Web Performance IN ACTION

BUILDING FAST WEB PAGES

Jeremy L. Wagner

FOREWORD BY Ethan Marcotte

SAMPLE CHAPTER

MANNING

Web Performance in Action

by Jeremy L. Wagner

Chapter 1

Copyright 2017 Manning Publications

brief contents

1 Understanding web performance

This chapter covers

- Why web performance matters
- How web browsers talk to web servers
- How poorly performing websites can be detrimental to the user experience
- How to use basic web optimization techniques

You've probably heard about performance as it relates to websites, but what is it and why should you and I care about it? *Web performance* refers primarily to the speed at which a website loads. This is important because shorter load times improve the user experience for your site on all internet connections. Because this improves the user experience, the user is more likely to see what your website has to offer. This helps you achieve goals as simple as getting more users to visit and read your website's content, or as lofty as getting users to take action. Slow websites test users' patience and might result in them abandoning your website before they ever see what it has to offer.

If your website is a major source of revenue, it literally pays to take stock of your site's performance. If you have an e-commerce site or a content portal that depends on advertising revenue, a slow site affects your bottom line.

In this chapter, you'll learn the importance of web performance, basic performance-boosting techniques, and ways to apply them in order to optimize a client's single-page website.

1.1 Understanding web performance

You may be a developer who has heard of web performance, but you don't know a lot about it. Maybe you've used a few techniques for quick wins, or you may already be well versed in the subject, and picked up this book to discover new techniques you can use to further tune your own websites.

Don't worry! Whether you have little experience in this arena or fancy yourself somewhat of an expert on the subject, the goal of this book is to help you better understand web performance, the methods used to improve the performance of a website, and the ways to apply these methods to your own website.

Before we can talk about the specifics of web performance, however, it's important to understand the problem we're trying to solve.

1.1.1 Web performance and the user experience

High-performing websites improve the user experience. By making sites faster, you improve the user experience by speeding up the delivery of content. Moreover, when your site is faster, users are more likely to care about what's on it. Not one user cares about the content of a site that doesn't load quickly.

Slow websites also have a measurable effect on user engagement. On e-commerce sites in particular, nearly half of users expect a website to load within 2 seconds. And 40% of users will exit if it takes more than 3 seconds to load. A 1-second delay in page response can mean a 7% reduction in users taking action (https://blog.kissmetrics.com/loading-time). This means not only a loss of traffic, but a loss of revenue.

In addition, the performance of your website impacts not only your users, but also your website's position in Google search results. As early as 2010, Google indicated that page speed is a factor in ranking websites in its search results. Though the relevance of your site's content is still the most important factor in your site's search ranking, page speed does play a role.

Let's take the search rankings for Legendary Tones, a relatively popular blog about guitars and guitar accessories that receives about 20,000 unique visitors a month. This site receives much of its traffic from organic search results, and has well-written, relevant content. Using Google Analytics, you can get data on the average speed of all pages and correlate them to their average rankings. Figure 1.1 shows the graphed findings for a month in 2015.

Search rankings remain stable, but when crawl times start straying beyond a second, the ranking slips. It pays to take performance seriously. If you're running a

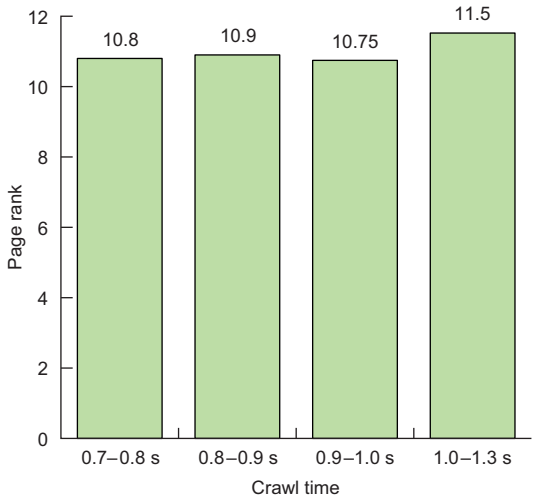

Figure 1.1 The average rankings of all pages on the Legendary Tones website according to its page download time by Google. Lower values are better.

content-driven site such as a blog, your organic search rankings are the greatest source of traffic you have. Reducing your website's load time is one part of a formula for success.

Now that you know why performance is important, we can begin to talk about how web servers communicate and how this process can lend itself to making websites slower.

1.1.2 How web browsers talk to web servers

To know why web optimization is necessary, you need to know where the problem lies, and that's in the basic nature of the way web browsers and web servers communicate. Figure 1.2 illustrates an overview of this concept.

When it's said that web performance focuses on making websites load faster, the primary focus is on reducing load time. The most simple interpretation of *load time* is the time between the instant a user requests a website and the instant it appears on the user's screen. The mechanism driving this is the time it takes for the server's response to reach the user after the user requests content.

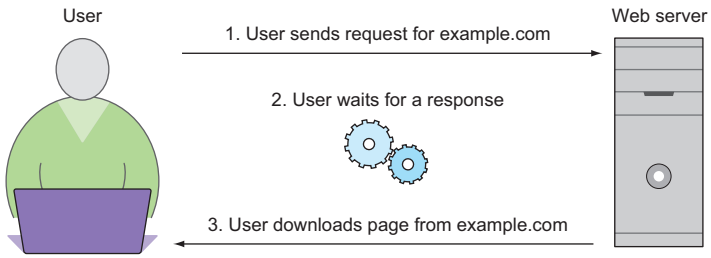

Figure 1.2 A user's request for example.com. The user sends the request for the web page via a browser and then must wait for the server to gather its response and send it. After the server sends the response, the user receives the web page in the browser.

Think of this process as being similar to walking into a coffee shop and asking for a cup of dark roast. After a bit of a wait, you get a cup of coffee. At its most basic level, talking with a web server isn't much different: you request something and eventually receive what you requested.

When a browser fetches a web page, it talks to a server in a language called *Hypertext Transfer Protocol*, commonly known as HTTP. The browser makes an *HTTP request*, and the web server replies with an *HTTP response*, which consists of a status code and the requested content.

In figure 1.3, you see a request being made to example.com (an actual website, believe it or not). The verb `GET` tells the server to locate `/index.html`. Because a few versions of HTTP are in use, the server wants to know which version of the protocol is being referenced (which in this case is HTTP/1.1). In the last step, the request is clarified with the host of the resource.

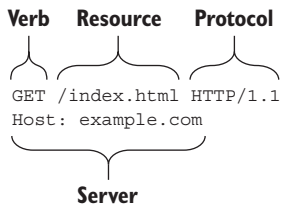

Figure 1.3 The anatomy of an HTTP request to example.com.

After making the request, you receive a response code of 200 OK, which assures you that the resource you've requested exists, along with a response containing the contents of `/index.html`. The content of `/index.html` is then downloaded and interpreted by the web browser.

All of these steps incur what is called *latency*, the amount of time spent waiting for a request to reach the web server, the amount of time for the web server to collect and send its response, and the amount of time for the web browser to download the response. One of the primary aims of improving performance is to reduce latency, the amount of time it takes for a response to arrive in full. When latency occurs across a single request as in the example of example.com, it's trivial. But loading practically any website involves more than a single request for content. As these requests increase in volume, the user experience becomes increasingly vulnerable to slower load times.

In communication between HTTP/1 servers and browsers, a phenomenon known as *head-of-line blocking* can occur. This occurs because the browser limits the number of requests it will make at a single time (typically, six). When one or more of these requests are processing and others have finished, new requests for content are blocked until the remaining request has been fulfilled. This behavior increases page-load time.

HTTP/2, a new version of HTTP, largely solves the head-of-line blocking problem and enjoys wide support among browsers. The responsibility is on servers to implement the protocol, however. As of July 2016, only approximately 8.5% of all web servers are using HTTP/2 (http://w3techs.com/technologies/details/ce-http2/all/all). Because HTTP/2 has the ability to fall back to HTTP/1 for clients that don't support it, clients that understand only HTTP/1 are still susceptible to the problems of the older protocol. Moreover, any browser communicating with an HTTP/1 server will encounter the same issues, regardless of its ability to support HTTP/2.

Because we live in a complex world, we need to be able to accommodate both versions of the protocol for the time being. Going forward, we'll discuss ways to optimize sites for HTTP/1, but also call out practices that may be counterintuitive on HTTP/2. To learn more about HTTP/2, as well as how to conditionally implement the best workflows for each version of the protocol, check out chapter 11.

The next section covers how websites load content and how this behavior can lend itself to performance problems with websites.

1.1.3 How web pages load

In a boring world, all websites would be like example.com: one page with no images or JavaScript, and with minimal styling. But in reality, websites are often more complex than a single HTML file. Websites are an assortment of visual media that provides accompaniments to content, style sheets that apply design to bland markup, and JavaScript that turns static pages into applications capable of complex behaviors. It sounds neat, but these pieces come at a cost. Figure 1.4 shows a user's request to get index.html from a web server.

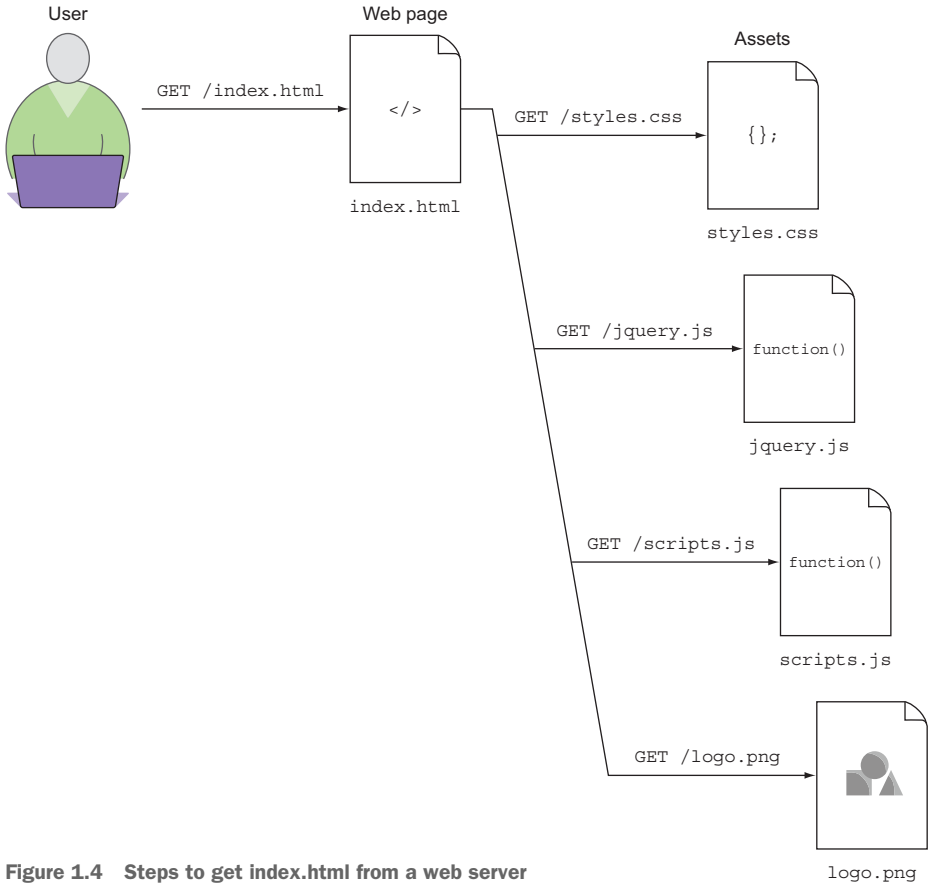

Figure 1.4 Steps to get index.html from a web server

After the browser downloads index.html, it discovers a `<link>` tag to a style sheet, a couple of `<script>` tags linking to JavaScript files, and an `<img>` tag referring to an image. When the browser discovers these references to other files, it makes new HTTP requests on the user's behalf to retrieve them. What started off as one request for a web page has now turned into five requests. Although five requests aren't much, a typical website can easily have ten times that many, or a complex one could have even a hundred or so. As these requests increase, so too does the amount of data downloaded. As requests and the data accompanying them increase, so does the amount of time it takes a page to load.

Therein lies the challenge of enhancing website performance: balancing the requirements of modern websites with the importance of serving them as fast as possible. You need to know performance-enhancement techniques so you can keep complex web experiences from encroaching on the most valuable part of the user experience: the ability to access content.

1.2 Getting up and running

Performance problems often signify issues in front end architecture. Although some issues *can* originate from a poorly configured application back end, those issues are specific to those application platforms (for example, PHP or .NET) and are admittedly outside the scope of this book. In this section, you'll investigate how to fix common performance problems through an interactive exercise that enhances the performance of a client's single-page website.

This client, Coyle Appliance Repair, is an appliance repair company from the Upper Midwest. The owners have approached you and asked whether you can make their site faster. You'll help them out by employing techniques that will decrease the load time of the website by 70% by the end of this chapter.

In this section, you'll get the client's website running on your computer. To do this, you'll use Node.js and Git. You'll also use Google Chrome to simulate a network connection to a remote server so that you can measure the results of your work in a meaningful way.

1.2.1 Installing Node.js and Git

Node.js (informally called Node) is a JavaScript runtime that allows JavaScript to be used outside the browser. It can be used for numerous things, but in this case you'll use a small Node program that runs as a local web server for running the client's website. You'll also use a couple of Node modules to achieve some optimization goals.

You'll use Node instead of a traditional web server (such as Apache) for simplicity. With Node, you can spin up a local web server quickly. It allows you to pull down exercises in this book without having to install or configure a web server. Using Node, you can pull down and run the example websites in this book in a matter of minutes, even if you have little or no experience with Node.

To install Node, go to http://nodejs.org. In the Download section, find the installer for your operating system. When running the installer, choose the standard installation option to ensure that the Node Package Manager (`npm`) is installed. `npm` provides access to the vast Node package ecosystem available on http://npmjs.com, and is required to complete the client website exercise.

You also need to install Git to pull down the client website in this chapter and the example websites later in this book. By using Git, you'll be able to grab code in this book whenever you need it from a centralized location. If you're familiar with Git, that's great, but previous experience is unnecessary for following along in this book's exercises. To download Git, head over to https://git-scm.com/downloads, choose the installer for your system, and run it. After you've installed Node and Git, continue on!

1.2.2 Downloading and running the client's website

You can download the client's website for this chapter from GitHub. To do this, download the repository into a folder of your choosing from the command line:

```
git clone https://github.com/malchata/ch1-coyle.git
cd ch1-coyle
```

This downloads the exercise files from the repository on GitHub into the current working directory on the command line. If you don't have Git installed, or you don't feel like cloning the repository, you can download the exercise as a zip file at https://github.com/webopt/ch1-coyle and extract it where you like.

After the exercise has been downloaded, you'll need to use `npm` to download the packages necessary for the web server to run. Run the following command in the same folder to download and install the needed packages:

```
npm install express
```

This command installs the Express framework to your current directory, which you can use to create a simple web server that serves static files for this and many other examples that you'll run locally on your computer. You don't need to know Express or how it works in order to follow along. None of the examples in this book makes heavy use of this framework beyond serving static files from your computer.

Permissions issues on UNIX-like operating systems

`npm` usually installs packages without a problem on most operating systems, but if you run into problems on a Mac or any other UNIX-like environment, running the `npm` command with `sudo` should clear up any permissions issues. In Windows, opening a new command line as an administrator should help.

Depending on your connection speed, the installation could take 10 or more seconds. After it finishes, you can run the following command to start the local web server:

```
node http.js
```

When you run this command, a local web server running the client website will be accessible on your computer at http://localhost:8080 and will appear as shown in figure 1.5.

Figure 1.5 The client's website in the web browser running from your local machine

If you have another service running on port 8080, you can open the http.js file in your text editor and change the port number on line 8. To stop the server from running, press Ctrl-C.

1.2.3 Simulating a network connection

Because you're running the client's website on a local machine, no latency occurs when you make requests to `localhost`. Without latency, it's difficult to measure any gains in performance, because no network bottleneck exists in this scenario.

One way to get around this is to deploy the website to a remote web server as you complete the steps, but this can be convoluted for our purposes. A better way is to use Google Chrome Developer Tools.

To get started, open Chrome. To open the Developer Tools on a Windows machine, press F12. On a Mac, press Command-Alt-I. The Developer Tools should appear within the Chrome window. Alternately, you can choose View > Developer > Developer Tools. When the Tools menu appears, click the Network tab that appears at the top of the window, as shown in figure 1.6.

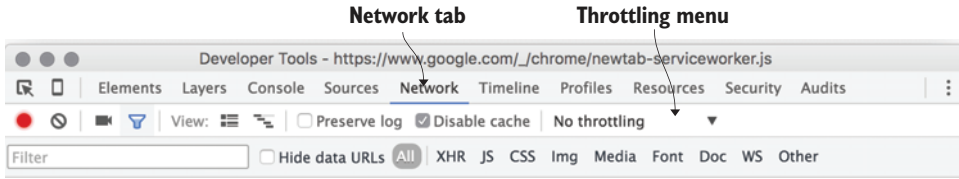

Figure 1.6 The location of the Network tab in the Google Chrome Developer Tools window. You can simulate internet connection speeds by using the throttling menu.

Near the top and to the right of the Disable cache check box is a drop-down menu labeled No throttling. This is the network throttling menu. When you click it, a list of options appears. These options allow you to simulate conditions that can be useful for performance testing. For now, select the Regular 3G profile, which simulates a slower mobile network connection.

> **Don't forget!**
> When you're finished optimizing the client's website, make sure you switch this drop-down menu back to No throttling. If you forget, all of your web browsing will be throttled to the selected setting while the Developer Tools are open.

With your client's website running and your network throttling set up, you're ready to audit the client's website and create a waterfall chart with Chrome's Developer Tools.

1.3 Auditing the client's website

To optimize a website, you have to be able to identify areas of improvement. This means analyzing the number of requests on a page, the amount of data the page contains, and the amount of time it takes for the page to load. This is where Chrome's network tools come in handy. In this section, you'll learn how to create waterfall charts with these tools and how to quantify aspects of your client's website so that you have a starting point for optimizing.

Chrome's network tools are accessible in the same place where you chose a network throttling profile, which is under the Network tab. To profile a site, the Record button in this pane must enabled, as shown in figure 1.7.

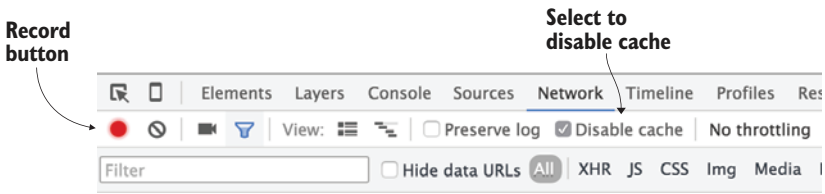

Figure 1.7 The Record button must be in the enabled state (red) before you can generate a waterfall chart of assets. The Disable Cache check box should also be selected so that no caching is done when you reload the page to measure the results of your work.

The first thing you'll want to do in the Network tab is ensure that the Disable cache check box is selected. When a website is first visited, none of the assets are cached, and this is the scenario that you want to be able to replicate. Otherwise, the site's assets will be served from the cache. Although a site loads faster when cached, it's best to assume that your average user won't have your site assets cached. For a small site such as this, this is likely.

In the Network tab, make sure the Record button in the upper-left corner is in the enabled state (see figure 1.7). It's red when enabled. If you haven't already, navigate to the client website running on your computer at http://localhost:8080 (or reload) to generate the waterfall chart. After the page is done loading, you can see the results. Figure 1.8 shows a waterfall chart for your client's website.

The waterfall chart generated for your client's site shows eight requests. Although this isn't an obscene number of requests, 536 KB of data is spread across them, and that's a significant amount for a small site like this. Because of the amount of data, the site loads in about 6.15 seconds on the Regular 3G throttling profile, which means that this site will take even longer to load on slower mobile networks than some users would like.

Because this is a responsive website, it's important to know that differences in load times will occur among devices. *Responsive websites* display differently at different screen widths because of mechanisms called *media queries* that are part of the site's CSS.

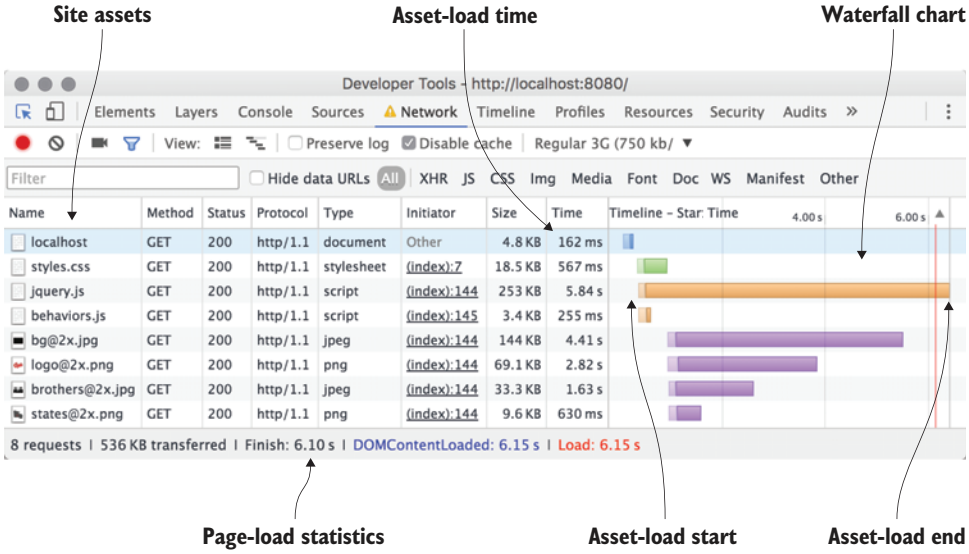

Figure 1.8 A waterfall chart generated for your client's website. At the top, you can see the request for index.html, followed by the site's CSS, JavaScript, and images. Each bar represents a request for a site asset. The bars are positioned on the x-axis according to the time they began downloading on the left, and the time they have finished downloading on the right. The length of a bar corresponds to the amount of time it takes for the asset to be requested and downloaded by the web browser.

These are covered in more detail in chapter 3, but the important point to know is that this site renders differently across three types of devices: desktop computers, tablets, and mobile phones.

More than that, screens across these devices vary not only in size, but in capabilities such as display density (the number of pixels per inch on the screen). If you've ever used an Apple product, for example, you've seen a high DPI (dots per inch) display at work. In order to retain high visual quality on these screens, a higher-resolution set of images is needed than for standard DPI displays. More information on these screen types and methods for serving images specific to them can be found in chapter 5.

Don't worry if you don't understand all this talk of CSS media queries and screen sizes right now. The point is that the client website's load time can differ not only because of the quality of its network connection, but also because of the characteristics of the device itself. Depending on the site visited, devices with higher display densities may download more data than devices with standard displays. Table 1.1 lists the amount of data transferred and website load times according to the device's type and display density.

Table 1.1 A comparison of page-load times across various devices. Results vary depending on the amount of data and the display density of the device.

Device type	Display density	Page weight	Load time
Mobile (phone and tablet)	Standard	378 KB	4.46 seconds
Mobile (phone and tablet)	High	526 KB	6.01 seconds
Desktop	Standard	383 KB	4.51 seconds
Desktop	High	536 KB	6.15 seconds

As you proceed in performance-tuning the client's website, you'll keep tabs on load times and the amount of data you reduce for each scenario as it pertains to the Regular 3G throttling profile you've chosen. Let's get to work!

1.4 Optimizing the client's website

When improving the performance of a website, the goal is simple: reduce the amount of data transferred. By pursuing this, you'll decrease the amount of time that the site loads on any device. The best part of this pursuit is that it benefits the user on both HTTP/1 and HTTP/2 servers. If there's one piece of advice that always wins out, it's this: fewer bytes transferred means faster load times.

Reducing requests can help, and some performance-boosting techniques that follow in this book will encourage you to do this, but be aware that this approach works best for an HTTP/1 workflow. This client's site is already light on requests and won't benefit much from it.

In these optimization efforts, you'll start by minifying the assets of the site, which includes the CSS, the JavaScript, and the HTML itself. Then you'll move on to optimize

Want to skip ahead?

If you get stuck at any point while working on the client's website (or you're curious to see how it all comes together), you can skip to the final, optimized code by using the `git` command. Type `git checkout -f optimized` in the root folder of the web project, and the final, optimized site will be downloaded to your computer. Be aware that performing this action overwrites any work you've done locally, so back up your work!

the images on the site without compromising their visual integrity. Finally, you'll finish by employing compression on the server for text assets.

1.4.1 Minifying assets

Minification is a process by which all whitespace and unnecessary characters are stripped from a text-based asset without affecting the way that asset functions. Figure 1.9 illustrates the basic idea of minification as it applies to CSS.

Many human-readable files such as CSS and JavaScript contain whitespace and characters that are inserted by developers during development. We use line breaks and indentation in our CSS and JavaScript to make them easier to read, as well as using comments in source code for documentation purposes.

Web browsers need no such help when reading these files. The fewer unnecessary characters that are in these files, the faster the web browser will download and parse them.

TIP When minifying files, it's important to preserve the original, unminified source. Chances are near certain that you'll have to edit files in a web project again after you minify them. Chapter 12 will help you in this endeavor.

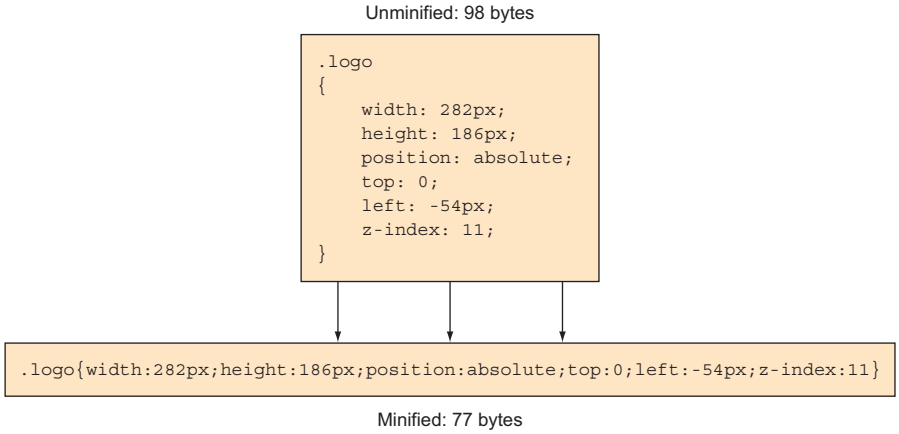

Figure 1.9 Minification of a CSS rule. In this example, a CSS rule is minified from 98 bytes down to 77, which represents a 21% reduction. When this concept is applied to all text assets on a site, the reductions can total many kilobytes.

In this section, you'll start by minifying the site's CSS, then JavaScript, and finally the HTML. Before you continue, you'll download a couple of packages by using `npm` that will allow you to minify files on the command line:

```
npm install -g minifier html-minify
```

This installation could take a minute or so. After the packages install, you'll be ready to minify the site's assets. When you're finished with this section, you'll have reduced the site's total weight by 173 KB.

Minifying the website's CSS

The site's CSS is 18.2 KB. By minifying it, you could reduce the weight of the page a bit. To minify the site's CSS, you need to do two things: run the minifier program and then update the HTML to point to the newly minified file. To minify the CSS, run this command inside the website's css folder:

```
minify -o styles.min.css styles.css
```

This command's syntax is simple. It specifies the output file (styles.min.css) with the `-o` argument. After this argument, the input filename (styles.css) is specified. After the command finishes, check the size of the output file, and you'll notice that the minified file is 14% smaller, at 15.6 KB. Not a huge savings, but it's a good start. Let's update the reference to this file in index.html by changing the `<link>` tag reference from styles.css to styles.min.css, like so:

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

Next, reload the client's website in your web browser to ensure that the website's styles still work. You can verify that the minified styles are in place by checking the updated waterfall graph and looking for a reference to styles.min.css. Your client website's CSS is now minified!

Minifying the website's JavaScript

The website's JavaScript has a much larger share of data than the CSS does. This site uses two JavaScript files: jquery.js (the jQuery library) and behaviors.js (the site's behaviors that are dependent on jQuery). These weigh in at 252.6 KB and 3.1 KB, respectively. To minify these files, you run the `minify` command on them, as you did for the site's CSS:

```
minify -o jquery.min.js jquery.js
minify -o behaviors.min.js behaviors.js
```

After the .js files are minified, check the size of the output files and compare them to the unminified versions. You'll see that behaviors.js has been reduced by 46% to 1.66 KB, and jquery.js has been reduced by 66% to 84.4 KB. This tremendous improvement knocks off a large chunk of the site's total weight (which you'll measure and compare at the end of this section).

You need to update the references to jquery.js and behaviors.js, to jquery.min.js and behaviors.min.js, in index.html. Locate the `<script>` tags that reference these files and change them to the following:

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

Then reload the page and check the Network tab to see that the minified files are referenced. If they are, you're ready to minify the last asset, which is the website's HTML.

MINIFYING THE WEBSITE'S HTML

Although not as large as the savings you've realized by minifying the site's JavaScript, the site's HTML is another asset that you can minify. Rather than using the `minify` Node package (which is intended for use with CSS and JavaScript files), you'll use the `htmlminify` package instead.

Unintended consequences of minifying HTML

Minification of HTML usually goes off without a hitch, but you may notice that minor shifts can occur to the layout. This is due to the influence of whitespace on CSS `display` types such as `inline` and `inline-block`. If you indent your HTML, these CSS `display` types could act a bit differently after the whitespace around them is removed. Some tweaking of your CSS may be necessary if the effects are dramatic. Also be aware of any properties or tags that treat whitespace literally, such as the CSS `white-space` property or the HTML `<pre>` tag.

Before you minify the site's HTML, you need to copy index.html in the site's root folder to a separate source file named index.src.html so you can preserve the original for changes. After you copy this file, you can minify it with `htmlminify`, like so:

```
htmlminify -o index.html index.src.html
```

You'll see that the minified file is 19% smaller than its original size—from 4.57 KB to 3.71 KB. Not a huge savings, but it does squeeze a bit more toothpaste out of the tube, so to speak, and for not much more effort.

With your site assets minified, you've managed to slim down your website by 173 KB. Because these assets are needed for the web page to work across all types of devices, this is a consistent performance gain for users of any device. Figure 1.10 compares load times before and after minification for all device types shown in table 1.1.

Through a modest effort, you were able to decrease load times by anywhere from 31% to 41%! This is no small improvement, and more is yet to come. In the next section, you'll further improve the yields on text assets via a server-side mechanism called *server compression*.

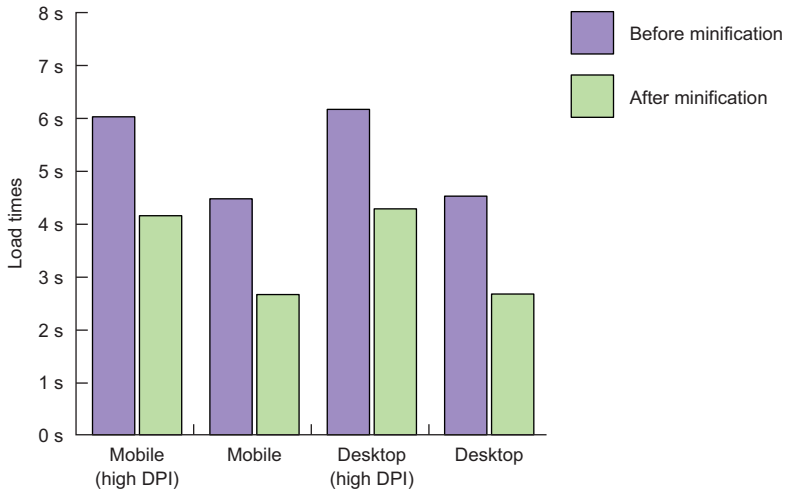

Figure 1.10 Load times of the client's website on the Regular 3G network throttling profile before and after minification. Improvements range anywhere from 31% to 41%, depending on the visitor's device.

1.4.2 Using server compression

Surely you've been emailed compressed files. These files are often used in online communications as a handy way to package multiple files into a single one. Aside from the convenience of consolidation, compressing files can also reduce their size. Server compression works on a similar principle with respect to reduction of file sizes, and web browsers are able to accept and decompress compressed content on behalf of the user. Figure 1.11 provides an overview of this concept.

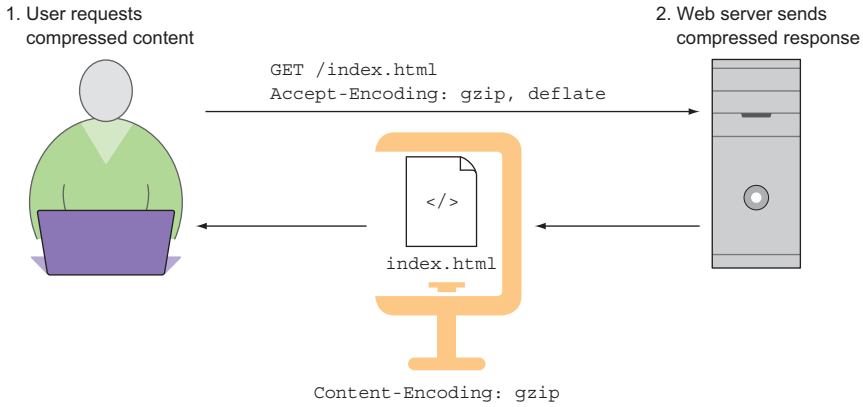

Figure 1.11 The process of server compression

Server compression works as follows: A user requests a web page from a server. The user's request is accompanied by an `Accept-Encoding` header that tells the server the compression formats the browser is capable of using. If the server is capable of encoding the content as indicated in the `Accept-Encoding` header, it will reply with a `Content-Encoding` header that describes the compression method used along with the compressed content.

This is useful because much of the content that's downloaded from websites tends to be text, which compresses well. A compression method called *gzip* has nearly universal browser support, and is *very* effective in reducing the size of text assets. In this step of optimizing your client's website, you'll configure your server to serve compressed content. As a result of these efforts, you'll reduce the weight of the page by an additional 70 KB and improve its load time by 18% to 32%, depending on the visitor's device. Before you do this, though, go to your command line and stop the web server by pressing Ctrl-C. Then type the following command to install the `compression` module:

```
npm install compression
```

After the installation finishes, open http.js in your text editor and add the bold lines that you see in this listing.

Listing 1.1 Configuring the Node HTTP server to use compression

```
var express = require("express");
var compression = require("compression");
var app = express();

// Run static server
app.use(compression());
app.use(express.static(__dirname));
app.listen(8080);
```

Compression module is imported into the script.

Script hooks the compression module into the web server.

After you've made these changes, restart the web server. Reload the page and view the waterfall graph to see the results. Table 1.2 compares text assets before and after compression.

Table 1.2 A comparison of text assets on the client's website before and after the application of server compression

Asset filename	Size before	Size after	Reduction
index.html	4 KB	1.8 KB	55%
styles.min.css	15.9 KB	3.1 KB	80.5%
jquery.min.js	84.7 KB	30 KB	64.5%
behaviors.min.js	1.9 KB	1.1 KB	42.1%
Total:	106.5 KB	36 KB	66.2%

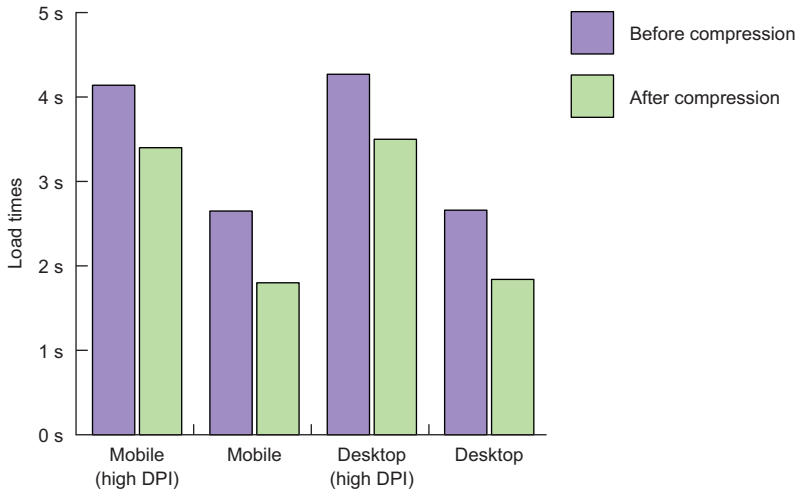

Figure 1.12 Load times of the client's site on the Regular 3G throttling profile before and after applying compression. Depending on the visitor's device, load times improve anywhere from 18% to 32%.

The reduction of file sizes is clearly significant. The size of all text assets prior to applying compression was 106.5 KB. After using compression, you were able to reduce this by about 66%, to an even lower 36 KB! So what does this do for load times? Quite a bit. Figure 1.12 compares load times across devices.

This simple step has significantly improved the site's load time. It's important to note that different web servers require different steps to configure compression for assets. The following listing shows how to enable compression for common asset media types in the software's httpd.conf configuration file.

Listing 1.2 Enabling server compression on Apache web servers

Checks if the mod_deflate module is loaded.

Compresses files that match the provided content types.

```
<IfModule mod_deflate.c>
    AddOutputFilterByType DEFLATE text/html text/css text/javascript
</IfModule>
```

In Microsoft Internet Information Services (IIS), compression can be configured by entering the admin panel via the `inetmgr` executable, going to a specific website, and editing the compression settings through the utility's GUI. No matter what kind of web server you use, the benefit of compression is largely the same. Some allow more configuration than others.

With compression applied and working on your client's website, you can move on to the final part of this optimization plan: optimizing images.

Compression pro tip

Have you ever tried to zip a JPEG or an MP3 file? Not only does this provide no additional savings, but the final zip file may end up being larger. This is because those types of files are already compressed when they're encoded. Compressing content on the web is no different. Avoid compressing file types that already use compression when they're encoded, such as JPEG, PNG, and GIF images and WOFF and WOFF2 font files.

1.4.3 Optimizing images

Image compression has come a long way since the days of Photoshop's Save for Web dialog box. Today's algorithms are so efficient at reducing the file size of full-color images that the end result is usually indistinguishable from the source image. The savings in file size, however, can be significant. Figure 1.13 compares two images, before and after optimization.

Figure 1.13 Image optimization in action on a PNG image. Optimizing images in this manner uses a re-encoding technique that discards unnecessary data from the image, but doesn't noticeably impact the image's visual quality.

If you can't notice a difference between the two images, that's the point. The idea behind this type of optimization is to retain as much visual quality as possible from the source, while discarding unnecessary data.

That's not to say that this type of optimization can't lead to undesirable results. Any optimization can go too far, leading to a noticeable loss in quality. Chapter 6 delves into image optimization not only for PNG files, but for JPEG and SVG images as well. The rule of thumb is to compare the result of any optimization to the original source, and make sure that you're satisfied with the results.

Many services can compress images for you, including some command-line and automated tools covered in chapters 6 and 12. For the sake of simplicity, though, you'll go with a web service named TinyPNG (http://tinypng.com), shown in figure 1.14.

Despite the name, this site compresses not only PNG images, but also JPEG images. Depending on the visitor's device, four images show in the desktop view, and only

Figure 1.14 TinyPNG compressing the client website's images and reporting a 61% reduction of total size

three in the mobile views. The size of these images depends on the kind of screen viewing them. High DPI screens (such as Retina screens on Apple devices) need the larger set of images to provide the best visual experience, whereas standard DPI screens can use the smaller set of images. The differences between these screens and the ways to serve them based on a device's capability are covered in chapter 5. At this point, the goal is to take whatever images are in the img folder, use the TinyPNG service to optimize them, and observe the gains.

To compress these images, upload them to the TinyPNG site, and the site will automatically optimize them. When finished, download all of them and copy them to the img folder of the website. When prompted, select the Overwrite option for any conflicts. Then reload the page and check the waterfall graph again in Chrome's Developer Tools to see the difference these smaller images have made. Table 1.3 lists images on the site before and after their optimization.

Table 1.3 A comparison of image sizes before and after their optimization using the TinyPNG web service

Asset filename	Size before	Size after	Reduction
bg.png	56.6 KB	32.0 KB	-43%
bg@2x.jpg	147.4 KB	29.4 KB	-80%
brothers.jpg	11.9 KB	9.7 KB	-18%
brothers@2x.jpg	33.8 KB	29.8 KB	-12%
logo.png	31.6 KB	12.0 KB	-62%
logo@2x.png	70.5 KB	25.2 KB	-64%
states.png	4.9 KB	1.8 KB	-63%
states@2x.png	9.6 KB	3.5 KB	-63%

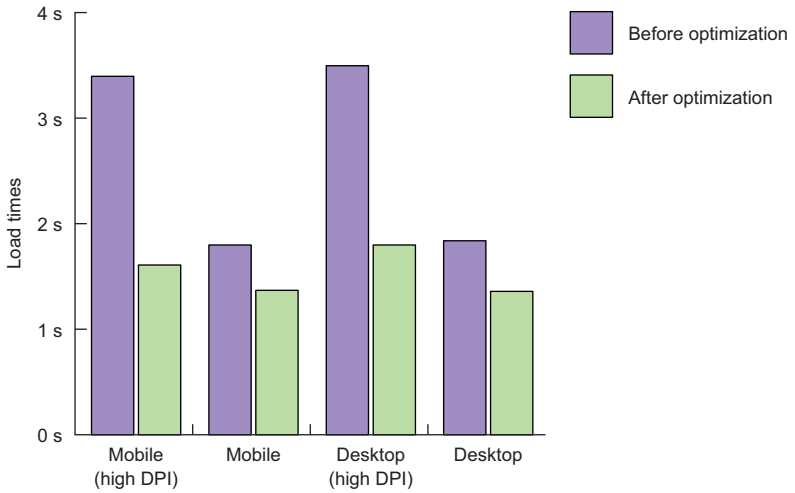

Figure 1.15 Load times of the client's website on the Regular 3G network throttling profile before and after optimizing images. Depending on the visitor's device, load times improve anywhere from 23% to 53%.

By the looks of it, all images benefit to a varying degree from this optimization—some more than others, certainly. But the real question is, how does this impact page-load time? Figure 1.15 compares load times before and after this image optimization effort.

Optimizing images has had a pronounced effect on your load times. Load times for all devices have been reduced to less than 2 seconds, which is significant, especially for 3G networks! With your work done, let's take a look at the full impact of your efforts.

1.5 Performing the final weigh-in

With your optimization efforts in the can, you can compare the amount of data transferred by the server before and after your efforts for each of the four scenarios in table 1.4.

Table 1.4 A comparison of page weights for the client's website for various device types before and after optimizations have been made

Device type	Page weight before	Page weight after	Reduction
Mobile (high DPI)	526 KB	118 KB	77.5%
Mobile	378 KB	87.4 KB	76.8%
Desktop (high DPI)	536 KB	121 KB	77.4%
Desktop	383 KB	89.5 KB	76.6%

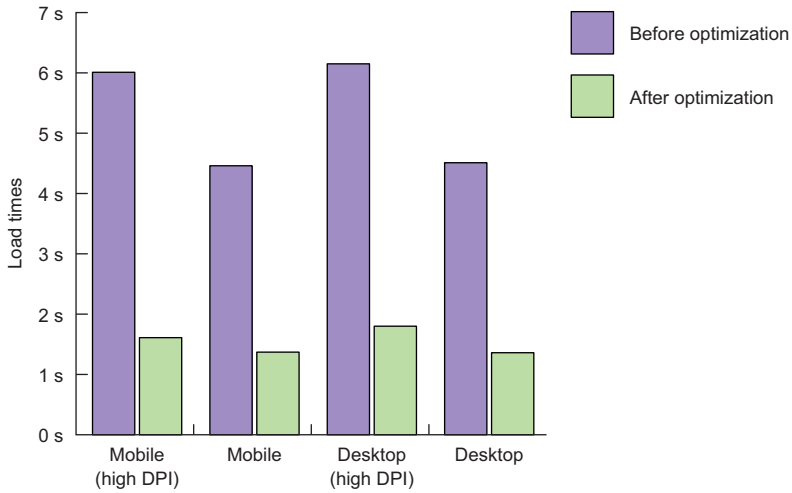

Figure 1.16 Load times of the client's website on the Regular 3G throttling profile before and after all optimizations were made. Load times improve approximately 70% for all visitors on all devices.

Of course, you'll want to see how this affects load times from end to end. Figure 1.16 compares load times before and after optimizations were made.

Your optimization efforts have improved load times for the client's website by nearly 70% for all users, regardless of which device they may be using to visit the site. As you can see, even basic performance-tuning techniques can be effective and can improve the user experience in a measurable way. We've only scratched the surface, and more-advanced tips and tricks reside in the chapters ahead.

1.6 Summary

You began this chapter by learning some high-level concepts indicating why web performance is important. You then set to work by improving a client's website through the following techniques:

- Analyzing the weight of a page by using the Developer Tools in Google Chrome
- Reducing the size of text-based assets by a process called minification, which strips unnecessary whitespace from assets without affecting their function
- Further reducing the size of these text assets through server compression
- Measuring the effectiveness of optimizing images

You're well on your way but you have far more to learn. You'll start in the next chapter by learning how to use the developer tools in various browsers to assess performance.

WEB DEVELOPMENT

Web Performance IN ACTION

Jeremy L. Wagner

Free eBook
SEE INSERT

Nifty features, hip design, and clever marketing are great, but your website will flop if visitors think it's slow. Network conditions can be unpredictable, and with today's sites being bigger than ever, you need to set yourself apart from the competition by focusing on speed. Achieving a high level of performance is a combination of front-end architecture choices, best practices, and some clever sleight-of-hand. This book will demystify all these topics for you.

Web Performance in Action is your guide to making fast websites. Packed with "Aha!" moments and critical details, this book teaches you how to create performant websites the right way. You'll master optimal rendering techniques, tips for decreasing your site's footprint, and technologies like HTTP/2 that take your website's speed from merely adequate to seriously fast. Along the way, you'll learn how to create an automated workflow to accomplish common optimization tasks and speed up development in the process.

What's Inside

- Foolproof performance-boosting techniques
- Optimizing images and fonts
- HTTP/2 and how it affects your optimization workflow

This book assumes that you're familiar with HTML, CSS, and JavaScript. Many examples make use of Git and Node.js.

Jeremy Wagner is a professional front-end web developer with over ten years of experience.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/web-performance-in-action

"An invaluable, accessible reference for the modern web developer."
—From the Foreword by Ethan Marcotte, author of *Responsive Web Design*

"An excellent and practical guide through the forest of web performance issues."
—Alexey Galiullin, Global Orange

"By far the most valuable book I have read on web performance. A true time-saver for you and your users."
—Kevin Liao, Sotheby's

"A thorough compendium of tools and techniques for improving web performance."
—Noreen Dertinger
Dertinger Informatics

ISBN-13: 978-1-61729-377-1
ISBN-10: 1-61729-377-6
54499
9 781617 293771

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]