "web" of electronic documents. These documents, written in HTML, were imbued with the ability to link to other documents via anchor tags. The HTTP 0.9 protocol achieved this goal admirably.

As time marched on, two new implementations of HTTP with additional capabilities and methods (such as `POST` for submitting form data) were added. These versions were v1.0 and v1.1, which were standardized in 1996, with support added in most browsers shortly thereafter.

HTTP/1 thus became the workhorse of the web for many years since. What occurred next, however, was that the web transformed from serving simple HTML documents to complex sites and applications, a phenomenon illustrated in figure 11.1.

The increasing complexity of the web means that richer experiences are possible through higher-quality media and content. The problem, however, is that an ongoing race between this complexity of content and the ability to serve it in a high-performing way has been raging since the first web developer dared to believe that the web was for more than serving static text documents. Although developers have come up with ingenious ways around common performance problems in HTTP/1, three significant



Figure 11.1 The 1996 (left) and 2016 (right) incarnation of the *Los Angeles Times*

problems still plague the protocol: head-of-line blocking, uncompressed headers, and nonsecure websites.

HEAD-OF-LINE BLOCKING

The biggest problem plaguing HTTP/1 client/server interactions is a phenomenon known as *head-of-line blocking*. This manifests from the HTTP/1 protocol's inability to handle more than a small batch of requests at the same time (typically, six at once). Requests are responded to in the order that they're received, and new requests for content can't begin downloading until all requests in the initial batch have finished. Figure 11.2 illustrates this problem.

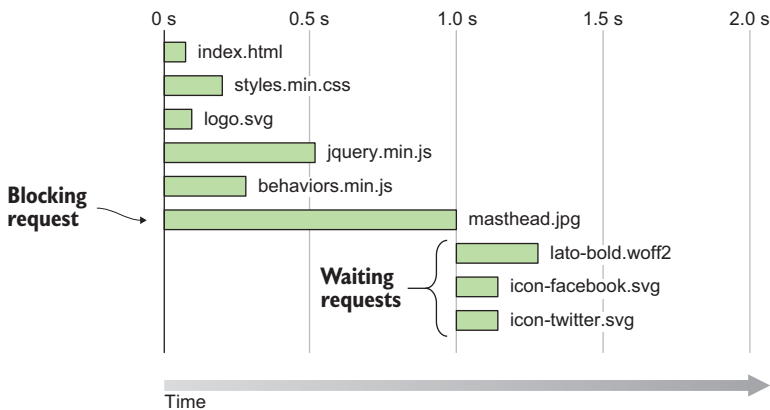


Figure 11.2 The head-of-line blocking problem as shown in a batch of nine requests. The first batch of six requests is fulfilled in parallel, but the remaining batch can't start downloading until the largest file (*masthead.jpg*) in the first batch finishes downloading. This problem can cause delays in load times.

One way to ameliorate this problem on the front end is to bundle files. Reducing requests minimizes the negative effects of the head-of-line blocking problem, but it's a rather hacky sort of antipattern in that when one piece of the bundled content changes, the entire bundled asset must be downloaded again rather than only the relevant portion of it that has changed.

Another rather hacky way around this request limit is to use a technique called *domain sharding*. This technique gets around the maximum simultaneous request limit of six per domain by spreading requests across domains. With two domains serving content, twelve requests can be fulfilled at once. With three, up to eighteen requests could be accommodated simultaneously. Although this technique *is* effective, it requires a significant investment of resources, both temporal and financial, to implement. It's not an option for every organization.

Some success has been achieved in mitigating this problem on the server side. For example, persistent HTTP connections (keep-alive connections) lighten the load by reusing a single connection to fulfill multiple batches of requests. This method falls short, however, in that it doesn't solve the head-of-line blocking problem. A technique called *HTTP pipelining* was designed to address this problem by serving all requests in parallel rather than in batches, but its implementation was met with significant challenges that prevented it from being successful.

UNCOMPRESSED HEADERS

As you know by now, when you request assets from a web server, headers accompany the request to and response from the web server. These headers describe many aspects of the request and response for an asset, most of which are expressed in redundant fashion.

A perfect example of this is the Cookie request header. Cookies are often used to track user sessions, and as such, contain session IDs. Imagine a web page that comprises around 60 assets, each carrying a cookie with a session ID of 128 bytes in length. Every single request must upload an additional 128 bytes of data to the server indicating the domain for which the cookie is valid. This isn't much when spread across a few requests, but imagine a page with 60 requests with this cookie attached. The client must send 7.5 KB of extra data to the server across all of those requests. Figure 11.3 illustrates this concept.

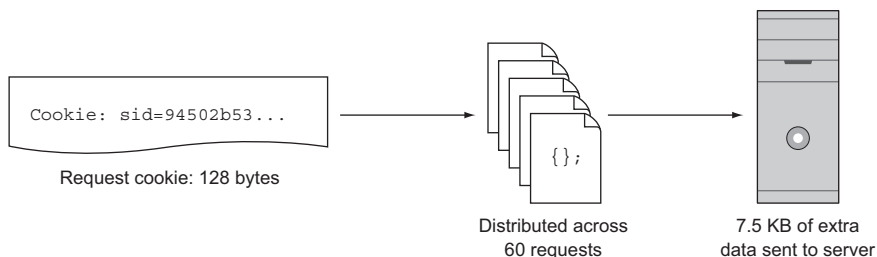


Figure 11.3 A session ID cookie of 128 bytes distributed across 60 requests, adding up to a total of 7.5 KB of extra data sent to the web server

This doesn't occur with just request headers. It also occurs with response headers, so the data that these headers can pile up occurs on both the trip to *and* from the web server.

"Doesn't server compression fix this?" is a question you may have as a result of reading earlier chapters. The answer is an emphatic no. Server compression compresses only the *body* of the response, *not* its response headers. Although the body of a response is undoubtedly the largest part of the payload, the uncompressed data present in response headers is certainly an aspect worthy of consideration. HTTP/1 fails to address this problem—yet another ill that plagues web developers who care about making websites faster.

NONSECURE WEB SITES

Although not necessarily a performance issue per se, HTTP/1 servers aren't required to implement SSL for their visitors. In an increasingly dangerous world in which data is regularly stolen and used by hackers to impersonate people, securing your website is necessary to ensure privacy and a safer web-browsing experience for your visitors.

Because HTTP/1 doesn't mandate implementation of SSL, it's entirely optional to implement. If security measures are optional, people likely won't implement them. People are reluctant to change, and will usually do so only when forced to, or when an adverse event occurs. Although this failure couldn't be foreseen when HTTP was first developed, this requirement can't be retroactively applied to force site owners to secure their websites. The cat is already out of the bag, so to speak.

These aren't the only problems that HTTP/1 causes, but they're major issues worthy of concern. Fortunately, solutions to these problems are inherent to HTTP/2 because of the way the protocol was designed.

11.1.2 Solving common HTTP/1 problems via HTTP/2

HTTP/2 didn't appear out of thin air. It was preceded in 2012 by a protocol named *SPDY* (pronounced *speedy*), which was developed by Google to address the limitations of HTTP/1. When the first draft of the HTTP/2 specification was written, its writers capitalized on the advances of SPDY and used it as a starting point. Now that HTTP/2 support has grown considerably, Google has removed SPDY support from Chrome 51 and later, and other browsers will follow suit. Let's see how HTTP/2 fixes the two problems outlined earlier in HTTP/1.

NO MORE HEAD-OF-LINE BLOCKING

Unlike HTTP/1, which has a limit on the number of requests it can satisfy before it can begin responding to other requests in the queue, HTTP/2 can satisfy many more requests in parallel by implementing a new communication architecture. Unlike HTTP/1, which uses multiple connections to transfer assets, HTTP/2 uses one connection capable of handling many, many more requests in parallel. A connection consists of these components in the following hierarchy:

- *Streams are bidirectional communication channels between the server and the browser.* A single stream consists of a request to and a response from the server. Because streams are encapsulated by the connection, many assets can be downloaded in parallel within the same connection by using multiple streams.
- *Messages are encapsulated by streams.* A single message is the rough equivalent of an HTTP/1 request to or response from the server, providing the mechanism needed to request assets, and to receive the content of those assets from a web server.
- *Frames are encapsulated by messages.* A frame is a delimiter in a message that indicates the type of data that follows. For example, a HEADERS frame in a response message indicates that the following data represents the HTTP headers for the response. A DATA frame in a response message indicates that the following data is the content of the requested asset. Other frame types exist, such as the PUSH_PROMISE frame that's used for Server Push, which is covered later in this chapter.

When you visualize this process, you get something like figure 11.4.

Because of this design, requests to HTTP/2 servers are cheap. Cheap enough, in fact, that bundling isn't worth the effort and could even lead to slower load times in some scenarios. The next main section covers the specifics, but the bottom line is that you no longer have to resort to antipatterns such as image spriting and bundling (though those techniques are useful in HTTP/1 clients and servers, many of which are still in the wild).

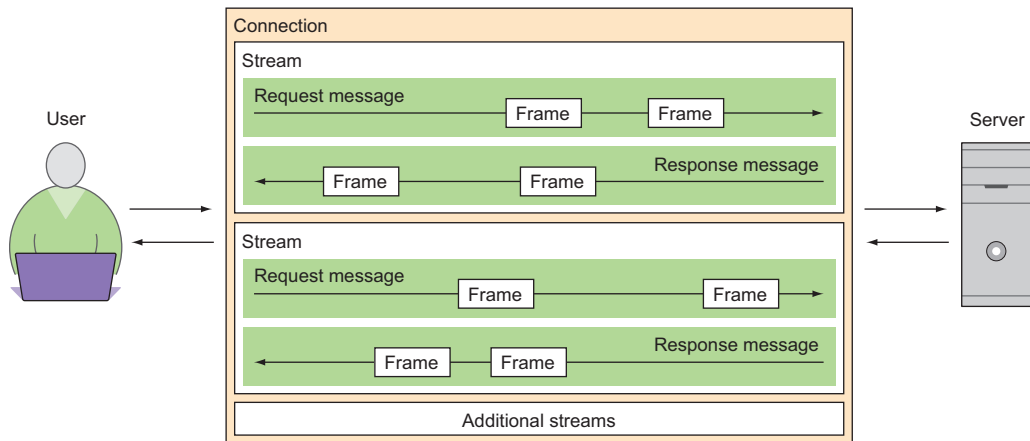


Figure 11.4 The anatomy of an HTTP/2 request. One connection houses multiple bidirectional streams, which in turn contain multiple messages that request and receive assets. These messages are delimited by frames, which in turn describe the content of messages (headers, response bodies, and so forth).

HEADER COMPRESSION

Headers in HTTP/1 are uncompressed, even when server compression is used, as I said in the preceding section. Server compression transforms only the asset, and not the headers that accompany it. Although headers don't make up the bulk of a page's total payload, they can add up quickly.

HTTP/2 fixes this problem by incorporating a compression algorithm called *HPACK*. HPACK not only compresses header data, but also strips redundant headers by creating a table to store duplicates. In HTTP/1 request headers, you'll notice that headers with longer content, such as *Cookie* and *User-Agent*, are unnecessarily attached to every single request, creating a potentially huge set of redundant data that must be transferred along with the request to the server (as illustrated in figure 11.3).

HPACK deduplicates headers via a table that uses indexes to keep track of duplicate header data found across requests. This is structured like a typical database table with indexes. When new header values are discovered, they're compressed, stored in the table, and given a unique identifier. If additional headers are discovered that match any previously indexed headers, the relevant identifier in the table's index is referenced rather than redundantly stored. Figure 11.5 shows this behavior.

This process is done on the client side when requests are made, and the table is transferred to and disassembled by the server and used to build the response. The server then repeats this process for the response headers, replies, and the client disassembles the server-generated response table, and applies the headers to the responses for each downloaded asset. The result is deduplicated headers and compressed data that's presented in the same way HTTP/1 headers are, making the process transparent. Your website loads a little faster for the trouble.

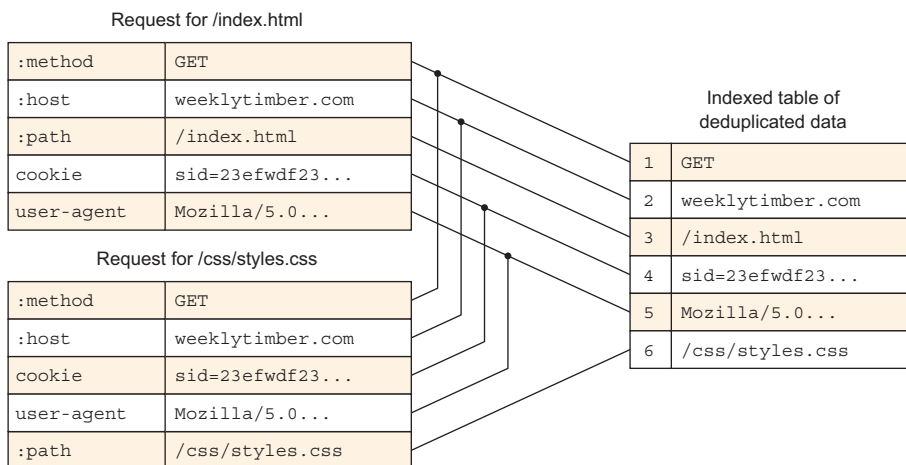


Figure 11.5 HPACK header compression in action. Headers are stored in an indexed table. Identical headers discovered in later requests for the same page are tied to an index in the table to avoid duplication of that data, whereas headers with new data are stored as new entries in the table.

HTTPS IS GUARANTEED

Although not necessarily specific to performance, browsers that support HTTP/2 are doing so with the de facto mandate that any communication over HTTP/2 must be secure. This has been a somewhat controversial requirement, but the mandate isn't without benefits. As more servers adopt HTTP/2 and implement SSL, the internet as a whole will become increasingly secure.

SSL performance overhead

A common complaint of SSL is that it has a measurable performance impact on TTFB due to the time it takes to set up an SSL connection between the server and client. Because HTTP/2 carries all data over one connection rather than several, this process needs to occur only once, rather than multiple times as in HTTP/1. Modern hardware has also made this process rather trivial. The result? You can stop worrying about SSL performance and worry instead about providing a secure browsing experience to your users. For more information, check out <https://istlsfastyet.com>.

The cost of SSL certificates couldn't be any cheaper. Certificate providers are offering reliable signed certificates for as little as \$5 a year for one domain. If that price is still too steep for you, you can get free certificates through Let's Encrypt (<https://letsencrypt.org>). I've found that the process is a bit more involved than setting up purchased certificates (depending on the hosting environment).

The point is this: You no longer have an excuse to deny your users a secure browsing experience, nor do you have a choice if you want to use HTTP/2. Get on board and encrypt!

Next, you'll get your feet wet with HTTP/2 by writing your own HTTP/2 server in Node.

11.1.3 Writing a simple HTTP/2 server in Node

Your client contact from Weekly Timber has asked whether you can do anything further to make the site faster than it is now. You know you can't make any promises, but you have a pretty good hunch that HTTP/2 might be an option.

Of course, downloading and installing an HTTP/2 server such as Apache or Nginx locally is a bit of a pain, and your current hosting provider doesn't offer the service. So how can you test the Weekly Timber site on an HTTP/2 server if one isn't readily available? Easy. You can use the `spdy` package from npm to write a simple HTTP/2 server in Node!

"Wait, SPDY?" I know, but don't worry! The name of the package is somewhat of a misnomer. Although SPDY is one of the protocols this package supports, it also supports HTTP/2, which is what you need for these purposes. To get started, you need to download some code with git, as you've done many times before in this book:

```
git clone https://github.com/webopt/ch11-http2.git
cd ch11-http2
npm install
```


This downloads all the source code and installs the Node packages you need for the HTTP/2 server you're going to write, including the `spdy` package. Once everything is good to go, open your text editor, and create a new file named `http2.js` in the root folder of the website. In this file, enter what you see here.

Listing 11.1 Importing modules needed for the HTTP/2 server

```

var fs = require("fs"),
    path = require("path"),
    http2 = require("spdy"),
    mime = require("mime"),
    pubDir = path.join(__dirname, "/htdocs");
  
```

Imports the SPDY module for HTTP/2 functionality. (points to `http2 = require("spdy")`)

Imports the filesystem module for reading files. (points to `fs = require("fs")`)

Imports the path module for normalizing paths. (points to `path = require("path")`)

Imports the MIME module for determining content types. (points to `mime = require("mime")`)

Sets the root directory to serve files from. (points to `pubDir = path.join(__dirname, "/htdocs");`)

Here you import the Node modules you need in order to write the server behavior. You also establish the root directory from which you'll be serving files. Unlike past examples, you'll serve the files out of a separate nested folder called `htdocs`, which contains the Weekly Timber website. With your modules imported, you need to set up the SSL certificates, because HTTP/2 requires SSL. The code you downloaded already comes with the certificate files you need in the `crt` folder. This listing shows you how to configure the server to point to these files.

Listing 11.2 Setting up SSL certificates on the server

```

var server = http2.createServer({
  key: fs.readFileSync(path.join(__dirname, "/crt/localhost.key")),
  cert: fs.readFileSync(path.join(__dirname, "/crt/localhost.crt"))
});
  
```

Creates an instance of the HTTP/2 server. (points to `var server = http2.createServer({`)

The key and certificate files needed for SSL (points to `key: fs.readFileSync(path.join(__dirname, "/crt/localhost.key")),` and `cert: fs.readFileSync(path.join(__dirname, "/crt/localhost.crt"))`)

The JavaScript written here sends the locations of the certificate files to the HTTP/2 server, which enables it to securely communicate with the browser. The next listing provides the bulk of the work that the server will do.

Listing 11.3 Writing the HTTP/2 server behavior

```

}, function(request, response){
  var filename = path.join(pubDir, request.url),
      contentType = mime.lookup(filename);

  if((filename.indexOf(pubDir) === 0) &&
      fs.existsSync(filename) &&
      fs.statSync(filename).isFile()){
    response.writeHead(200, {
      'Content-Type': contentType
    });
    response.end(fs.readFileSync(filename));
  } else {
    response.writeHead(404, {
      'Content-Type': 'text/plain'
    });
    response.end('404 Not Found');
  }
}
  
```

The content type of the asset. (points to `contentType = mime.lookup(filename);`)

The request handler. (points to `function(request, response){`)

The filesystem path to the asset. (points to `var filename = path.join(pubDir, request.url),`)

Checks if the asset exists. (points to `if((filename.indexOf(pubDir) === 0) && fs.existsSync(filename) && fs.statSync(filename).isFile()){`)

```

    response.writeHead(200, {
      "content-type": contentType,
      "cache-control": "max-age=3600"
    });

    var fileStream = fs.createReadStream(filename);
    fileStream.pipe(response);
    fileStream.on("finish", response.end);
  }
  else{
    response.writeHead(404);
    response.end();
  }
});

server.listen(8443);

```

Sends a 200 response to the client.

Sets the Content-Type response header.

Caches assets for 1 hour.

Sends the asset to the user.

If the asset isn't found, a 404 response is sent.

Starts the server on port 8443

After you enter this code into your text editor, run the script in your terminal with the following command:



```
node http2.js
```

When this script runs, you can head to <https://localhost:8443/index.html> in your browser and see that the client's website loads for you.

Making an exception

The certificate that's provided with the source code you downloaded from GitHub is unsigned. Therefore, when you go to view the client's website on your local server, you'll be prompted with an SSL warning in your browser. Make an exception or ignore the warning, and you'll be able to proceed just fine. Just remember that you should *always* use a valid signed certificate on a production web server!

So that's all fine and dandy, but how can you tell whether the protocol is HTTP/2? Easy! Open your Network tab in Chrome's Developer Tools and right-click the column headers to ensure that the Protocol column is selected. You'll then see something like figure 11.6.

Name	Method	Status	Protocol	Scheme
 index.html	GET	200	h2	https
 styles.min.css	GET	200	h2	https

Indicates asset transferred over HTTP/2

Figure 11.6 The Network panel in Chrome's Developer Tools indicating assets transferred over HTTP/2. Assets transferred over HTTP/1 will have the value `http/1.1` in this field.

11.1.4 Observing the benefits

The benefits at first may not seem so obvious on a site such as Weekly Timber. Discerning what the benefits are, even on your local machine where you're not experiencing any network bottlenecks, can be difficult. One thing you *can* see is that requests now execute in parallel, as opposed to serialized batches, as shown in figure 11.7.

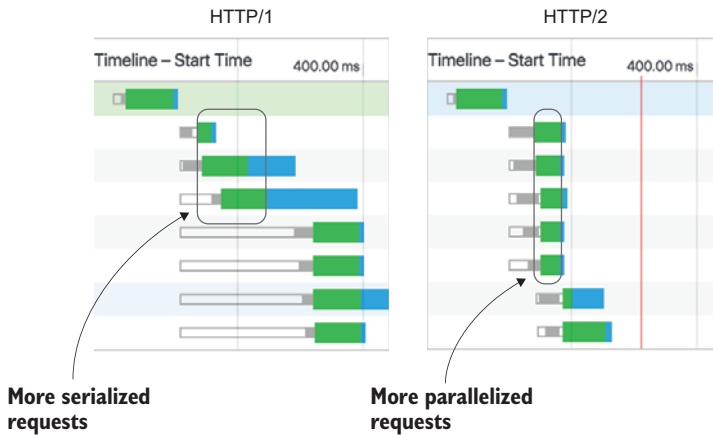


Figure 11.7 The effect on asset downloads on HTTP/1 (left) versus HTTP/2 (right): downloads in HTTP/2 are parallelized more than in HTTP/1, meaning that they begin roughly at the same time.

You can observe this phenomenon yourself by running the `http1.js` script in the root folder of the website, navigating to `https://localhost:8080/index.html`, and comparing its behavior in the Network tab to the HTTP/2 server you've just written. One thing you might notice is that if you use a throttling profile to compare the performance of the two protocols on your local machine, the load time of the client's website is about the same for each. This occurs because although creating an artificial bottleneck is good for testing some scenarios, it's not a good tool for comparing the performance of one protocol over another. Both of these servers run on your local machine rather than on a remote server somewhere, and are serving no other traffic than the requests you're making to them. The best way to get an idea of the performance of HTTP/2 versus HTTP/1 is to run two servers on a remote host somewhere: one running HTTP/2 and the other running HTTP/1. From there, you can observe the differences.

That's an unreasonable thing to ask, so I've done the hard work for you. I set up two versions of the client's website: one running on HTTP/1 at <https://h1.jeremywagner.me> and another running on HTTP/2 at <https://h2.jeremywagner.me>. Feel free to visit those URLs and do your own testing to see how they perform in the wild as opposed to in the comparatively clinical setting of your local machine. Figure 11.8 shows my testing on each protocol across all five pages of the client's website.

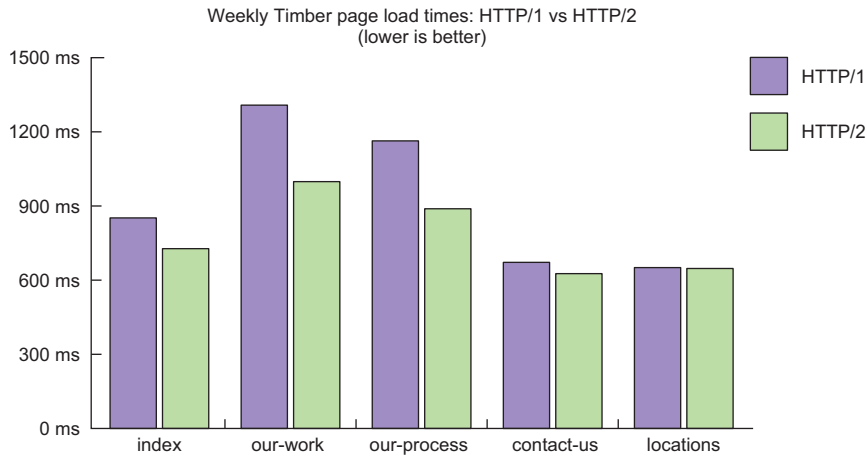


Figure 11.8 Comparing page-load times on the Weekly Timber website on HTTP/1 versus HTTP/2

I saw a 24% improvement in total load time on pages with many assets, such as in the `our-work.html` and `our-process.html` pages. On typical pages such as `index.html` and `contact-us.html`, I saw improvements of 15% and 7%, respectively. The one page that didn't improve in performance was `locations.html`, which had few assets in comparison to other pages.

The gains from header compression are harder to quantify. But if you open <chrome://net-internals#timeline> in Chrome, you can see the effect of header compression on request size. Uncheck every option on the left except for Bytes Sent, load the page for each protocol, and you'll see a comparison of request sizes. Figure 11.9 shows this tool at work.

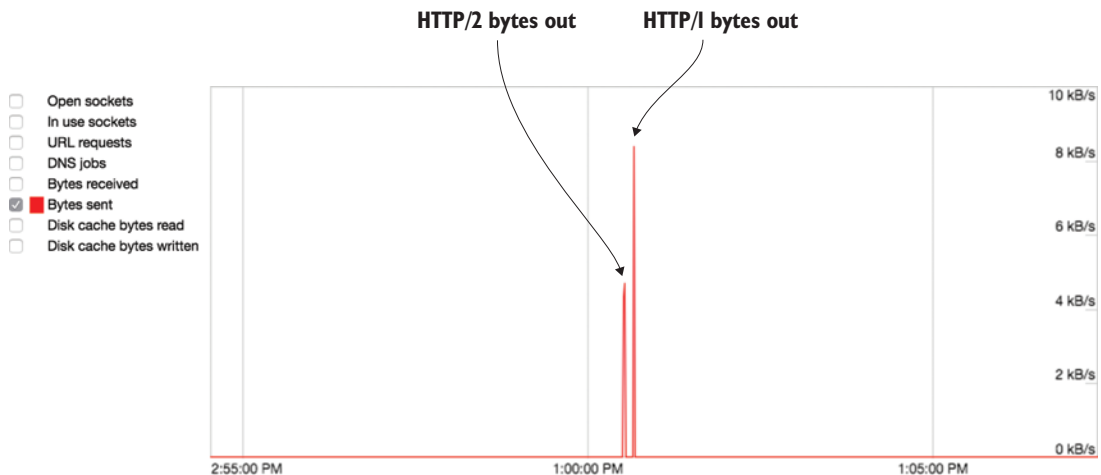


Figure 11.9 A comparison of the bytes sent during an HTTP/2 session versus that of an HTTP/1 session

As you can see in figure 11.9, fewer bytes are sent to the server when you use HTTP/2, due to header compression. Although the tool in the net internals panel doesn't reveal the exact size, you can see that the improvement is about 50%. This smaller request payload means that the user will spend less time waiting for that first byte of content to arrive.

All of these benefits are realized by switching to HTTP/2. They don't require any special optimization techniques or changes in your code. They're simply a benefit of implementing the protocol.

Next, you'll learn how optimization techniques that you currently know change when running your site over HTTP/2, and why your approach needs to be different.

11.2 Exploring how optimization techniques change for HTTP/2

"Oh great," you're thinking. "I got this book on web performance, and everything I learned in it is wrong." Not necessarily. Although some techniques covered in this book are antipatterns when applied to HTTP/2 client/server interactions, they don't necessarily impede the protocol's performance—but they can affect the effectiveness of your caching policy. The rules of optimization techniques on HTTP/2 are this simple:

- Techniques that *reduce* the size of assets are things you should *still* do on HTTP/2. These are techniques such as minification, server compression, and image optimization. Reducing the size of an asset contributes to lower load times, always and forever.
- Techniques that *combine* files are things you should *stop* doing on HTTP/2. Although useful in alleviating latency in HTTP/1 client/server interactions, requests are much cheaper in HTTP/2, and combining files can have an adverse effect on your caching effectiveness.

The first rule speaks for itself, but we need to talk a bit more about the second rule, how caching is affected by concatenating resources, and what antipatterns fit under concatenating.

11.2.1 Asset granularity and caching effectiveness

When you set out to squeeze every last drop of performance out of HTTP/1, you adopted many techniques that, albeit effective, are hacky in HTTP/2 environments. The main category of techniques that hurt HTTP/2 performance are those that rely on concatenation. *Concatenation* is the process of combining files in order to reduce the number of HTTP requests that are sent. As I said before, this is great for HTTP/1, but it could harm performance on HTTP/2.

Why *is* this, exactly? The answer is in caching. As you learned in chapter 10, caching helps reduce the payload of a page on visits subsequent to the first. The problem isn't in caching itself, however, because a properly configured caching policy will operate whether or not we concatenate files. The problem is that when you concatenate files, you're reducing the efficiency of your caching when assets change. This is true of

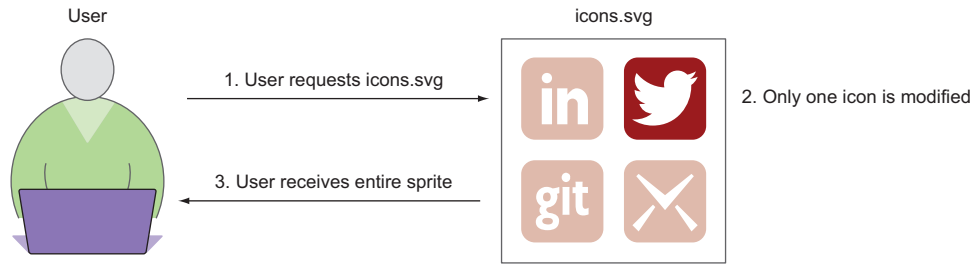


Figure 11.10 Concatenation can reduce caching efficiency. One of four icons in the image sprite is modified, but even though 75% of the file content remains unmodified, the user will be forced to download the entire asset instead of just the changed portion.

both protocols, but when using HTTP/1, you were willing to forego a certain level of efficiency to minimize load times for users visiting a site for the first time.

Here's a good example of how concatenation hurts caching: Say you have an image sprite of icons, and you need to update just one icon in the set. Even though you've changed only one of the icons, the asset is monolithic, and must thus be purged from browser caches. This creates an issue as the entire file must be invalidated and retrieved, even though only a portion of it has changed. This is illustrated in figure 11.10.

In HTTP/1 optimization workflows, we accepted this suboptimization as just a bump on the road to building fast websites. Now that HTTP/2 affords us cheaper connections, you don't have to make the choice between shorter page-load times for first-time visitors and efficient caching. You can have your cake and eat it, too! Next up, let's look at techniques you should avoid when your site is on HTTP/2.

11.2.2 Identifying performance antipatterns for HTTP/2

As I said before, the only detriment to performance on HTTP/2 servers occurs when you concatenate assets in one fashion or another, which will reduce the efficiency of your caching policy. But concatenation isn't relegated to only one technique. This section enumerates the techniques that fall under this category and the reasons you should avoid them.

BUNDLING CSS AND JAVASCRIPT

A common use of concatenation is in bundling CSS and JavaScript files. This serves a few purposes on HTTP/1 connections. The first is the obvious one outlined already: fewer requests benefit HTTP/1 client/server interactions. The second is that it can aid in making subsequent page loads faster by loading all of your assets up front.

The second reason also works the same way for HTTP/2-powered websites, but because requests are cheaper, it makes more sense to make your CSS and JavaScript more granular. For CSS, this is simple: create different CSS files for each unique page template. That way, you can segment and load your CSS on pages that need it, and you

can limit the impact of any CSS updates to a particular template, which will maximize the effectiveness of your caching policy.

As for splitting up JavaScript files, that depends on your website and the functionality it requires. You can split these scripts by what page template they apply to, but that may not work for all websites, because pages may share common functionality. Do what seems logical. There's no right answer that works for every single website, except that bundling on HTTP/2-driven websites isn't optimal.

IMAGE SPRITES

Yes, I did cover and recommend this technique in chapter 6, but *only* if you're going to be stuck hosting your website on an HTTP/1 server. Otherwise, image sprites carry the same consequences as any other form of concatenation.

One odd scenario you may run into is that an image sprite may be slightly smaller than the sum of its individual image files. If you find this to be the case, stick to keeping your images separate rather than spriting them for HTTP/1. The benefit you'll realize from your caching policy when you need to update an image later will be worth the trade-off when your visitors won't be forced to download an entire sprite of images to get one image that's been changed.

ASSET INLINING

This one is slightly trickier to explain, but it still falls under the umbrella of concatenation: asset inlining occurs when you take a CSS, JavaScript, or binary asset and embed it in your HTML and/or CSS. For text assets, this means you're copying and pasting some CSS inside `<style>` tags, or doing the same with JavaScript inside `<script>` tags. You can also inline SVG images straight into HTML.

Inlining binary assets *can* be achieved using something called the *data URI scheme*. This method encodes data into a base64 string and combines it with a content type. This string can then be used in something like an `` tag, as demonstrated in figure 11.11.

The encoded string is truncated in the preceding example, but you get the gist. The data URI scheme can be used in many places, such as `<link>` tags, `` tags, in CSS `url` references, basically any place that allows you to reference an external asset. If you're interested in encoding files of your own, numerous sites on the web enable you to do so. An example of such a site is Base64 Decode and Encode (<https://www.base64encode.org>).

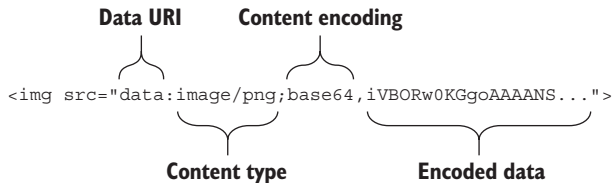


Figure 11.11 An example of a data URI. The scheme begins with the data URI, followed by the encoded data's content type, the name of the encoding scheme, and the encoded data (truncated in this example).

Using data URI schemes seems like a good idea and may have some usefulness in a few scenarios, but they're inefficient. The encoded string is often larger than its source, sometimes by 33% or more.

Worse yet, all methods of asset inlining suffer from an inability to be cached effectively. Inlined data that's used across more than one document is redundantly downloaded and is cached only in the context of the document it's contained within.

When we discussed the critical CSS technique in chapter 4, we recommended inlining the CSS for the above-the-fold content in `<style>` tags. This is still an effective technique to promote faster paint times on HTTP/1 client/server interactions and should be considered in those cases. With HTTP/2, however, you don't need it. In fact, an HTTP/2 feature covered in the next section, called Server Push, allows you to gain the benefits of inlining while maintaining high effectiveness of your browser cache.

I know that I sound like a broken record, but you don't inline assets in HTTP/2 for the same reason you shouldn't use image sprites or bundle your CSS and JavaScript. The simple way to break this down is that if you're aiming to reduce requests, do so only for websites running on HTTP/1. For HTTP/2, keep your assets as granular as is practical for your workflow.

11.3 *Sending assets preemptively with Server Push*

In the past, if you wanted to speed up page rendering, you'd inline assets into your HTML. It wouldn't appreciably decrease the size or overall load time of the page, but it would have the potential benefit of decreasing the rendering time of a web page.

Of course, as said before, asset inlining is an antipattern that, although effective for sites running on HTTP/1, ruins the effectiveness of a good caching policy for the inlined content. We're willing to accept this shortcoming on HTTP/1 for the trade-off of a lower perceived load time.

So if you're not supposed to inline assets on HTTP/2, how do you achieve the benefits of inlining? With a new feature called Server Push! In this section, you'll learn about this feature, how it works, how to use it in your Node-driven HTTP/2 server, and the benefits of its use.

11.3.1 *Understanding Server Push and how it works*

Server Push is a feature available in HTTP/2 that enables you to realize the benefits of asset inlining while still maintaining the granularity of your page assets. It's a mechanism that allows the server to "push" assets the user hasn't explicitly requested but needs in order to render a page.

When Server Push is used, the user makes a request for a page. Then, depending on its configuration, the server can reply with the contents of the requested document, along with assets that the server was configured to "push" to the client.

Imagine that a user goes to the Weekly Timber website (which runs on an HTTP/2 server for the purpose of this example) and requests `index.html`. Predictably, the

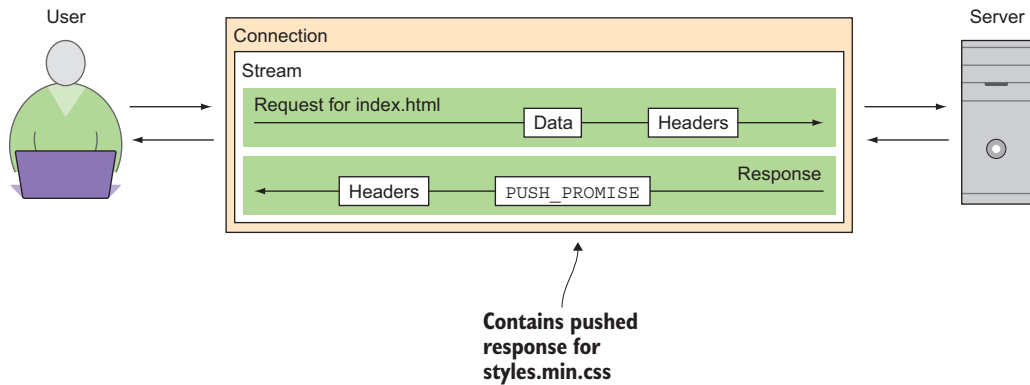


Figure 11.12 The anatomy of a Server Push event: the user requests index.html, and the server responds with a PUSH_PROMISE frame that contains the pushed copy of styles.min.css, as per its configuration.

server receives the request for index.html and constructs a response for it. But let's also imagine that the owner of the web server has configured the server to *also* respond with a copy of styles.min.css, which is the site's style sheet. This reduces the amount of time that the user has to wait for the styles to download, because the server doesn't have to wait for the client to request styles.min.css. The server sends it in parallel along with the response for index.html. Figure 11.12 shows this process in action.

You can see how this feature behaves like asset inlining, because the two assets are pushed to the client at the same time when the server responds with the contents of the HTML. Of course, you're not limited to a single asset at a time. You can push as many assets as you'd like.

With a rudimentary understanding of the way Server Push works, you can now go on to learn how various servers implement it, including how to write your own Server Push behavior in your Node HTTP/2 server!

11.3.2 Using Server Push

Using Server Push can be challenging if you're not sure how your web server implements it. This short section explains how Server Push is used on some commonly used web servers, how to use Server Push on your Node web server, as well as how to tell whether it's working.

HOW SERVER PUSH IS TYPICALLY INVOKED

For web servers such as Apache that are running HTTP/2, Server Push is invoked by setting up a Link HTTP response header when a specific asset is requested:

```
Link: </css/styles.min.css>; rel=preload; as=style
```

If this looks familiar to you, it's because the preload resource hint HTTP header covered in chapter 10 takes on the same format. That said, *don't* confuse this with the <link> tag used for resource hints! You *don't* need to modify your site's HTML to use

Server Push, as it's a server-driven behavior. If you're modifying your HTML to add a preload resource hint via the `<link>` tag, and expecting Server Push to just magically work, you're mistaken. It'll just preload the resource for the user; it *won't* invoke a Server Push event.

For web servers that implement Server Push in the fashion shown previously, it'll take the asset specified inside the angle brackets and serve it simultaneously with the asset that the header was set for (usually an HTML file). The `as` attribute is there only to inform the browser of the nature of the pushed content, and isn't necessary for Server Push to work. In this case, a value of `style` is used to indicate that the pushed resource is a CSS file.

Informing the browser of other pushed content types

If you want to inform the browser of the nature of the pushed content for content types other than `style`, you can find a full list of content types to use with `as` in the W3C specification for this feature at <http://mng.bz/r840>.

This implementation is convenient and works well. The following listing shows how to push a CSS file to a client that requests `index.html` on a server.

Listing 11.4 Pushing content in Apache when a user requests an HTML file

```
<Location /index.html>
  Header add Link "</ch11-http2/htdocs/css/styles.min.css>; rel=preload;
  ➡ as=style"
</Location>
```

This configuration directive is pretty straightforward: When the user navigates to `index.html` on the server, the `Link` header is set, and `styles.min.css` is pushed. The specific way you set this header for web servers that implement Server Push depends on the server software you use. For instance, our Node example does it quite differently, and you have to implement the Server Push behavior on your own.

WRITING SERVER PUSH BEHAVIOR IN NODE

Because you're responsible for implementing all of your own behavior for your Node HTTP/2 server, you can't set a `Link` header and expect Server Push to work. You need to write the logic that pushes content to the client.

Mercifully, this logic isn't much different from the typical way you serve assets to the user. The only difference is that you create a separate push response on requests for specific assets. For example, the Weekly Timber website has a style sheet named `styles.min.css`, and maybe it would make sense to push that CSS to the client whenever it requests an HTML file. Makes sense, especially because every HTML file on Weekly Timber's site references this CSS.

So let's do just that. Pull up the `http2.js` web server you wrote earlier in this chapter, navigate to the request handler function that serves assets to the client, and enter the code in the following listing right before the `response.writeHead` call that sends the 200 response to the client.

Listing 11.5 Writing a Server Push response in a Node HTTP/2 server

```

if ((filename.indexOf(pubDir) === 0) &&
    fs.existsSync(filename) &&
    fs.statSync(filename).isFile()) {
    if (filename.indexOf(".html") !== -1 && response.push) {
        var pushAsset = "/css/styles.min.css",
            pushAssetFSPath = path.join(pubDir, pushAsset),
            pushAssetContentType = mime.lookup(pushAssetFSPath);

        response.push(pushAsset, {
            response: {
                "content-type": pushAssetContentType,
                "cache-control": "max-age=3600",
                "link": "<" + pushAsset + ">; rel=preload; as=style"
            },
            function(error, stream) {
                if (error) {
                    return;
                }

                pushStream = fs.createReadStream(pushAssetFSPath);
                pushStream.pipe(stream);
                pushStream.on("finish", stream.end);
            }
        });

        pushStream = fs.createReadStream(pushAssetFSPath);
        pushStream.pipe(stream);
        pushStream.on("finish", stream.end);
    }
}

```

Checks if the request is for an HTML file.

Initiates the push for the CSS file specified.

The Link response header for the asset

The callback function for the push response

If an error is encountered during the push, abort.

Closes the stream and signals the end of the push response.





Ensures the asset exists on disk.

Variable definitions needed to set the Link header

Response headers for the CSS file's content type and its caching policy

Creates a readable stream of the CSS and pushes it.

With this code, you push `styles.min.css` to users whenever they request an HTML file. It can be a bit tricky to tell whether assets are being pushed. Since version 53, Chrome indicates whether an asset has been pushed in the Network panel's Initiator column. If you restart the server and go to `https://localhost:8443/index.html`, you'll see something like figure 11.13.

Name	Status	Protocol	Type	Initiator	Size
 index.html	200	h2	document	Other	4.7 KB
 css?family=Lato:400,700,300,900	200	h2	stylesheet	index.html:9	933 B
 styles.min.css	200	h2	stylesheet	Push / index.html:10	18.5 KB
 icon-facebook.svg	200	h2	svg+xml	index.html:22	301 B

Pushed asset

Figure 11.13 The Network tab in Chrome indicating a pushed asset by way of the Push keyword in the asset's Initiator column

Other browsers are less obvious about it. Firefox shows the asset as being read from the browser cache, and Edge shows assets with the TTFB measurement omitted. These representations are technically true, but not obvious. These browsers will likely be updated in the future to indicate explicitly whether an asset has been pushed.

Profiling HTTP/2 Server Push on the command line

A solid way of finding out whether an asset is being pushed is to use the `nghttp` command-line client, which shows you all of the frames in an HTTP/2 session. If you see a `PUSH_PROMISE` frame and the contents of the pushed asset, you'll know for sure that Server Push is working. You can learn more about `nghttp` at <https://nghttp2.org/documentation/nghttp.1.html>.

With Server Push working properly for the client website's CSS, let's measure performance.

11.3.3 Measuring Server Push performance

Measuring Server Push performance is tricky. Locally, it can be difficult. Possible ways to measure performance include disabling throttling, or hosting the website on a remote HTTP/2 server, and measuring performance under real network conditions. The problem in testing locally without throttling is that it's not a realistic scenario.

In my case, I set the website on a remote HTTP/2 server to test Server Push. To test, I set up a version of the client's website with Server Push enabled at <https://serverpush.jeremywagner.me>, which pushes the site's CSS to the user on all HTML pages. A version of the same site without Server Push was set up for comparison at <https://h2.jeremywagner.me>. The test results between the two can be seen in figure 11.14.

When the CSS was pushed to the user, the page began painting about 19% faster than when not pushed. On my broadband connection, this translated to about an 80 ms increase in rendering speed. This is nothing to sneeze at, especially when you consider

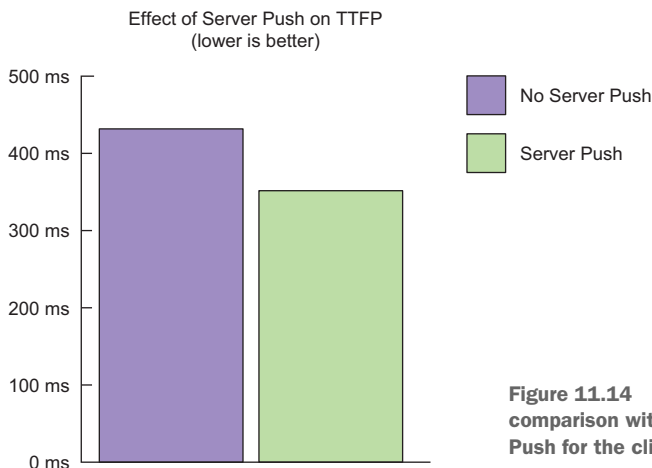


Figure 11.14 Time to First Paint comparison with and without Server Push for the client website's CSS

that devices on slower mobile networks will realize a proportionately larger increase in rendering speed. Considering that it takes little effort on most servers to use, it's practically a gimme for the performance-minded web developer.

Although Server Push is difficult to use “incorrectly,” so to speak, you need to remember a few basic guidelines when you use it:

- *You're not limited to pushing just one asset.* You can modify your Node HTTP/2 server code to push more than one asset, or add more Link headers for other HTTP/2 servers to push multiple assets at a time.
- *Don't push what you don't need.* Makes sense, right? It may be tempting to push everything and the kitchen sink to the client, but push only what makes sense. A good rule of thumb is to push assets that are used on *all* pages of your site.
- *You can push assets that aren't on the current page.* Yes, you can push an asset that isn't even needed by the current HTML document. You may decide to do this in order to preload an asset that's on a page you anticipate that the user may navigate to. Of course, this can be dicey, and you may end up wasting the user's bandwidth. If you don't have a good reason for doing this, then *don't do it*.

A note on server push and browser caches

Sometimes server push can end up pushing content that's already been cached by the client. For a possible server side mechanism that can help you mitigate this potential problem, check out an article I've written at CSS-Tricks at <https://css-tricks.com/cache-aware-server-push/>.

Now that you've had a chance to play with Server Push and have witnessed its benefits, you're ready to cap off this chapter by learning to simultaneously optimize for both HTTP/2 and HTTP/1 on the same server!

11.4 Optimizing for both HTTP/1 and HTTP/2

You've heard the expression *having your cake and eating it too*. This section is all about allowing your website's visitors to have all the benefits of your optimization techniques, whether or not their browser can support HTTP/2.

This section covers what happens when a visitor arrives at your HTTP/2-powered site using an HTTP/2-incapable browser. You'll learn how to use Google Analytics to determine the segment of your users incapable of using HTTP/2, and how to transform your Node server to accommodate optimization techniques for both protocols.

It should be made clear before we proceed that this section is illustrating a proof of concept. The methods in this section aren't necessarily designed to be a robust way of solving this problem, but rather that a solution to the problem *exists*. If you discover a more efficient approach using tools apart from those used here, give it a go and see how it works. Let's begin!

11.4.1 How HTTP/2 servers deal with HTTP/2-incapable browsers

Up to this point, you may have been curious about how browsers that don't support HTTP/2 can communicate with HTTP/2 servers. The fact is that under the hood of every HTTP/2 server is an HTTP/1 server waiting for a client to come along that doesn't support HTTP/2.

It's like those alarms you see in buildings that read, "In case of emergency, break glass." Except here, it's more apt to say, "In case of HTTP/2-incapable browser, downgrade to HTTP/1."

When a user with an older browser comes by your HTTP/2-powered site, the connection initially begins as an HTTP/2 conversation. But if the client hints to the server that it wants to downgrade the connection to HTTP/1, the server will comply and use the older version of the protocol. Figure 11.15 illustrates this process.

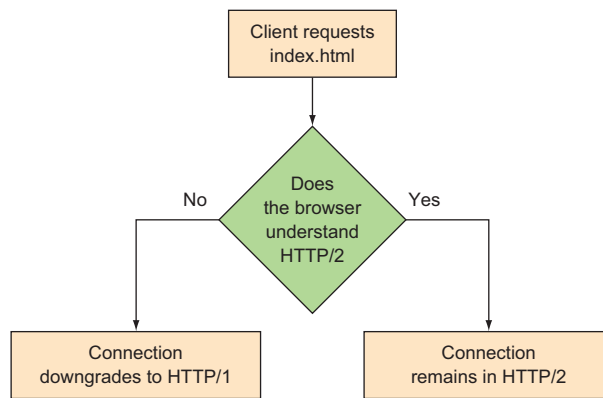


Figure 11.15 The anatomy of an HTTP/2 negotiation. The client requests an asset, and the server then checks whether the browser is capable of using HTTP/2. If so, it proceeds accordingly. If not, the connection downgrades to HTTP/1.

You can use the dual nature of this design to serve optimized web experiences to both classes of user: the one who can use HTTP/2, and the one who can't. You can do so reliably because this is a part of the specification for HTTP/2. In order for a server to be considered fully compliant with the specification, it needs to be able to downgrade the protocol for older browsers.

Of course, you need to be able to see whether a two-pronged effort is worth your time. After all, it's no small effort to optimize for both protocol versions. To do this, you'll lean on data from Google Analytics to help inform your decision.

11.4.2 Segmenting your users

Statistics are your friends, especially in a case like this when you want to see whether it's worth the trouble to adopt two sets of optimization practices. This means combining two sources: Can I Use (caniuse.com) and Google Analytics.

Can I Use is an exhaustive resource for determining the browser support for certain features, of which HTTP/2 is one. If you navigate to the site and enter HTTP/2 in the top search box and click the Usage Relative toggle button, you'll see something like figure 11.16.

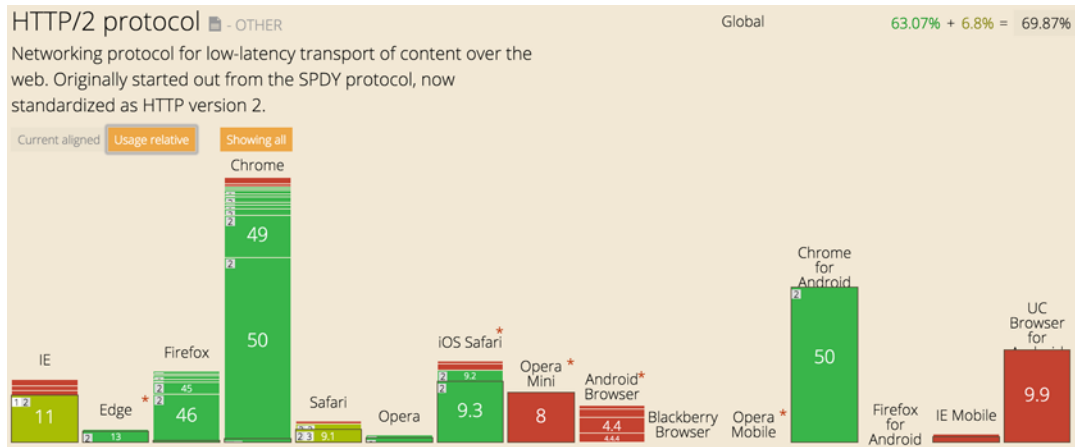


Figure 11.16 The Can I Use website displaying support of HTTP/2 by browser

With this tool, you can determine the browser support for features. Browsers that support HTTP/2 appear in green, those that don't appear in red, and those with partial support appear in a muted green color. For example, IE11 shows partial support. When you hover over the IE11 entry, you'll see that HTTP/2 support is limited to IE11 only on Windows 10.

This isn't all you can do with this tool. You can import visitor data from your Google Analytics account, and see what segments of your audience support or don't support a particular feature (such as HTTP/2!). To import data, click the Settings button at the top of the page. This opens a menu on the left side of the page. Beneath that is a section where you can import your data from Google Analytics, which looks like figure 11.17.



Figure 11.17 The section to import your site data from Google Analytics

When you click the Import button, you have to authorize Can I Use to access your analytics data. After you authorize, you then choose the website you want to import data from. When you do this, the visuals representing the level of support for a feature change to reflect the capabilities of your site visitors. In this case, you can see the segment of your users who can support HTTP/2. In the upper-right corner of the box containing this data, you see a percentage showing the level of support that looks like figure 11.18.

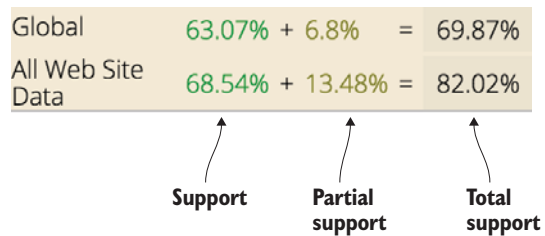


Figure 11.18 The support formula for a feature on Can I Use after Google Analytics data has been imported. All Web Site Data is the data imported from Google Analytics.

When I import the data for Weekly Timber, I can see that around 18% of the site's visitors are using

browsers that don't support HTTP/2. This is a conservative estimate too, because partial support in this case doesn't imply that every user in that segment can use HTTP/2, either.

With this data in hand, a decision has to be made. If roughly 20% of users visiting Weekly Timber can't use HTTP/2, it seems rational to optimize for both segments of users. That said, it's important to remember that this data will change depending on the site, and at the time you import it. Make informed decisions with the data you have for your site!

With data in hand, you can now move on to serving assets in a way that benefit users of both protocols. Let's optimize!

11.4.3 Serving assets according to browser capability

How you serve content to users based on their HTTP/2 support hinges on detecting the protocol version in use. If you can do this, you can modify the way the HTML is sent to the browser.

Remember that the only real difference in optimization techniques between HTTP/1 and HTTP/2 is that the former performs better when assets are concatenated. The latter performs best when assets are more granular. When you can change the HTTP response containing the HTML in flight and modify the way assets are loaded, you'll have full control over which assets are delivered for each segment of users.

Before you get started, you need to install a package for Node called `jsdom`. This package allows you to modify the content of HTML on the server in Node, much as you would in the browser using familiar methods available in the `window.document` object. To install this plugin, go to the root folder of the client's website and type `npm i jsdom`, and then `git checkout -f protocol-detection` to update your code, and you'll be ready to start tinkering. If you want to skip ahead to the finished code, you can do so by typing `git checkout -f protocol-detection-complete` in your terminal.

DETECTING THE PROTOCOL VERSION

The first step in all of this is to determine which version of the protocol is running in the request. To test this, you need a modern browser such as Chrome or Firefox that supports HTTP/2, and another older browser that can't use HTTP/2. In my case, I went to <https://modern.ie> and grabbed a free Windows 7 virtual machine with IE10 installed on it. If you don't have a paid virtualization program such as VMWare, you can grab a free program called VirtualBox at www.virtualbox.org. If you have an older web browser installed on your computer, use that to test instead.

Detecting the protocol version is trivial. The `spdy` package you used to make the HTTP/2 server in Node has a member in the `request` object called `isSpdy`. Although the name of this property isn't as straightforward as you'd like, it indicates whether the current connection is using HTTP/2. Open `http2.js` in your text editor, and after the `contentType` variable is declared, add the following bold line from the following listing.

Listing 11.6 Detecting the HTTP version

```
var filename = path.join(pubDir, request.url),
    contentType = mime.lookup(filename),
    protocolVersion = request.isSpdy ? "http2" : "http1";
```


This single line of code is the piece of logic you'll use from here on out to determine how to adjust the HTML in order to accommodate users of both protocols. We check the value of the `request.isSpdy` object member, and based on its Boolean value, we assign a string value of "http2" or "http1". Next, you use the `jsdom` package to add a class to the `<html>` tag when the user has downgraded to HTTP/1.

Why would you do this? Simple: When you add a class to the `<html>` tag that identifies when the user's protocol has downgraded, you can change the way assets are delivered in our CSS. By default, your CSS is written to deliver assets in an optimal fashion for HTTP/2 first, so you'll need to make modifications only if the protocol is downgraded.

ADDING THE HTTP/1 CLASS

Adding the HTTP/1 class to the document with `jsdom` requires a little bit of shuffling around of our HTTP/2 server's code. When you switched to the protocol-detection branch of code, the Server Push logic should have been erased, which will make things much simpler than if you wanted to accommodate everything.

Look for the `response.writeHead` call in the code that sets the headers for each request. After that, you can insert the code in the following listing.

Listing 11.7 Adding a class to the `<html>` tag when the HTTP version downgrades

```

if(protocolVersion === "http1" && filename.indexOf(".html") !== -1){
  fs.readFile(filename, function(error, data){
    jsdom.env(data.toString(), function(error, window){
      window.document.documentElement.
        classList.add(protocolVersion);
      var newDocument = "<!doctype html>" + window.document.
        documentElement.outerHTML;
      response.end(newDocument);
    });
  });
}

```

Reads the asset from the filesystem.

Checks if HTTP/1 is used and if the asset requested is an HTML file.

jsdom reads the content of the file and creates a window object for us to work with.

The http1 class is added to the `<html>` tag.

The modified document content is sent to the browser.

This puts you on the right path, but you need to make further modifications to the server code. Because you're intercepting and modifying the contents of the request if the request has downgraded to HTTP/1, and if the requested asset is an HTML document, problems will arise when requests come in that don't match that criteria. You need to isolate other requests with an `else` condition.

Listing 11.8 Isolating other requests that don't require modification

```

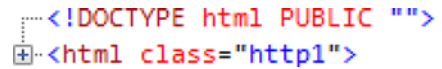
else{
  var fileStream = fs.createReadStream(filename);
  fileStream.pipe(response);
  fileStream.on("finish", response.end);
}

```

Be sure to place this `else` condition right after the initial `if` that checks for the protocol version, and HTML asset request. Failure to do this will trigger a server error.

When you've finished, run the server with Node and pull up the site in a modern browser.

You'll see that the response is unchanged. But if you pull up the website in a browser that doesn't support HTTP/2, you'll notice that the `<html>` tag has the `http1` class added to it, as shown in figure 11.19.



```
<!DOCTYPE html PUBLIC ''>
<html class='http1'>
```

Figure 11.19 The `<html>` tag is modified on the server when the web server downgrades to HTTP/1.

With this in place, you now have full control over the way you deliver assets based on the presence of this protocol version class. If you open `styles.min.css` in the `htdocs/css` folder and scroll to the bottom, you'll see that some styles are written that use an image sprite (`sprite.svg`) when the protocol version is HTTP/1. If you go to `https://localhost:8443/index.html` and compare the number of requests between a modern browser (such as Chrome) that's capable of using HTTP/2 versus that of a browser that can't (such as IE10), you'll notice that the HTTP/2-capable browser processes four more requests for SVG images. The HTTP/2-incapable browser will have used the image sprite instead, lowering its number of requests for images by three.

This isn't the only way to reduce requests on the server side. Next, you'll use `jsdom` to further reduce requests for HTTP/1 clients by replacing the numerous scripts with a single, concatenated version that will comport to the optimization requirements of HTTP/1.

REPLACING MULTIPLE SCRIPTS WITH CONCATENATED ONES FOR HTTP/1 USERS

The Weekly Timber site has quite a few scripts in it. Seven, in fact. One of these is a CDN-hosted copy of jQuery, though, so you should look to optimize the delivery of really just six.

Consider that the client-imposed maximum number of parallel requests in HTTP/1 server/client communications is usually six. If you can replace these six scripts with one concatenated version for your HTTP/1 users, you may be able to improve the delivery of site assets for those users. The following listing shows the `<script>` tags that are on each page.

Listing 11.9 Scripts on the Weekly Timber site

```
<script src="https://code.jquery.com/jquery-2.2.4.min.js"
integrity="sha256-BbhdLvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
crossorigin="anonymous">
</script>
<script src="js/jquery.colorbox.min.js"></script>
<script src="js/colorbox-init.min.js"></script>
<script src="js/scooch.min.js"></script>
<script src="js/carousel.min.js"></script>
<script src="js/lazyload.min.js"></script>
<script src="js/collapsible-content.min.js"></script>
```

**Candidate scripts for
concatenation on
HTTP/1 connections**

The first script is the CDN-hosted copy of jQuery, which you want to keep referencing from the CDN. The last six, however, can be concatenated for the benefit of your HTTP/1 visitors. I've already provided a concatenated version of these scripts in the `js` folder called `scripts.min.js`. The goal is to use `jsdom` to transform this markup on the server to look like the contents of the next listing when the site is accessed over HTTP/1.

Listing 11.10 Optimal handling of scripts for HTTP/1 on the Weekly Timber website

```
<script src="https://code.jquery.com/jquery-2.2.4.min.js"
      integrity="sha256-BbhdLvQF/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
      crossorigin="anonymous">
</script>
<script src="js/scripts.min.js"></script>
```

Concatenated version of all candidate scripts in listing 11.9

Seems easy enough, but you first need to write a little code to change this markup on the server. Because your code is HTTP/2-first, in that your scripts are referenced more granularly by default, you need to transform the markup shown in listing 11.9 to that in listing 11.10. This will be done in the section of your server code where you transform the response in the event that the user is requesting an HTML document over an HTTP/1 connection. This code is in the following listing, which shows added lines in bold.

Listing 11.11 Transforming the delivery of scripts based on the HTTP version

```
jsdom.env(data.toString(), function(error, window){
    window.document.documentElement.classList.add(protocolVersion);

    var scripts = window.document.
        querySelectorAll("script:not([crossorigin])"),
        jQueryScript = window.document.
            querySelector("script[crossorigin]"),
        concatenatedScript = window.document.createElement("script");
        concatenatedScript.src = "js/scripts.min.js";

    for(var i in scripts){
        scripts[i].remove();
    }

    jQueryScript.parentNode.
        insertBefore(concatenatedScript, jQueryScript.nextSibling);

    var newDocument = "<!doctype html>" +
        window.document.documentElement.outerHTML;
    response.end(newDocument);
});
```

Grabs the `<script>` element that references the CDN-hosted copy of jQuery.

Grabs all of the non-CDN hosted scripts in the document.

Removes all scripts that aren't CDN-hosted.

Creates a new `<script>` element referencing the concatenated scripts.

Adds the new `<script>` element pointing to the concatenated script.

After making this change, restart the server. Then, open the site in an HTTP/2-capable browser and see that your scripts are granular as they were. If you open the site in an older browser such as IE10, you'll see something like figure 11.20.

https://code.jquery.com/jquery-2.2.4.min.js	GET	200
/js/scripts.min.js	GET	200

Figure 11.20 The scripts for the client website delivered in concatenated fashion for HTTP/1 browsers

This approach, though not robust and only a proof of concept, illustrates the fact that you can serve assets in a way that benefits everyone. If this is an approach you want to take, you need to keep in some considerations in mind.

CONSIDERATIONS

If you decide to embark on this errand, you have to make some decisions about how you want to implement optimizations tailored for various protocol versions.

The first decision depends on whether you even *need* to tailor your website to accommodate all segments of user capability. Some sites are simple enough that both protocol versions will serve them equally well, but some are decidedly more complex. Another aspect to consider depends on the capabilities of your audience, which we covered earlier in this section.

The second decision depends on the technologies available to you. For instance, on a PHP server, you can use the `$_SERVER["SERVER_PROTOCOL"]` environment variable to discover the protocol version. The following listing shows the use of this variable to affect how assets are served.

Listing 11.12 Serving assets by protocol in PHP

Scripts to load if the protocol version is HTTP/1.	<pre> <script src="https://code.jquery.com/jquery-2.2.4.min.js" integrity="sha256-BbhdLvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44=" crossorigin="anonymous"> </script> <?php if(\$_SERVER["SERVER_PROTOCOL"] == "HTTP/1.1"){ ?> <script src="js/scripts.min.js"></script> <?php }else{ ?> <script src="js/jquery.colorbox.min.js"></script> <script src="js/colorbox-init.min.js"></script> <script src="js/scooch.min.js"></script> <script src="js/carousel.min.js"></script> <script src="js/lazyload.min.js"></script> <script src="js/collapsible-content.min.js"></script> <?php } ?> </pre>	Scripts to load regardless of the protocol version.
		Scripts to load if the protocol version is HTTP/2.

How you do this depends on the server-side language you use. Some implementations of this logic will be more or less straightforward, depending on the language.

Now that you've learned about HTTP/2, how it differs from its predecessor, and had a chance to get your hands dirty with it, it's time to go over some of the knowledge you've picked up in this chapter.

11.5 Summary

This chapter exposed you to a few concepts regarding HTTP/2 and the ways it contrasts with its predecessor, HTTP/1. While reading this chapter, you've learned the following key concepts:

- HTTP/1 was designed to serve a much simpler function in its infancy than what web developers eventually forced it to do. As a result, problems such as a lack of connection multiplexing and uncompressed headers contributed to performance degradation.
- HTTP/2 evolved out of Google's experimental SPDY protocol, and grew to address the problems of limitations in parallel connections and uncompressed headers.
- You got a chance to get your feet wet and witness the benefits of HTTP/2 firsthand by writing a Node-powered HTTP/2 server.
- HTTP/2 necessitates some changes in the way we approach optimization. Optimization practices that encouraged developers to combine assets such as bundling, image sprites, and asset inlining are now antipatterns in this new version of the protocol.
- Server Push allows you to enjoy the performance benefits of asset inlining, but with none of the icky problems that come with inlining, such as maintainability and caching.
- If you're running an HTTP/2 server, and a significant segment of your users are using browsers that support only HTTP/1, you can tune your asset delivery to be optimal for everyone.

You're nearing the end of this book. Before we part ways, we'll spend the last bit of our time together on how to automate many of the optimization practices you learned so far by using a JavaScript task runner called gulp. By the time you close this book, you'll not only be armed with knowledge of techniques to make your websites blazing fast, but also able to automate those techniques!