10 Fine-tuning asset delivery

This chapter covers

- Understanding compression basics, the impacts of poor compression configuration, and the new Brotli compression algorithm
- Using caching to improve the performance of your website for repeat visitors
- Exploring the benefits of CDN-hosted assets
- Verifying the integrity of CDN resources by using Subresource Integrity

Until now, we've spent much of this book talking about techniques specific to the constituent parts of web pages, such as CSS, images, fonts, and JavaScript. Understanding how to fine-tune the delivery of these assets on your website can give it an added performance boost.

In this chapter, we'll spend time investigating the effects of compression, both for good and ill, as well as a new compression algorithm named Brotli. We'll also touch on the importance of caching assets, an optimal caching plan for your website, and how to invalidate stubborn caches when you update your content or release new code.

Moving away from web server configuration, we'll learn how your website can benefit from using assets hosted on Content Delivery Networks (CDNs), which are geographically dispersed servers. You'll also learn how to fall back to locally hosted assets in the unlikely event that a CDN fails, and how to verify the integrity of CDN assets using Subresource Integrity.

Finally, we'll enter the realm of resource hints, which are enhancements you can use in some browsers via the `<link>` tag in your HTML, or via the `Link` HTTP header. Resource hints give you the power to prefetch DNS information for other hosts, preload assets, and prerender entire pages. Without any further ado, let's dive in!

10.1 Compressing assets

Recall that chapter 1 showed the performance benefits of server compression. To review, *server compression* is a method in which the server runs content through a compression algorithm prior to transferring it to the user. The browser sends an `Accept-Encoding` request header that indicates the compression algorithm(s) supported by the browser. If the server replies with compressed content, the `Content-Encoding` response header will specify the compression algorithm used to encode the response. Figure 10.1 shows this process in action.

As you delve further into this section, you'll learn basic compression guidelines and the pitfalls of poorly compressed configurations. You'll then learn about the new Brotli compression algorithm that's gaining support, and how it stacks up to the venerable gzip algorithm.

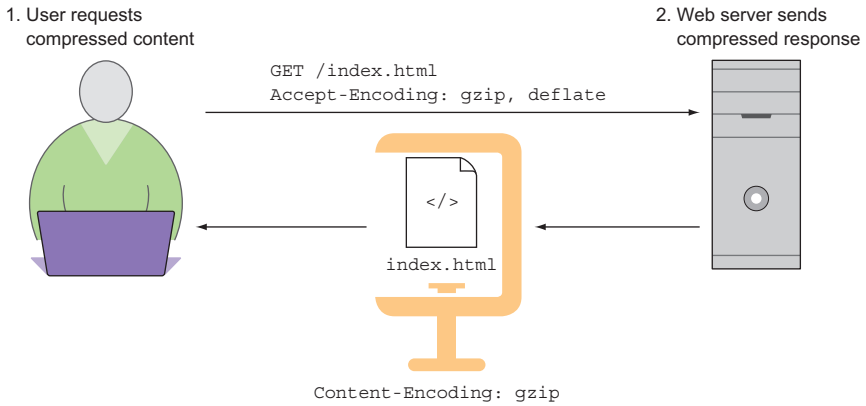

Figure 10.1 The user makes a request to the server for index.html, and the browser specifies the algorithms that are supported in the `Accept-Encoding` header. Here, the server replies with the compressed content of index.html, and the compression algorithm used in the response's `Content-Encoding` header.

10.1.1 Following compression guidelines

Compressing assets isn't as simple as "compress *all* of the things!" You need to consider the *types* of files you're dealing with and the level of compression you apply. Compressing the wrong types of files or applying too much compression can have unintended consequences.

I have a client named Weekly Timber whose website seems like a good subject for experimentation. You'll start by tinkering with compression-level configuration. Let's grab Weekly Timber's website code and install its Node dependencies:

```
git clone https://github.com/webopt/ch10-asset-delivery.git
cd ch10-asset-delivery
npm install
```

You aren't going to run the http.js web server at this time, as you have in chapters past. You need to make some tweaks to the server code first.

CONFIGURING COMPRESSION LEVELS

You may remember the `compression` module that you downloaded with `npm` when you compressed assets for a client's website in chapter 1. This module uses gzip, which is the most common compression algorithm in use. You can modify the level of compression that this module applies by passing options to it. Open http.js in the root directory of the Weekly Timber website and locate this line:

```
app.use(compression());
```

This is where the `compression` module's functionality kicks in. You'll notice that the invocation of this module is an empty function call. You can modify the compression level by specifying a number from `0` to `9` via the `level` option, where `0` is no compression and `9` is the maximum. The default is `6`. Here's an example of setting the compression level to `7`:

```
app.use(compression({
    level: 7
}));
```

Now you can start the web server by typing `node http.js`, and start testing the effects of this setting. Be aware that anytime you make a change, you need to stop the server (typically by pressing Ctrl-C) and restart it.

From here, experiment with the `level` setting and see the effect it has on the total page size. If you set `level` to `0` (no compression), the total page size of http://localhost:8080/index.html will be 393 KB. If you max it out to `9`, the page size will be 299 KB. Bumping the setting from `0` to `1` will lower the total page size to 307 KB.

Setting the compression level to `9` isn't always the best policy. The higher the `level` setting, the more time the CPU requires to compress the response. Figure 10.2 illustrates the effects of compression levels on TTFB and general load times as it applies to the jQuery library.

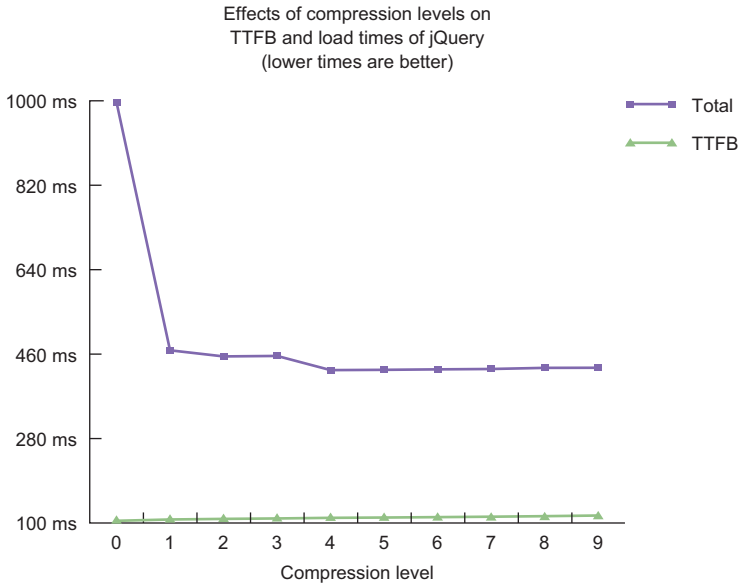

Figure 10.2 The effects of the compression-level setting on overall load times and TTFB when requesting jquery.min.js. Tests were performed on Chrome's Regular 3G network throttling profile.

You can see that the most dramatic improvement occurs when compression is turned on. But the TTFB seems to creep up steadily as you increase the level toward `9`. Overall load times seem to hit a wall around `5` or `6` and start to increase slightly. There's a point of diminishing returns, and worse yet, a threshold at which raising the compression level any further doesn't help matters any.

It's also worth noting that these tests aren't "in the wild" per se, but on a local Node web server, where the only traffic is from my local machine. On a busy production web server, extra CPU time spent compressing content can compound matters and make overall performance worse. The best advice I can give is to strike a balance between payload size and compression time. Most of the time, the default compression level of `6` will be all you need, but your own experimentation will be the most authoritative source of information.

Furthermore, any server that uses gzip should provide a compression-level setting that falls in the `0` to `9` range. On an Apache server running the `mod_deflate` module, for example, the `DeflateCompressionLevel` is the appropriate setting. Refer to your web server software's documentation for the relevant information.

COMPRESSING THE RIGHT FILE TYPES

In chapter 1, my advice on which types of files to compress was twofold: always compress text file types (because they compress well), and avoid files that are internally compressed. You should avoid compressing most image types (except for SVGs, which are XML files) and font file types such as WOFF and WOFF2. The `compression` module

you use on your Node web server doesn't try to compress everything. If you want to compress all assets, you have to tell it to do so by passing a function via the `filter` option, as shown here.

Listing 10.1 Compressing all file types with the compression module

```
app.use(compression({
    filter: function(request, response){
        return true;
    }
}));
```

Compress everything

Apply compression based on logic that you define

If you change your server configuration to reflect the preceding code, restart your server for changes to take effect. Navigate to http://localhost:8080, look in your Network panel, and you'll see that compression is now applied to everything. In my testing, I compared the compression ratios of JPEG, PNG, and SVG images across all compression levels. Figure 10.3 shows the results.

As you can see, PNGs and JPEGs don't compress at all. SVGs compress well because they're composed of compressible text data. This doesn't mean that only text-based assets compress well. As discussed in chapter 7, TTF and EOT fonts compress *very* well, and those are binary types. Because JPEGs and PNGs are already compressed when they're processed, compression confers no advantage. When it comes to these types of images, the best savings you'll get will be through the image optimization techniques you learned in chapter 6.

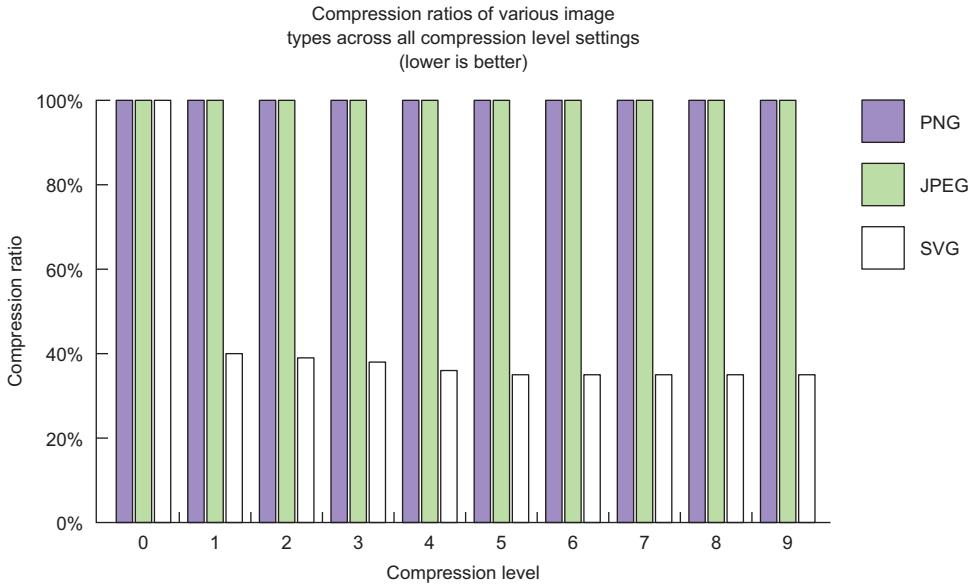

Figure 10.3 Compression ratios of PNG, JPEG, and SVG images across all gzip compression levels.

Worse yet, compressing internally compressed file types is *detrimental* to performance because more CPU time is required to compress uncompressible files. This delays the server from sending the response, which results in a lower TTFB for that asset. On top of *that,* the browser still has to decode the encoded asset, expending CPU time on the client side to perform work that has no benefit.

If you encounter a file type that you're not sure is compressible, do a bit of rudimentary testing. If you get little to no yield, it's a safe bet that compressing that type of file isn't likely to improve performance for your website's visitors.

Next, you'll look at the new Brotli compression algorithm, and how it compares to the venerable gzip.

10.1.2 Using Brotli compression

gzip has been the preferred method of compression for years, and that doesn't appear to be changing anytime soon. But a promising new contender has appeared on the scene, and its name is Brotli. Although its performance is comparable to that of gzip in some aspects, Brotli shows promise and is continually being developed. With this in mind, Brotli is worth your consideration. But before you look at Brotli performance, let's see how to check for Brotli support in the browser.

> **Want to learn more about Brotli?**
> Although this section covers Brotli with some depth, it's not complete. You can learn more about this emerging compression algorithm in an article I've written for *Smashing Magazine* at http://mng.bz/85Y1.

Checking for Brotli support

How do you know whether your web browser supports Brotli? The answer is in the `Accept-Encoding` request header. Brotli-capable browsers will compress content only with this algorithm over HTTPS connections. If you have Chrome 50 or later, open the Developer Tools, go to the Network tab on any HTTPS-enabled website, and look at the value of the `Accept-Encoding` request header for any asset. It'll look like figure 10.4.

If a browser supports Brotli, it'll say so in the `Accept-Encoding` request header by including the `br` token in the list of accepted encodings. When a capable server sees this token, it'll reply with Brotli-compressed content. If it doesn't, it should fall back to the next supported encoding scheme.

Next, you'll write a web server in Node that uses Brotli via the `shrink-ray` package. If you want to skip ahead, enter the command `git checkout -f brotli` in your terminal window in the root folder where you checked out the `ch10-asset-delivery` repository. Otherwise, continue on!

accept-encoding: `gzip, deflate, sdch, br`

Figure 10.4 Chrome showing support for Brotli compression with the `br` token

WRITING A BROTLI-ENABLED WEB SERVER IN NODE

Until this point in the book, you've been using the `compression` package to compress assets. Unfortunately, this package doesn't support Brotli. A fork of this package called `shrink-ray` does, however. Because Brotli also requires SSL, you need to install the `https` package as well:

```
npm i https shrink-ray
```

When this finishes, create a new file in the project's root directory named brotli.js, and enter the this content.

Listing 10.2 A Brotli-capable web server written in Node

```
var express = require("express"),
    https = require("https"),
    shrinkRay = require("shrink-ray"),
    fs = require("fs"),
    path = require("path"),
    app = express(),
    pubDir = "./htdocs";

app.use(shrinkRay({
    cache: function(request, response){
        return false;
    }
}));

app.use(express.static(path.join(__dirname, pubDir)));

https.createServer({
    key: fs.readFileSync("crt/localhost.key"),
    cert: fs.readFileSync("crt/localhost.crt")
}, app).listen(8443);
```

Imports the https module you need for Brotli to work.

Imports the Express framework.

Imports the shrink-ray module containing the Brotli compression middleware.

The relative path to the htdocs directory, your root web folder

Turns off caching in the shrink-ray module. You do this so that you can properly test the compression algorithm.

Instructs Express to use the shrink-ray compression module.

Creates a static file server to serve files out of the local directory.

Creates an HTTPS server instance.

Reads the SSL key and certificate necessary for the local HTTPS server to work.

Instructs the server to listen for requests on port 8443.

As usual with Node programs, the first part of your script imports everything you need, including your newly installed `shrink-ray` and `https` packages. From here, you create an Express-powered static web server much the same way you have in the past, except on an HTTPS server instead.

One thing you'll notice that's different with this server is that you're placing your assets in a separate subfolder named htdocs, below the project root. You do this because your certificate files are being held in the project root in the crt folder. If you serve your website files from a folder that also allows public access to the crt folder, that's pretty terrible for security. Although this is only a local website test, adhering to good security practices can't hurt.

After you've written this code, you can launch the server by typing `node brotli.js`. Provided there aren't any errors, you should be able to navigate to https://localhost:8443 and see the client's website come up.

Creating a security exception

When browsing to your local web server, you may receive a security warning. This is because the provided certificate isn't signed by a recognized authority. You can ignore this warning *in this case*, but always ensure that you're using a signed certificate on production web servers.

If you enabled Brotli encoding in Chrome, you can open the network request panel in the Developer Tools and look in the Content-Encoding column to see which requests are being compressed with Brotli, as shown in figure 10.5.

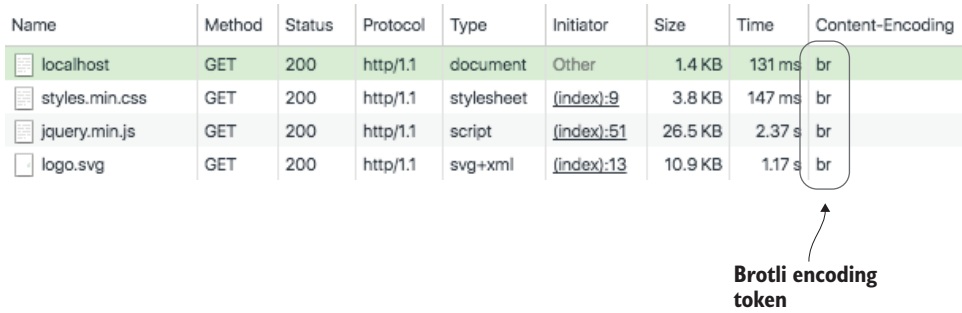

Figure 10.5 Brotli-encoded files can be seen in the network request panel in Chrome by looking for the `br` token in the Content-Encoding column.

Now that your local web server is compressing content with Brotli, you can go on to compare its performance to gzip.

Comparing Brotli performance to gzip

It's not enough to compare the default level of performance of Brotli to gzip. You have to compare the two along the entire spectrum of compression levels. As you may recall from earlier in this section, gzip's compression level is configurable by specifying an integer from `0` to `9`. With Brotli, you do something similar, except the range is from `0` to `11`. The way you do this is by passing in the `quality` parameter to the `shrinkRay` object in brotli.js, as shown in the following listing.

Listing 10.3 Configuring the Brotli compression level

```
app.use(shrinkRay({
     cache: function(request, response){
            return false;
```

```
    },
    brotli: {
            quality: 11
    }
}));
```

The compression level, configurable from 0 to 11. Higher values yield lower file sizes.

Like the `level` setting with gzip, the `quality` setting adjusts the Brotli compression level. Higher values yield lower file sizes. Figure 10.6 compares gzip to Brotli when compressing a minified version of jQuery.

In my testing, Brotli provided anywhere from a 3% to 10% improvement at comparable compression levels (except at a quality setting of `4`, where the result was the same as with gzip). The lowest size gzip could deliver was 29.4 KB, whereas Brotli's is 26.5 KB. This seems like a decent enough improvement for such a commonly used asset, but what does Brotli do to your TTFB? Compression is a CPU-intensive task, so it makes sense to measure Brotli's effect on this important metric. Figure 10.7 shows the average TTFB at all compression levels for both algorithms for the same jQuery asset.

The performance of Brotli and gzip are roughly similar until you hit the `quality` settings of `10` and `11`. At this point, Brotli gets a bit sluggish. Although these two settings yield lower file sizes, they do so at the expense of the user. The best setting in this instance is `9`.

That said, this is a performance comparison on a small JavaScript web server. Many web servers don't yet support Brotli. Nginx (https://www.ngnix.com) has a Brotli-encoding

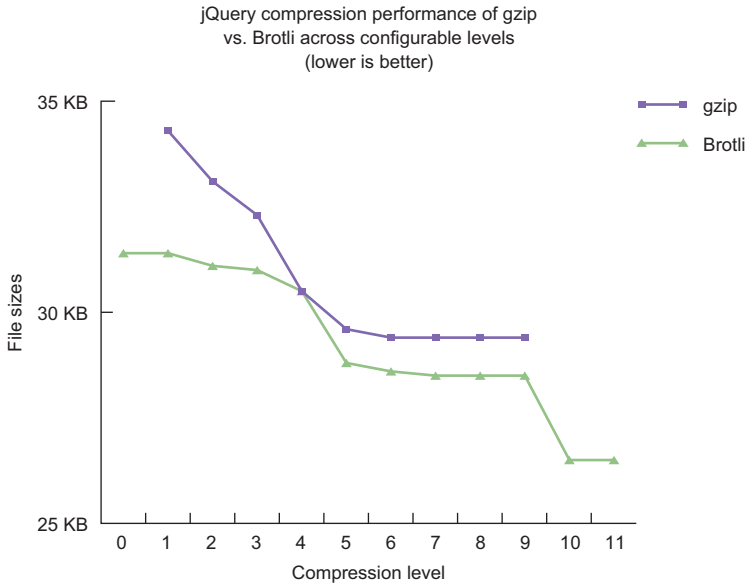

Figure 10.6 Performance of compressing the jQuery library with gzip versus Brotli compression across all comparable compression levels. (A gzip compression level of 0 is the same as no compression, and so is omitted.) Gzip's maximum compression level is 9, so comparisons to Brotli's quality settings of 10 and 11 aren't available.

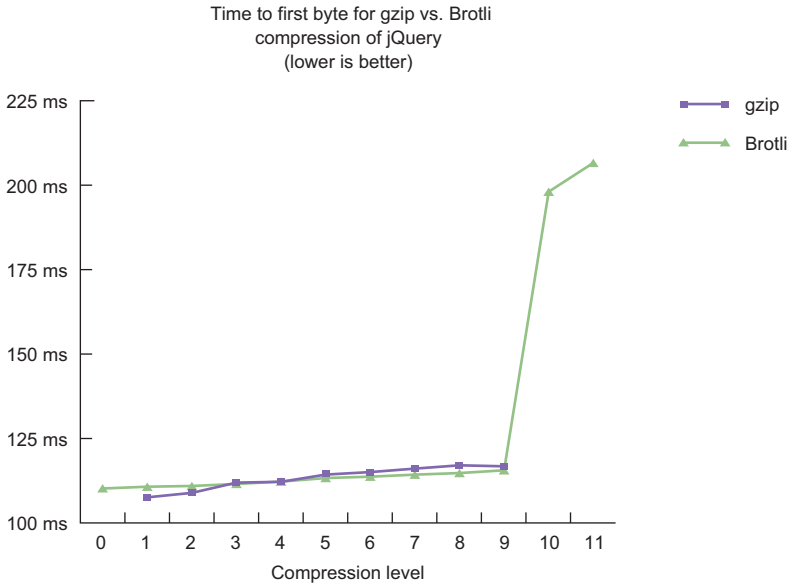

Figure 10.7 TTFB performance of gzip versus Brotli when compressing the jQuery library

module at https://github.com/google/ngx_brotli, and a `mod_brotli` module for Apache is being developed, but you have to know how to compile it on your own. If you're interested, you can check it out at https://github.com/kjdev/apache-mod-brotli.

> **Compression caching mechanisms**
>
> `shrink-ray` has a caching mechanism that you're disabling so that you can test compression performance. If your web server caches compressed content to speed up its delivery to the browser, you should take advantage of it. Be aware, though, that many compression modules, such as the popular Apache module `mod_deflate`, don't cache compressed content.

Support for this compression technology, although somewhat limited, is growing and shows promise for outperforming gzip. If you decide to provide it to your users, do a *lot* of testing on your website to ensure that you're not creating any performance issues. It's also important to remember that this is an ongoing project, and its performance can change in the future. It's, at the *very* least, worth looking into at the present.

10.2 Caching assets

Most of this book hasn't directly addressed caching, but not without reason: the first impression is the most important. You should optimize with the assumption that any one visit to your website is the first time that a particular user has been to your site. A bad first impression may be enough to prevent someone from returning.

Although this is a good assumption to operate under, it's also important to remember that a significant portion of your users could be returning visitors or are navigating to subsequent pages. In both scenarios, your site will benefit from a good caching policy.

In this section, you'll learn about caching. You'll see how to develop a caching strategy that provides the best performance for your website, as well as how to invalidate cached assets when you update your website's content. Let's begin by learning how caching works.

10.2.1 Understanding caching

Caching isn't difficult to understand. When the browser downloads an asset, it follows a policy dictated by the server to figure out whether it should download that asset again on future visits. If a policy isn't defined by the server, browser defaults kick in—which usually cache files only for that session. Figure 10.8 illustrates this process.

Caching is a powerful performance improvement that has an immense effect on page-load times. To see this effect, open Chrome's Developer Tools, go to the Network panel, and disable caching. Then go to the Weekly Timber website you downloaded from GitHub earlier and load the page. When you do, take note of the load time and the amount of data transferred. Then, re-enable caching, reload the page, and take note of the same data points. You'll see something similar to figure 10.9.

The behavior that you see here can be divided into two cache states: unprimed and primed. When a cache is *unprimed*, the page is being visited by a user for the first time. This

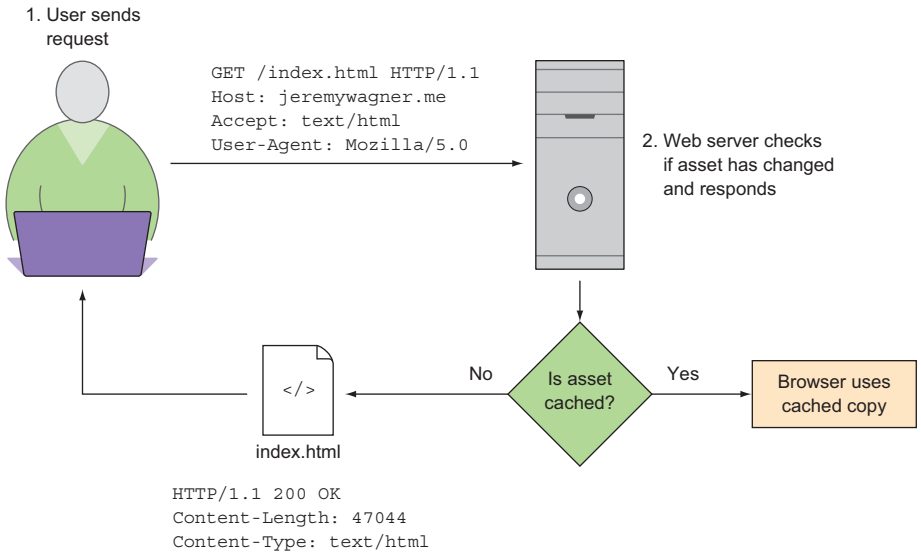

Figure 10.8 A basic overview of the caching process. The user requests index.html, and the server checks whether the asset has changed since the time the user last requested it. If the asset hasn't changed, the server responds with a 304 Not Modified status and the browser's cached copy is used. If it has changed, the server responds with a 200 OK status along with a new copy of the requested asset.

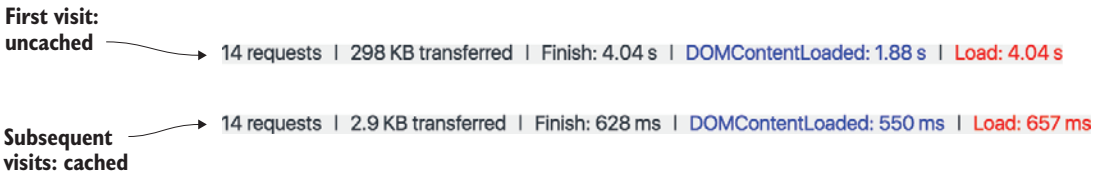

Figure 10.9 The load times and data payload of a website on the first uncached visit and on a subsequent visit. The page weight is nearly 98% smaller, and the load time is much faster, all due to caching.

occurs when the user has an empty browser cache, and everything must be downloaded from the server to render the page. The *primed* state exists when the user visits a page again. In this state, the assets are in the browser cache, and thus aren't downloaded again.

Naturally, you're curious about what drives this behavior. The answer to your burning curiosity is the `Cache-Control` header. This header dictates caching behavior in nearly every browser in use, and its syntax is easy to understand.

USING THE CACHE-CONTROL HEADER'S MAX-AGE DIRECTIVE

The easiest way to use `Cache-Control` is through its `max-age` directive, which specifies the life of the cached resource in seconds. A simple example of this in action is illustrated here:

```
Cache-Control: max-age=3600
```

Let's say that this response header is set on an asset named behaviors.js. When the user visits a page for the first time, behaviors.js is downloaded because the user doesn't have it in the cache. On a repeat visit the requested resource is good for the amount of time specified in the `max-age` directive, which is a 3,600 seconds (or, more intuitively, an hour).

A good way to test this header is to set it to a low value, something like 10 seconds or so. For our client's website, you can specify a `Cache-Control max-age` value by modifying http.js, and editing the line that invokes the `express.static` call to something like this:

```
app.use(express.static(__dirname, {
    maxAge: "10s"
}));
```

The value `10s` is shorthand for *10 seconds*. When you start/restart the server with this modification, reload the page at http://localhost:8080. Then, rather than *reload* the page again, place your cursor into the address bar and hit Enter, or click the logo at the top of the page that links to index.html. Look at the request for jquery.min.js in the network request listing and you'll see something similar to figure 10.10.

Name	Method	Status	Size
jquery.min.js	GET	200	(from cache)

Figure 10.10 A copy of jQuery being retrieved from the local browser cache

When you *navigate* to a page as opposed to *reloading* it (such as when you click the reload icon), the value of

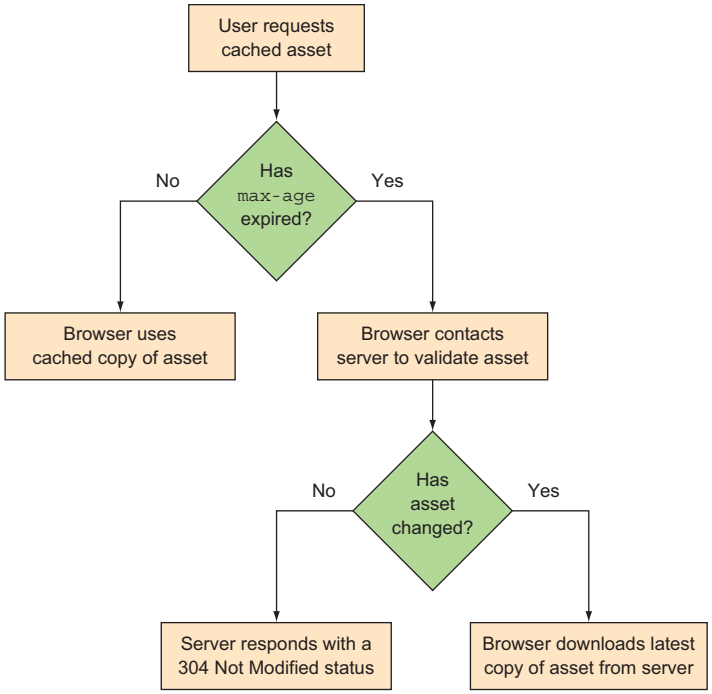

Figure 10.11 The effect of the `Cache-Control` header's `max-age` directive and the browser/server interaction that results in its use

the `Cache-Control` header's `max-age` directive influences whether the browser grabs something from the local cache. If the item is present in the local cache, a request is never made to the server for that item.

If you reload, or the time specified in the `max-age` directive elapses, the browser contacts the server to revalidate the cached asset. When the browser does this, it checks whether the asset has changed. If it has, a new copy of the asset is downloaded. If not, the server responds with a 304 Not Modified status without sending the asset. Figure 10.11 illustrates this process.

The way the server checks to see whether an asset has changed can vary. A popular method uses an *entity tag*, or *ETag* for short. This is a checksum generated from the contents of the file. The browser sends this value to the server, which validates it to see whether the asset has changed. Another method checks the time the file was last modified on the server, and serves a copy of the asset based on its last modification time. You can modify this behavior with the `Cache-Control` header, so let's cover those options briefly.

CONTROLLING ASSET REVALIDATION WITH NO-CACHE, NO-STORE, AND STALE-WHILE-REVALIDATE

The `max-age` directive is fine for most websites, but at times you'll need to put limits on caching behavior or abolish it altogether. For example, you might have applications with data that needs to be as fresh as possible, such as online banking or stock market sites. Three `Cache-Control` directives are at your disposal to help limit caching behavior:

- `no-cache`—This says to the browser that any asset downloaded can be stored locally, but that the browser must always revalidate the resource with the server.
- `no-store`—This directive goes one further than `no-cache`. `no-store` indicates that the browser shouldn't store the affected asset. This requires the browser to download any affected asset *every time* you visit a page.
- `stale-while-revalidate`—Like `max-age`, `stale-while-revalidate` accepts a time measured in seconds. The difference is that when an asset's `max-age` has been exceeded and becomes stale, this header defines a grace period during which the browser is allowed to use the stale resource in the cache. The browser *should* then fetch a new copy of the stale asset in the background and place it into the cache for the next visit. This behavior *isn't* guaranteed, but it can boost cache performance in limited scenarios.

Obviously, these directives affect or remove the performance benefits that caching provides, but at times you need to ensure that assets are never stored or cached. Use these directives sparingly and with good cause.

Cache-Control and CDNs

You might use a CDN in front of your site. A *CDN* is a proxy service that sits in front of your site and optimizes the delivery of your content to your users. Figure 10.12 illustrates the basic CDN concept.

A CDN has the power to distribute your content across the globe. Your site assets and content can be served from computers that are closer to your users than if you served content solely from your own host. The shorter distance can result in lower latency for those assets, which in turn boosts performance.

To accomplish this, the CDN hosts your assets on their network of servers, so your content is effectively cached by the CDN. You can use two `Cache-Control` directives in combination with `max-age`—`public` and `private`—and they can help you control the way that your content is cached by CDNs.

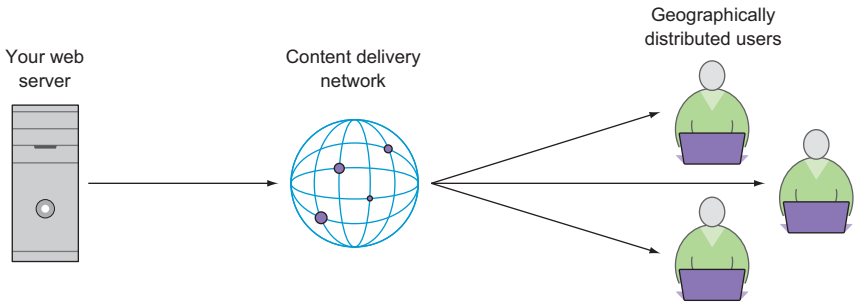

Figure 10.12 The basic concept of a CDN. A CDN is a proxy that sits in front of your website and distributes your content to users across the world. The CDN can do this through a network of geographically distributed servers that host your content. Users have their content requests fulfilled by servers that are closest to them.

Using a `Cache-Control` directive of `public` in conjunction with `max-age` can be done like so:

```
Cache-Control: public, max-age=86400
```

This instructs any intermediary (such as a CDN) to cache the resource on its server. You generally shouldn't need to specify `public` if you're using `Cache-Control`, because it's implied.

The `private` directive is used in the same syntax as `public`, but instructs any intermediary to *not* cache the resource. Using this header treats the resource as if the CDN isn't in play at all. This directive passes the asset through to the user. The user's browser still caches the resource according to the header's `max-age` value, but only with respect to the origin web server behind the CDN, and not to the CDN itself.

From here, you'll take all of this knowledge of the `Cache-Control` header and learn how to create a caching strategy that makes sense for your website.

10.2.2 Crafting an optimal caching strategy

Now that you have all of this knowledge about the `Cache-Control` header, how do you apply it to your website? As with any new piece of information, you'll apply it to something practical, such as the Weekly Timber website you downloaded earlier in this chapter. You'll begin by categorizing assets, choosing a good `max-age` policy for each category as well as relevant directives that make sense. Then, you'll apply this policy in your web server.

CATEGORIZING ASSETS

When categorizing assets, the best criteria to use is how often an asset is likely to change. HTML documents are likely to change often, for example, and assets such as CSS, JavaScript, and images are somewhat less likely to change.

The client's website has basic caching requirements, which makes it a great introduction to using `Cache-Control`. The asset categorization for this website is simple: HTML, CSS, JavaScript, and images. The fonts are loaded via Google Fonts, so caching is handled by Google's servers, leaving you with just the basics to consider. Table 10.1 is a breakdown of these asset types and the caching policy I've selected for them.

Table 10.1 Asset types for the Weekly Timber website, their modification frequencies, and the `Cache-Control` header value that should be u

Asset type	Frequency of modification	Cache-Control header value
HTML	Potentially frequently, but needs to be as fresh as possible	`private, no-cache, max-age=3600`
CSS and JavaScript	Potentially monthly	`public, max-age=2592000, stale-while-revalidate=86400`
Images	Almost never	`public, max-age=31536000, stale-while-revalidate=86400`

The rationale behind these choices is nuanced, but easy to understand when you break it down by asset:

- *HTML files* or server-side languages that output HTML (for example, PHP or ASP.NET) can benefit from a conservative caching policy. You never want the browser to assume that the page should be read only from the browser cache without ever revalidating its freshness.
  - `no-cache` ensures that the resource will *always* be revalidated, and if it has changed, a new copy will be downloaded. The revalidation of the asset *does* lessen the load on the server if the content of the file hasn't changed, but `no-cache` never caches the HTML so aggressively that content is stale.
  - A `max-age` of one hour ensures that no matter what, a new copy of the asset will be fetched after the `max-age` period expires.
  - Using the `private` directive tells any CDN in front of the origin web server that this resource shouldn't be cached on their server(s) at all, only between the user and origin web server.
- *CSS and JavaScript* are important resources, but don't need to be so aggressively revalidated. Therefore, you can use a `max-age` of 30 days.
  - Because you'd benefit from a CDN distributing this content for you, you should use the `public` directive to allow CDNs to cache the asset. If you need to invalidate a cached script or style sheet, you can do so easily. That process is explained in the next section.
- *Images* and other media files such as fonts rarely (if ever) change and are often the largest assets you'll serve. Therefore, a long `max-age` time (such as a year) is appropriate.
  - As with CSS and JavaScript files, you want CDNs to be able to cache this asset for you. Using the `public` directive makes sense here as well.
  - Because these assets don't change often, you want to have a grace period in which a certain amount of staleness is okay. Therefore, a `stale-while-revalidate` period of one day is appropriate while the browser asynchronously validates the freshness of the resource.

The caching strategy that's best for your website may vary. You may decide that no caching should ever occur whatsoever with your HTML files, and this isn't necessarily a bad approach if your site constantly updates its HTML content. In this case, it may be preferable for you to use the `no-store` directive, which is the most aggressive measure that assumes you never want to cache anything or bother with revalidation.

You may also decide that your CSS and JavaScript should have a long expiration time, as for images. This is also fine, but unless you invalidate your caches, this could result in asset staleness when you make updates. You might go the other way and decide that the browser should revalidate a cached asset's freshness with the server on every request. In any case, the next section covers how to properly invalidate your cached assets. For now, though, you need to implement your caching strategy on your Node web server!

IMPLEMENTING THE CACHING STRATEGY

Putting your caching strategy into effect on the local web server is simple. You add a request handler that allows you to set response headers before assets are sent to the client. In this section, you'll open http.js and use the `mime` module to inspect the types of the assets requested, and set a `Cache-Control` header based on their type. If you want to skip ahead, you can do so by entering `git checkout -f cache-control` in your terminal window. Otherwise, the following listing shows the changed portions of http.js in bold, with annotations.

Listing 10.4 Setting `Cache-Control` headers by file type

```
var express = require("express"),
    compression = require("compression"),
    path = require("path"),
    mime = require("mime"),
    app = express(),
    pubDir = "./htdocs";

// Run static server
app.use(compression());
app.use(express.static(path.join(__dirname, pubDir), {
    setHeaders: function(res, path){
        var fileType = mime.lookup(path);

        switch(fileType){
            case "text/html":
                res.setHeader("Cache-Control",
                              "private, no-cache, max-age=" + (60*60));
                break;

            case "text/javascript":
            case "application/javascript":
            case "text/css":
                res.setHeader("Cache-Control",
                              "public, max-age=" + (60*60*24*30));
                break;

            case "image/png":
            case "image/jpeg":
            case "image/svg+xml":
                res.setHeader("Cache-Control",
                              "public, max-age=" + (60*60*24*365));
                break;
        }
    }
}));
app.listen(8080);
```

Imports the mime module that you'll use to look up file types for requested assets.

setHeaders callback lets you specify behavior that will run just before the response is sent.

File type of the requested asset is determined by the mime module.

A switch statement is run on the value of the fileType variable.

HTML files set Cache-Control: private, no-cache, max-age=3600.

JavaScript and CSS files set Cache-Control: public, max-age=2592000.

PNG, JPEG, and SVG images set Cache-Control: public, max-age=31536000.

When you're finished, start or restart the server. Testing cache policy changes can be tricky, but here's a four-step process you can use in Chrome to see how things are working:

1. Open a new tab and open the Network panel. Make sure the Disable Cache check box is selected so that you get a fresh copy of the page.
2. Navigate to a web page that you want to test (http://localhost:8080, in this case).
3. When the loading has finished, uncheck the Disable Cache box.
4. Don't *reload* the page, as this will cause the browser to contact the server to revalidate assets. Instead, navigate to the page. To do this on a page you're already on, click in the address bar and hit Enter.

When you do this, you can then see the effects of your `Cache-Control` headers at work. Figure 10.13 shows a partial listing of assets in the Network panel.

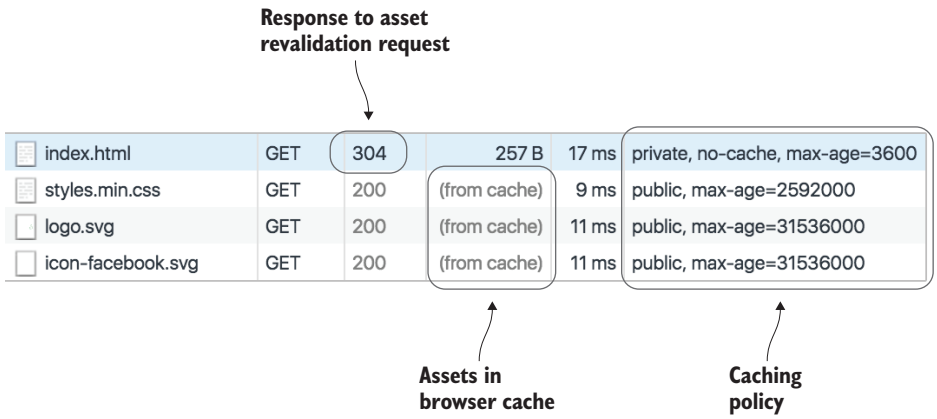

Figure 10.13 The effects of your cache policy on the Weekly Timber website. The HTML is revalidated from the server on every request, and the server returns a 304 status if the document hasn't changed on the server. Items reading from the browser cache don't trigger a return trip to the web server.

This caching strategy is optimal for our purposes. When the cache is primed, only one request is made to validate the HTML file's freshness with the server on a return visit. If the locally cached document is still fresh, the total page weight is less than half a kilobyte. This makes subsequent visits to this page fast, and the assets shared on all pages are cached on subsequent pages, decreasing the load time of those pages as well.

Of course, at times you'll update your website and need to invalidate assets in your browser cache. In the next section, I explain how to do exactly that.

10.2.3 Invalidating cached assets

You've been in a scenario like this: You've worked hard on a project for the folks at Weekly Timber for weeks and finally deployed the site to production, only to find out hours later that there's a bug. This bug might be in your CSS or JavaScript, or perhaps a content problem in an image or your HTML. The bugs have been fixed and deployed to production, but the site still isn't updating for your users because their browser cache is preventing them from seeing your changes.

Although advice such as "reload the page" or "clear your cache" may pacify nervous marketers and business clients, this isn't how users normally interact with web pages. It may not occur to users to reload a page in typical circumstances. You need to find a way to force the page's assets to be downloaded again.

Although possibly tedious, this is an easy problem to overcome. If you're using the caching strategy outlined in the previous section, your browser will always validate the freshness of the HTML with the server. As long as this occurs, you have a good shot at getting updated assets out to users who have outdated ones in their browser cache.

INVALIDATING CSS AND JAVASCRIPT ASSETS

Just your luck: The Weekly Timber website has a CSS or JavaScript bug in production that made it past QA somehow. With the bug identified and the fix deployed, you're able to verify that it's in production because you reloaded the page, but your project manager is insistent that users aren't seeing see the updated content. The concern is warranted, and you need to do something.

The fix here is simple. Remember that with your current caching policy in place, the browser always validates the HTML document's freshness with the server. You can make a small change in the HTML that will not only trigger it to download again, but also trigger the modified assets to download again. All it requires is adding a query string to a CSS or JavaScript reference. If you need to force the CSS to update, you can update the `<link>` tag reference to the CSS to something like this (change bolded):

```
<link rel="stylesheet" href="css/styles.min.css?v=2" type="text/css">
```

Adding a query string to the asset causes the browser to download the asset again because the URL of the asset has changed. After this change in the HTML is uploaded to the server, site visitors with the old version of styles.min.css in their cache will now receive the new version of styles.min.css. This same method can be used to invalidate any asset, including JavaScript and images.

This can come off as a hacky way of solving the problem. It's a fine stopgap when you need to make sure something won't be cached, but you don't want to have to be responsible for versioning your own files, either. A more convenient way around this is to use a server-side language such as PHP to handle this problem for you automatically whenever you update a file. The following listing shows one way of handling this problem.

Listing 10.5 Automated cache invalidation in PHP

Creates an MD5 hash of styles.min.css. This is unique based on the file's contents.

```
<?php $cssVersion = md5_file("css/styles.min.css"); ?>
<link rel="stylesheet" href="css/styles.min.css?v=<?php echo($cssVersion); ?>"
➥ type="text/css">
```

Appends the hash string to the query string.

This solution works well because the `file_md5` function generates an MD5 hash based on the contents of the file. If the file never changes, the hash stays the same. If just one byte of the file changes, however, the hash changes.

You can accomplish this in other ways, of course. You can use the language's `filemtime` function to check for the file's last modification time and use that instead. Or you can write your own versioning system. The point is that this is illustrative of a concept: no matter what language you're working with, tools are available that can automate this piece for you.

Invalidating images and other media files

Sometimes the problem isn't with your CSS or JavaScript; it's with media files such as images. You could use the query string method as explained earlier, but the sensible option may be to point to a new image file.

If you're running a small website such as Weekly Timber's, it doesn't make much of a difference whether you use the query string trick or not. But if your site uses a content management system (CMS), pointing to an entirely new file is the easiest way to avoid caching issues altogether. Upload a new image, and the CMS will point to it. The new image URL, never having been previously cached by users before, will be reflected in the HTML and be immediately visible to your users.

With our adventures in caching coming to a close, you'll now venture into the territory of CDN-hosted assets, and how they can help your website's performance.

10.3 Using CDN assets

The preceding section briefly touched on CDNs and their effect on caching. But we didn't dive into how a CDN can help your website's performance. This section covers CDNs that host commonly used JavaScript and CSS libraries, and the benefits they can provide. It then goes on to explain how to fall back to a locally hosted copy of a library if a CDN fails, as well as how to verify the authenticity of the assets you're referencing with Subresource Integrity.

10.3.1 Using CDN-hosted assets

CDNs can provide a performance boost by distributing assets such as JavaScript and CSS files across the world, and serving them to users based on their proximity. These assets are hosted on an origin server, and then distributed to servers that are closest to potential end users. These servers are called *edge servers*. Figure 10.14 illustrates this concept.

Although CDNs are comprehensive services that you can place in front of your server to serve and cache your content for you, the cost of these services range anywhere from free to expensive. Moreover, documenting their ever-changing features and offerings would be a fool's errand in a printed book such as this. The scope of this section is to cover the benefits of CDNs that host common libraries that you can link to in order to avoid serving this content from your own server. These services are offered freely and can improve the performance of your website with little effort.

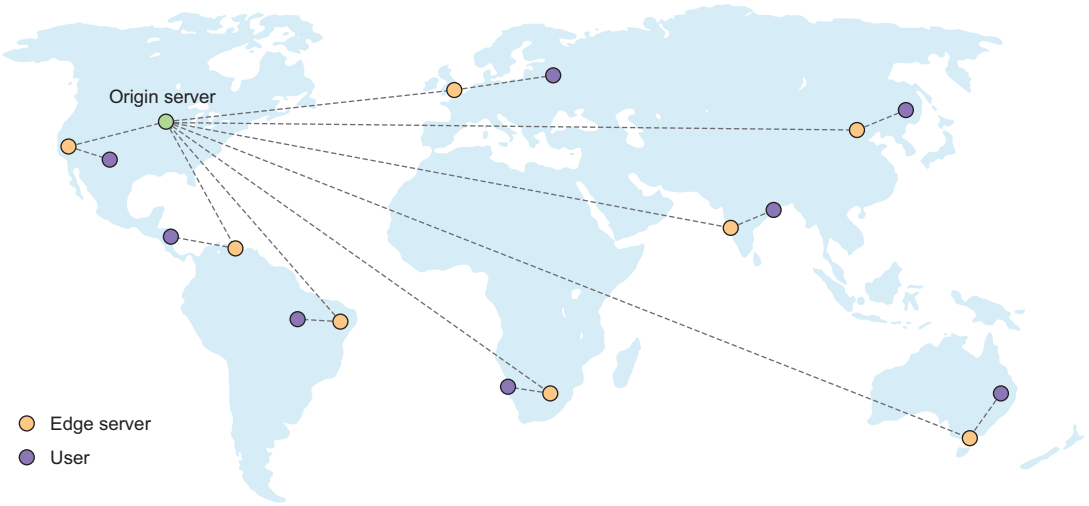

Figure 10.14 In a CDN, assets hosted on an origin server are distributed to edge servers, which are servers that are located closer to potential website visitors.

REFERENCING A CDN ASSET

Using a CDN-hosted asset is as simple as it gets. A good example of a CDN-hosted asset is jQuery. The developers of jQuery offer a CDN-hosted version of this asset through MaxCDN, a fast CDN service. The Weekly Timber website uses a local copy of jQuery v2.2.3. Open index.html in the website's root folder and locate the line where jQuery is included:

```
<script src="js/jquery.min.js"></script>
```

You can change this `<script>` tag's `src` attribute to point to a CDN-hosted version of the library provided for free by MaxCDN:

```
<script src="https://code.jquery.com/jquery-2.2.3.min.js"></script>
```

Okay, so now what? How is this helping you? I could ramble on for a whole paragraph about how the CDN transfers the asset faster, and how much lower the latency is than serving this from the shared host that Weekly Timber is on at http://weeklytimber.com, but I'll let the graph in figure 10.15 do the talking instead.

Any CDN is more capable than the low-cost shared host that Weekly Timber is hosted on in both TTFB and total load time. Two caveats on this test: It was done from the Upper Midwest on a 1 gigabit fiber connection, with the shared host operating on the West Coast. Don't accept this as a comprehensive evaluation of CDN speed. Always do your own testing. Even with these caveats, however, the benefits are clear. You're likely to realize *some* benefit unless you have incredible infrastructure behind your website. Still, even enterprise applications use CDN-hosted resources because of the speed and convenience they provide.

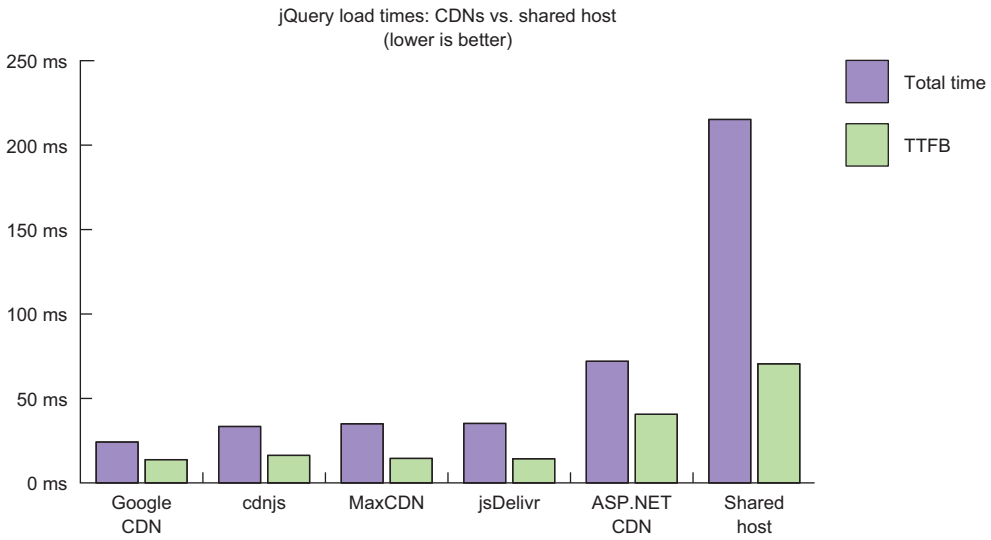

Figure 10.15 A comparison of load times and TTFB for jQuery over several CDNs versus a low-cost shared hosting environment

The benefits aren't limited to speed. CDNs manage asset caching for you, leaving you with one less thing to worry about. The CDN will invalidate assets when it needs to, saving you the hassle of having to update your code. Moreover, if a CDN asset such as jQuery is widely used, there's a good chance that if a user visits a site prior to visiting yours that uses the same asset, it will already be in that user's cache. This translates into a lower overall load time for your page. Performance for free!

It isn't just for jQuery

You're not using just jQuery, and heck, maybe you're not even using jQuery at all. Many CDNs are out there that host a variety of resources for you, not just one popular library. Here's a short list of CDNs that host all kinds of goodies for you:

- *cdnjs* (https://cdnjs.com) is a CDN that hosts almost any popular (or not so popular) library in existence. It provides a clean interface that enables you to search for any commonly used CSS or JavaScript asset you can think of, such as widely used MVC/MVVM frameworks, jQuery plugins, or anything else your project depends on.
- *jsDelivr* (http://jsdelivr.com) is another CDN similar to cdnjs. Try searching here if cdnjs doesn't provide what you're looking for.
- *Google CDN* (https://developers.google.com/speed/libraries) is much less comprehensive than cdnjs or jsDelivr, but it does provide popular libraries such as Angular and others. In my testing, this was the fastest CDN.
- *ASP.NET CDN* (http://www.asp.net/ajax/cdn) is Microsoft's CDN. It's less comprehensive than cdnjs or jsDelivr, but slightly more so than Google. In my testing, this was the slowest option, but was still three to four times faster than my shared hosting, making it a viable option.

A piece of advice if you're going to go whole hog referencing CDNs for all of your common libraries: use as few distinct CDNs as possible. Every new CDN host you point to will incur another DNS lookup, which can increase latency. If all of your assets can be found on one CDN, use that one. Don't point to three or four hosts if one will do the job.

Another piece of advice: If you're using a library such as Modernizr or Bootstrap that can be configured to deliver a specific part of that library's functionality, configure your own build instead of pointing to the entire library on the CDN. Sometimes it's faster to configure a smaller build and host it on your own server than it is to reference the full build from a CDN. Figure out your needs, and do your homework on which method performs better.

Next, we'll dive into what happens when a CDN fails, and how to work around this unlikely (but possible) event.

10.3.2 What to do if a CDN fails

Perhaps the largest criticism I see against CDNs is, "What happens if the CDN fails?" Although the smug among us are quick to dismiss this scenario as unlikely (and I've been guilty of this in the past myself), the truth is that it does happen. Like any service, CDNs can't practically guarantee 100% uptime. They're available the vast majority of the time, but service interruptions can and do happen.

More likely than service interruptions, however, are networks that are configured to block particular hosts. These networks can be security-conscious corporations, public facilities, military organizations, or even governments that block entire domains as part of internet censorship efforts. Having a backup plan to address these situations makes sense.

You can fall back to a local copy of an asset by using a simple JavaScript function. The following listing shows a fallback loader function that you can place in index.html of the Weekly Timber site in order to provide a fallback copy when a CDN-hosted library fails to load.

Listing 10.6 A reusable fallback script loader

```
<script>
    function fallback(missingObj, fallbackUrl){
        if(typeof(missingObj) === "undefined"){
            var fallbackScript = document.createElement("script");
            fallbackScript.src = fallbackUrl;
            document.body.appendChild(fallbackScript);
        }
    }
</script>
<script src="https://code.jquery.com/jquery-2.2.3.min.js"></script>
<script>
    fallback(window.jQuery, "js/jquery.min.js");
</script>
```

Checks for the presence of the target library's object by checking whether it's undefined.

If the object tested for is undefined, create a new fallback <script> element.

Set the fallback script's location to another URL.

Load the asset by appending the fallback <script> element to the end of the body.

CDN reference to jQuery

The fallback script specifying the object to test for and the relative URL to the fallback script

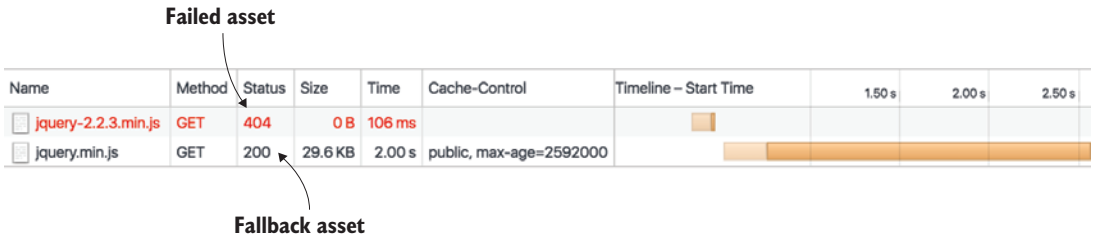

Figure 10.16 The Network panel in Chrome showing the CDN asset failing to load and the page falling back to the locally hosted version

To test this, you can disable your network connection so that the page can't access the CDN asset, or change the URL to something bogus. When you do this, reload the page and check your network panel and you'll see the locally hosted asset being loaded as the fallback, as shown in figure 10.16.

This works because of the nature of the `<script>` tag. Multiple `<script>` tags parse and execute in the order they're defined in the markup, each waiting for the one before it to finish. The `<script>` tag that attempts to load the fallback won't run until the one before it that references the CDN version of jQuery fails to load.

Of course, this isn't limited to jQuery. To conditionally load fallbacks, test for the JavaScript library's global object. For example, if you need to fall back to a locally hosted version of Modernizr, you can use the `fallback` function like so:

```
fallback(window.Modernizr, "js/modernizr.min.js");
```

Next, I'll cover how to verify the integrity of CDN assets via Subresource Integrity, so you know that the assets you request are what you expect them to be.

10.3.3 Verifying CDN assets with Subresource Integrity

When you've downloaded software from the web, you've likely seen a checksum string near the download link. *Checksums* are a sort of signature that helps you ensure that the file you've downloaded is what the publisher of the program has intended for you to run. This is done for your own safety so you don't unwittingly run malicious code. If the checksum of the file doesn't match what the publisher has given you, the file isn't safe to use.

This same kind of integrity check can now be done in HTML in some browsers to make sure that an asset included via a `<script>` or `<link>` element from a CDN is what the publisher has intended for you to use. This process, called *Subresource Integrity*, is illustrated in figure 10.17.

Although this feature doesn't impact the performance of your website, it does provide a safeguard against tampered assets for your users, and bears mentioning in the context of using CDN assets.

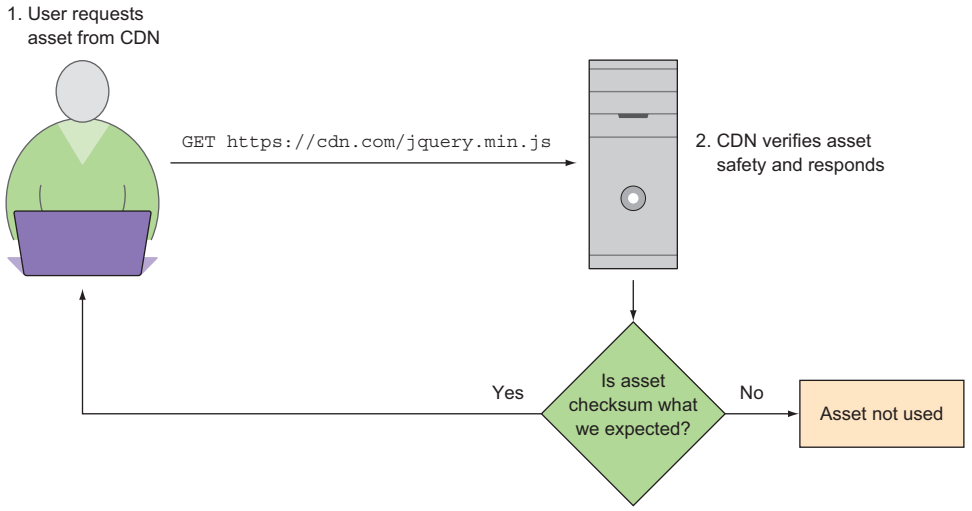

Figure 10.17 The process of verifying assets by using Subresource Integrity. A user requests an asset from a CDN, and the asset's safety is determined via a checksum verification process. If the asset is safe, it's used. If not, the asset is discarded.

USING SUBRESOURCE INTEGRITY

The syntax for Subresource Integrity uses two attributes on either a `<script>` or `<link>` tag referencing a resource on another domain. The `integrity` attribute specifies two things: the hash algorithm used to generate the expected checksum (for example, MD5 or SHA-256), and the checksum value itself. Figure 10.18 shows the format of this attribute value.

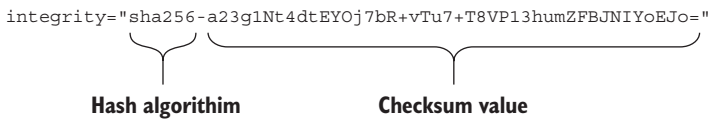

Figure 10.18 The format of the `integrity` attribute. This value starts off with the hashing algorithm (SHA-256, in this case) and is followed by the checksum value for the referenced resource.

The second attribute is `crossorigin`, which always has a value of `anonymous` for CDN assets to indicate that the resource doesn't require any user credentials in order to be accessed. When combined and used on a `<script>` tag for version 2.2.3 of jquery.min.js, it looks something like this:

```
<script src="https://code.jquery.com/jquery-2.2.3.min.js"
        integrity="sha256-a23g1Nt4dtEYOj7bR+vTu7+T8VP13humZFBJNIYoEJo="
        crossorgin="anonymous">
</script>
```

In compliant browsers, everything will work when the checksum of the CDN asset matches what the browser expects. If the checksum match fails, you'll see an error message in your console that alerts you that the affected asset fails the integrity check. When this happens, the asset won't be loaded. If you specify a fallback mechanism shown in the previous section, however, you'll be covered and a local copy can be loaded.

This verification method isn't supported in all browsers, but those that *do* support it, such as Firefox, Chrome, and Opera, are widely used. Browsers that don't support Subresource Integrity will ignore the `integrity` and `crossorigin` attributes and load the referenced assets.

GENERATING YOUR OWN CHECKSUMS

Some CDNs provide code snippets that have Subresource Integrity already set up for you, but this isn't a standard practice yet. You might have to generate your own checksums. The easiest way to do this is to use the checksum generator at https://srihash.org, but if you prefer to generate your own checksums, you can rely on the `openssl` command-line utility. To generate a SHA-256 checksum for a file, use this syntax at the command line:

```
openssl dgst -sha256 -binary yourfile.js | openssl base64 -A
```

This generates a checksum for the file you provide and outputs it to the screen. If you're running Windows, you can download an OpenSSL binary for Windows or use the `certutil` command. Your best bet in both instances is to use the online tool, as it's more convenient and generates the same output. If you do decide to generate your own checksums, use a reliable hash algorithm such as SHA-256 or SHA-384. Algorithms such as MD5 or SHA-1 aren't secure enough for today's needs. If you have any doubts about what you're doing, allow the online tool at https://srihash.org to do the work for you. It's less of a hassle anyway, right?

In the final section of this chapter, you'll learn how to fine-tune the delivery of resources on your website with resource hints, which can be used in HTML or in HTTP headers.

10.4 Using resource hints

As HTML has matured, features have been added to the language that assist in the delivery of assets to the user. These features are called *resource hints*, which are a collection of behaviors driven by the HTML `<link>` tag or the `Link` HTTP response header that perform tasks such as DNS prefetching and preconnecting to other hosts, prefetching and preloading resources, and prerendering pages. This section covers all of these techniques and the pitfalls that can be associated with their use.

10.4.1 Using the preconnect resource hint

As I said in chapter 1, one component of poor application performance can be due to latency. One way to limit the effects of latency is through the `preconnect` resource hint, which connects to a domain that hosts an asset that the browser hasn't started downloading.

This doesn't provide a benefit when used to point to the host that the current page is being accessed from. Doing this doesn't improve performance, because the DNS lookup to the requested document would've already occurred by the time the document loads and discovers the `preconnect` resource hint.

`preconnect` works best when you're referencing assets on different domains (such as CDNs). Because HTML documents are read from top to bottom by browsers, a connection is established to an asset's domain when the reference to the asset is discovered by the browser. If this asset reference is in a `<script>` tag, say, in the footer, placing a `preconnect` resource hint in the header can give the browser a head start on connecting to the domain that hosts the resource.

The Weekly Timber website has a reference to the jQuery library hosted on code.jquery.com. You can use the `<link>` tag to establish an early connection to the domain the resource is hosted on:

```
<link rel="preconnect" href="https://code.jquery.com">
```

Alternately, you can configure the web server to send a `Link` response header along with an HTML document:

```
Link: <https://code.jquery.com>; rel=preconnect
```

Both methods accomplish the same task, but the level of effort varies. Using a `<link>` tag in HTML involves little effort. Adding a `Link` response header is more involved, but the resource hint will be discovered sooner than if it were in the document. I tested both methods in index.html to instruct the browser to establish a connection to code.jquery.com as soon as possible. Figure 10.19 shows the impact of this test on loading jQuery from the CDN.

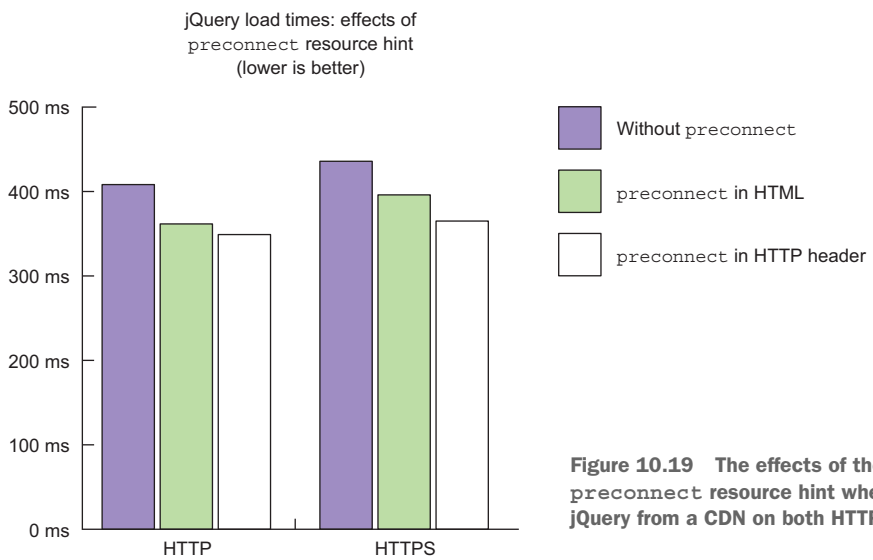

Figure 10.19 The effects of the `preconnect` resource hint when loading jQuery from a CDN on both HTTP and HTTPS

This technique has the potential to improve your website's performance, but as always, perform your own testing to determine the benefit to your specific situation. Although well supported in Firefox and Chromium browsers (Chrome, Opera, and Android browsers), `preconnect` isn't supported everywhere, so not all of your users will realize its benefits.

> **The dns-prefetch resource hint**
>
> A less effective but more widely supported resource hint is `dns-prefetch`. It's used in the same way, except instead of using the `preconnect` keyword in your `rel` attribute or `Link` header, you replace it with `dns-prefetch`. This resource hint doesn't perform a full connection to the specified domain, but rather a DNS lookup to resolve the domain's IP address. I didn't realize any benefits when using this header in my own testing, but it could provide some benefits in situations where latency is a serious problem.

Next, we'll discuss the performance enhancements of the `prefetch` and `preload` resource hints, and how to use them to more quickly load assets on your website.

10.4.2 Using the prefetch and preload resource hints

Two resource hints exist for downloading specific assets: `prefetch` and `preload`. Both are similar in what they do but have distinct differences. We'll start by covering `prefetch`.

Using the prefetch resource hint

In capable browsers, `prefetch` tells the browser to download a specific asset and store it in the browser cache. This resource hint can be used to prefetch resources on the same page as the request, or you can make an intelligent guess as to what pages the user may visit next and request assets from that page. Be especially careful with the second approach, as it could force the user to unnecessarily download an asset. The syntax for `prefetch` is just like that of `preconnect`; the only difference is the value in the `rel` attribute of the `<link>` tag:

```
<link rel="prefetch" href="https://code.jquery.com/jquery-2.2.3.min.js">
```

It can also be specified in an HTTP header much the same way as `preconnect`:

```
Link: <https://code.jquery.com/jquery-2.2.3.min.js>; rel=prefetch
```

For example, you can improve the load time of the Weekly Timber home page by including this resource hint in index.html for jQuery as shown previously. When you do this, you can cut the load time of the page by nearly 20%, as shown in figure 10.20.

As you can see, there's a noticeable benefit in using `prefetch` in this scenario. Because the `<script>` element that includes jQuery is at the bottom of the page, it's not discovered and downloaded until the page is nearly done parsing the HTML. By adding

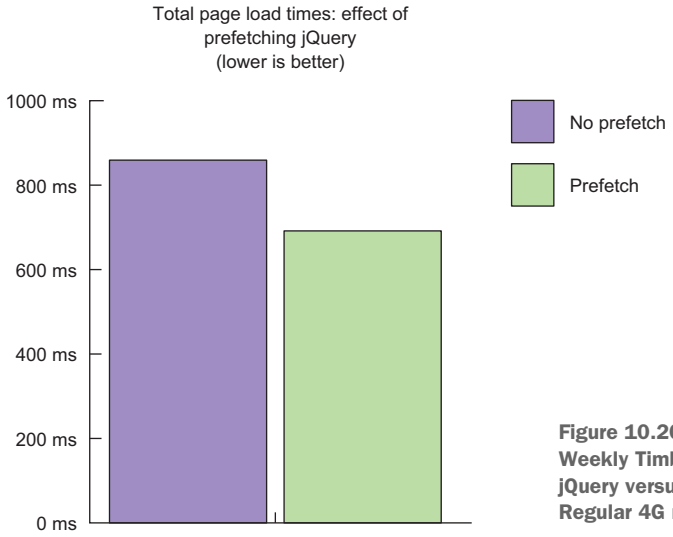

Figure 10.20 Page load times for the Weekly Timber home page when prefetching jQuery versus no prefetching using Chrome's Regular 4G network throttling profile

a `prefetch` hint in the HTML's `<head>`, the browser gets a head start on downloading the file. By the time the reference to jQuery is found, the `prefetch` hint has already grabbed it and stored it in the browser cache, reducing the load time of the site.

prefetch testing tip

Testing `prefetch` can be tricky. If you use Chrome and have the Disable Cache check box selected in the Network panel, it can appear that `prefetch` results in a performance penalty because a prefetched asset will download twice. To see and measure the benefit, you need to clear your cache, re-enable caching, and monitor performance with an unprimed cache.

There are limitations to `prefetch`, and the browser makes no guarantee that it'll prefetch the resource as specified. Each browser has its own rules about `prefetch`, so be aware that the browser may not always honor the resource hint. This feature is well supported, but like any unsupported HTML feature, browsers that don't understand it will ignore it. This ensures normal behavior for noncompliant browsers.

USING THE PRELOAD RESOURCE HINT

The `preload` resource hint is much like `prefetch`, except that it guarantees that the specified resource *will* be downloaded. It's like `prefetch`, but without the ambiguity. Unlike `prefetch`, though, `preload` enjoys less browser support, with only Chromium-based browsers supporting the feature at the time of this writing.

Predictably, `preload` is used in the same fashion as prior resource hints:

```
<link rel="preload" href="https://code.jquery.com/jquery-2.2.3.min.js"
     as="script">
```

It's also used in a similar fashion when in an HTTP header:

```
Link: <https://code.jquery.com/jquery-2.2.3.min.js>; rel=preload; as=script
```

The major difference here with `preload` is that you can use the `as` attribute to describe the type of content being requested. Values of `script`, `style`, `font`, and `image` can be used for JavaScript, CSS, fonts, and images, respectively. This attribute is entirely optional, and causes no ill effects when omitted.

A quick side note: an HTTP/2 feature called *Server Push* uses the same HTTP header syntax to preemptively "push" a resource to the user along with the HTML document response. This functionality and its performance benefits are covered in depth in chapter 11.

In the preceding example, you use `preload` to grab the CDN copy of jQuery much as you did in the preceding section with `prefetch`. The performance advantages are largely the same as they are with `prefetch`, except that you can count on compliant browsers to honor your request with `preload`. Figure 10.21 shows the `preload` hint at work in Chrome's Network panel.

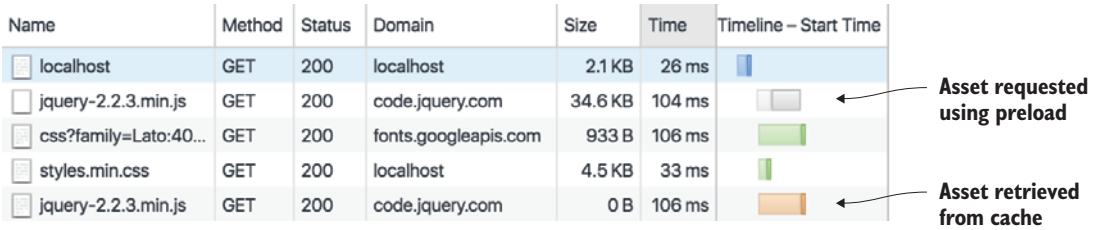

Figure 10.21 The Network panel showing jquery-2.2.3.min.js loaded with the `preload` resource hint. The first line for the jQuery library is from the `preload` hint, whereas the second occurs when the item is retrieved from the cache. Note the size of 0 bytes on the second entry for jquery-2.2.3.min.js.

As with any resource hint, you should always test the performance of your page before and after its addition. If you need the broadest possible browser support and have to choose between `preload` and `prefetch`, choose `prefetch`. If browser support isn't as important, and you want your request to preemptively load content to be honored no matter what, choose `preload`.

To wrap up this section, you'll investigate the `prerender` resource hint, and how it can be used to render an entire page before the user even navigates to it.

10.4.3 Using the prerender resource hint

The last resource hint we'll cover here is `prerender`, and it's the most powerful of all. Rather than point to a specific asset such as a JavaScript file, style sheet, or image, the target of the `prerender` hint must be a URL to a web page. When used, this hint suggests that the browser download *and* render the *entire* document specified. If the user clicks through to the prerendered document, the page will appear instantaneously.

This can occur when the browser discovers the resource hint in either an HTML `<link>` tag or in a `Link` HTTP header.

Clearly, you need to be conservative in using this feature. If you're not sure whether you should use `prerender`, I have one piece of advice for you: don't. An improperly used `prerender` hint has high potential to saddle the user with the unnecessary downloading of data. For mobile users on restricted data plans, this is tantamount to abuse.

But in some limited instances, using `prerender` makes sense. For example, let's say your organization is sponsoring a promotion, and you know that a vast majority of your visitors are going to click through to a specific page to sign up. Or perhaps you have multipage content such as articles that you want to load in advance for the user. Intranet applications are possibly the best use of this feature, because bandwidth used on a local network isn't as consequential to users as bandwidth used on the internet at large.

These situations aren't always candidates for `prerender`, however. Do your homework. Look at your analytics and see how your users are behaving, and accept that *any* use of `prerender` on a sufficiently large audience will result in wasted bandwidth for some portion of your users.

Using `prerender` is just like using any other resource hint. The following is an example of this hint used on index.html of the Weekly Timber website to prerender the contact-us.html document:

```
<link rel="prerender" href="http://localhost:8080/contact-us.html">
```

This can also be specified by using the `Link` HTTP header:

```
Link: <http://localhost:8080/contact-us.html>; rel=prerender
```

If you employ this technique, remember that because a `prerender` hint can kick off an expensive operation, any browser that supports it reserves the right to ignore it. When used judiciously, `prerender` can provide the feeling that a navigation event is occurring instantaneously, so it's worth your consideration in some limited applications.

With your journey through the world of resource hints complete, you're ready to wrap up this chapter with a review of what you've learned.

10.5 Summary

Unlike previous chapters, this chapter zigged and zagged through several seemingly unrelated concepts, but they're all focused on the same goal: helping you to fine-tune the delivery of assets for your website. Let's review what we've covered:

- Poorly configured compression can increase latency for users when you either apply too much compression or compress file types that are already internally compressed.
- Brotli compression is a new algorithm that can provide some benefits over gzip but can also increase latency at its highest settings. The future of this compression algorithm looks promising, and currently enjoys decent browser support.

- Configuring caching behaviors for your website through use of the `Cache-Control` header can improve performance for return visitors to your website.
- `Cache-Control` settings can make for stubborn caches. You learned how to invalidate cached assets when you update your website with new content.
- By using assets hosted on CDNs, you can improve overall load times for your website. Sometimes these services fail, however, so it's good form to provide fallbacks to local copies so that your users aren't left in a lurch when the unthinkable happens.
- Using CDN-hosted assets means that you're relinquishing some control of the content you're loading in exchange for better performance, but you don't have to sacrifice your users' safety if you verify the integrity of these assets with Subresource Integrity.
- Resource hints can be used to speed up the loading of web pages, fine-tune the delivery of specific page assets, and prerender pages that the user hasn't even visited yet.

In the next chapter, you'll explore new frontiers by investigating the new HTTP/2 protocol. You'll learn how it can bring further performance improvements to your websites and the effect this protocol will have on your optimization techniques.